Advanced search

Linux Journal Issue #25/May 1996

Fortran Programming Tools under Linux

Steven A. Hughes

Issue #25, May 1996

Steve Hughes proves that the Fortran Programming language's variety of high-quality programming tools and libraries provide a capability that, when coupled with all the features of Linux, makes for a potent programming platform for engineers and scientists.

Although demand for Linux support of punch card readers has been light, there are still quite a of number of us who learned to write computer code in the days when programming prowess was judged by the size of your card deck in the “360” queue. Of course, back then, Fortran was **the** high level language of choice for engineers and scientists, and it was the only programming language many of us (including myself) ever learned.

Many (well, maybe most) argue that Fortran is old-fashioned compared to modern computer languages; but Fortran-trained engineers, researchers, and scientists pay little heed to these arguments. We view programming as simply a tool to accomplish our goals, and we are most productive with familiar tools; therefore, we stick with Fortran. Fortunately for us, Linux offers a rich selection of Fortran programming tools.

This article surveys some of the basic tools available in Linux for Fortran users migrating to Linux from a non-Unix environment, such as VAX/VMS or DOS. My goal is to convince fellow “Fortran Fogeys” that they can take the Linux plunge without sacrificing their “native” programming capability. (It is worth noting that Linux users ranked Fortran fifth out of twenty-five programming languages in a recent *Linux Journal* survey.)

Compiling Fortran Programs under Linux

Several options exist for compiling FORTRAN 77 programs under Linux. The most established method is a public domain Fortran-to-C converter developed by AT&T Bell Laboratories and Bellcore. This Fortran converter, known as **f2c**, is normally included in all the popular Linux distributions. A second option is the newly released GNU Fortran compiler known as **g77** (see sidebar). Because GNU g77 is still being beta-tested and is not as widely distributed as f2c, this article is restricted to the use of f2c. Finally, there are at least two commercial Fortran compilers available for Linux: NAG's F90 Fortran 90 compiler and Microway's NDP Fortran compiler.

The f2c converter reads the FORTRAN 77 source file and converts it into equivalent C language source code. This sounds like an impossible feat, but the f2c developers have done a remarkable implementation. The second step involves compiling the resulting C code using the GNU C compiler, gcc. Included in this step is the automatic creation of an executable file with proper links to the necessary static and shared libraries.

As usual with Unix, the new user is faced with a bewildering array of options associated with f2c and gcc. Fortunately, we have been protected from the “raw” power of Unix by a bash shell script called **f77**. This script has fewer options, and it makes compiling Fortran programs easy by serving as an interface to the f2c/gcc combination. In my Slackware distribution, the f77 script is located in the /usr/bin directory. I don't have a man page for f77, though it may exist. Fortunately, all the information you need to get started with f77 is listed in the first 18 lines of the f77 script itself.

Using f77

The simple program listed below illustrates the basics of compiling Fortran programs under Linux. (I know the subroutine is not really necessary, but I have included it to illustrate some points later in the article.)

```
C===============================
C  Simple Program to Illustrate
C  Fortran Programming Tools
C===============================
      PROGRAM F77DEMO
      DIMENSION X(100), Y(100)
      PI=2.*ACOS(0.)
      N=100
      DO 10 I=1,N
      X(I)=I*(2*PI/N)
 10   CONTINUE
      CALL TRIG(N,X,Y)
      DO 20 J=1,5
      PRINT 15, X(J), Y(J)
 15   FORMAT(2X,2F8.3)
 20   CONTINUE
      STOP
      END
C
      SUBROUTINE TRIG(N,X,Y)
      DIMENSION X(1), Y(1)
      DO 10 I=1,N
      Y(I) = SIN(X(I))*EXP(-X(I))
 10   CONTINUE
      RETURN
      END
```

Compiling and linking our example program, named f77demo.f, is easy (in this article, the commands that you type are preceded by the shell prompt **$.**):

```
$ f77 -o demoexe f77demo.f
  f2ctmp_f77demo.f:
     MAIN f77demo:
     trig:
```

No errors are listed, so we know the **MAIN** program and subroutine **trig** compiled successfully. The executable file, demoexe, is ready to run by simply typing **demoexe** on the command line. (If demoexe is not in your **$PATH**, you will have to indicate where it is—in this case, by typing **./demoexe**.) In the compilation process, the f77 script also created an object module with the name f77demo.o.

The f77 script expects the Fortran source code to be in a file having the .f extension. If you are porting code having a different file extension, such as .FOR or .FTN, change it to .f. (When porting DOS files, use the fromdos program to strip line-ending Control-M and file-ending Control-Z characters. If you don't have the fromdos program, try using the shell script in Listing 1.) The **-o** switch tells the script to create an executable file having the name demoexe (in this example). Without this switch, the executable file name defaults to a.out.

More than one source file can be compiled into the executable module using f77. For example, if we break our example program into two files called maindemo.f and trig.f, our command line would be:

```
$ f77 -o demoexe maindemo.f trig.f
```

This usage might be typical for programs containing only a few subroutines. For large program development, the **make** utility can be used for more efficiency. [See *Linux Journal* Issue 6 or read the GNU make manual. —ED] The f77 script also allows you to compile Fortran, C, and assembly source files at the same time by including the source file names on the command line. This activity will be left as an exercise for the reader...

Among the several f77 command line options there are two that are particularly useful. The first is the **-c** option. This option tells f77 to create an object module, but not an executable. For example, the command

```
$ f77 -c trig.f
```

will result in the creation of the object file trig.o in your present working directory. Why would we want to do this? One reason is to create our own custom object libraries of Fortran-callable subroutines using the library manager available with Linux. Another reason is to create dynamic links between Fortran subroutines and other Linux programs (more later!).

The second important f77 switch is the **-l*lib*** option. This option is used to tell the gcc compiler to search for subroutine calls in user-supplied libraries normally not examined by gcc. Static object libraries have the naming convention **lib*mystuff*.a** where the name used in the **-l** option is the part wedged between **lib** and **.a**. Shared libraries also begin with **lib**, but have an extension similar to **.so.#.##.#**, where the #-sign is replaced by numbers corresponding to the library version.

Including libraries when compiling a Fortran program is illustrated by the command:

```
$ f77 -o foobar foobar.f -lmystuff -lmorestuff
```

During compilation the gcc compiler will search the libmystuff.a and libmorestuff.a libraries (in that order) for any unresolved subroutine and function calls. The order of the libraries is important. Suppose each library contained an object module named **gag**. Any calls to **gag** in the program foobar.f will be resolved by the **gag** object module in the first library, libmystuff.a.

It is important to remember that the **-l** option should always follow the list of source files. By default, f77 searches the f2c (libf2c.so.0) library and the math intrinsics library m (libm.so.4) before searching libraries specified on the command line. These libraries provide system calls, Fortran I/O functions, mathematical intrinsic functions, and run-time initialization.

A common gcc compiler error stems from the compiler being unable to locate a user-specified library. If this happens, you will need to determine which directories are searched for libraries. Then, either relocate your library to a valid library directory or create a symbolic link to your library from one of the library directories. You can also add directories to the library path on the f77 command line using the **-L** switch.

The above examples should be enough to get you started making use of Fortran in Linux using f77. Of course, more flexibility is offered using the f2c/gcc combination because more options are available. (f2c and gcc have good man pages, so I don't need to describe them here.) For example, the command:

```
$ f2c f77demo.f
```

creates the file f77demo.c, and:

```
$ gcc -o demoexe f77demo.c -lf2c -lm
```

creates the demoexe executable, and together they are equivalent to the f77 command shown earlier. Notice that we had to include the f2c and m libraries explicitly this time—and they must be listed in this order. Failure to include these two libraries would cause the compiler to complain. For example, leaving off the **-lm** option for our example program produces the errors:

```
Undefined symbol _exp referenced from text segment
Undefined symbol _sin referenced from text segment
```

because the references to the **sine** and **exponential** functions in subroutine **TRIG** could not be resolved.

Finally, you may be interested in trying out a newer Perl driver script called fort77. It is billed as a replacement for f77, and it includes support for debugging, a feature missing from f77. I haven't tried fort77 yet, but it is definitely on my list of things to do.

Finding Fortran Program Errors

Two methods exist for tracking down compiling errors in your Fortran source code: compiler messages and “lints”. Error messages reported by the gcc compiler are helpful, and this may be adequate for relatively simple programs. For example, suppose I made an error in the main program shown in the first example by forgetting to put in the statement:

```
 10   CONTINUE
```

that was line 11. Furthermore, assume I also messed up the array declaration in the subroutine **trig** by typing:

```
      DIMENSION X(1)
```

instead of

```
      DIMENSION X(1), Y(1)
```

(Hey, it could happen!) Attempting to compile this incorrect program, now named baddemo.f, results in the following error sequence:

```
$ f77 -o baddemo baddemo.f
f2ctmp_baddemo.f:
   MAIN f77demo:
Error on line 17 of f2ctmp_baddemo.f:
 DO loop or BLOCK IF not closed
Error on line 17 of f2ctmp_baddemo.f:
 missing statement label 10
   trig:
Error on line 22 of f2ctmp_baddemo.f:
 statement function y amid executables.
Warning on line 25 of f2ctmp_baddemo.f:
 local variable sin never used
Warning on line 25 of f2ctmp_baddemo.f:
 local variable exp never used
```

The error messages give information to help isolate the problems, but the line numbers don't always seem to correspond to the the line numbers of the original Fortran source. This makes it a little harder to track down obscure errors, especially in longer programs. Unfortunately, there doesn't seem to be an option in f77 or f2c to generate a program listing with line numbers. (That doesn't mean it can't be done!)

In addition to compiler error messages, there are several source code checkers or “lints” that can be used to help isolate errors in the source code. An easy-to-use checking program is **ftnchek**. In its simplest usage, ftnchek examines your program for a variety of potential errors, and it can make life easier by generating a program listing. ftnchek has a long list of options and a thorough man page. Remember, Fortran checking programs will not identify all the errors in your program. However, the combination of a checker and f77 error messages should help you combat compilation errors. (Of course, careful programming will help as well!)

The hard-core (and really adventurous) programmer can obtain a package called **Toolpack** from the usual Linux sites. This large package is a set of programs and C shell scripts that provides rigorous FORTRAN 77 code checking, along with static and dynamic analysis. See the Fortran_FAQ (directory /usr/doc/faq/lang in the Slackware distribution) for a description of **Toolpack**.

Fortran Tools and Libraries

Fortran's main usage has been in the scientific and engineering fields, and because the language has survived for decades, thousands of high-quality programs and subroutine libraries exist, many of which are freely available. These programs include application programs written for specific purposes, mathematical subroutine libraries, general purpose run-time routines, and graphical plotting and display packages.

Describing even a small percentage of available programs is impossible, but you can get a rough idea of what's out there by pointing your Web browser at netlib2.cs.utk.edu. This site is the Netlib Repository at UTK/ORNL, and it archives over 100 packages containing mathematical software, papers, and databases. Fortran programmers will be particularly interested in a “code motherlode” collection called **slatec**. This is a “...comprehensive library containing over 1400 general purpose mathematical and statistical routines written in FORTRAN 77.” (Source code, folks!) To give an idea of the **slatec** library's magnitude, its Table of Contents takes up 222,161 bytes on my disk.

One key factor in my decision to use Linux as a Fortran programming platform was the **PGPLOT** package. This highly versatile Fortran library provides 100 primitive and higher level subroutines for drawing scientific graphs on various graphic display devices. For example, with the PGPLOT library, you can create multiple graphs in multiple X windows, output plots to PostScript and other supported printing devices, or create files that are compatible with HPGL format or Latex picture environment.

In addition to Linux, PGPLOT is available for twelve other flavors of Unix (AIX, Cray, HP, SGI, NeXT, etc.), two versions of OpenVMS, and MS-DOS using Microsoft Power Station 32-bit Fortran. This wide availability is an attractive feature if you want to develop consistent Fortran applications across platforms. I am also informed that PGPLOT capabilities are available in a compiled form (PGPERL) for use with Perl scripts.

A simple demonstration of PGPLOT is provided by the program listed below. This program is the same as the one given previously, with nine (indented) lines of code added to create a simple plot.

```
C==============================
C  Simple Program to Illustrate
C  PGPLOT Graphic Tools
C==============================
      PROGRAM PGDEMO
        INTEGER PGBEG
      DIMENSION X(100), Y(100)
      PI=2.*ACOS(0.)
      N=100
        IER = PGBEG(0,'?',1,1)
        IF (IER.NE.1) STOP
        CALL PGSCRN(0, 'AntiqueWhite', IER)
        CALL PGSCRN(1, 'MidnightBlue', IER)
      DO 10 I=1,N
      X(I)=I*(2*PI/N)
 10   CONTINUE
      CALL TRIG(N,X,Y)
        CALL PGENV(0., 4., 0., .4, 0, 1)
        CALL PGLAB('X Values', 'Y Values', 'PGPLOT Demo')
        CALL PGLINE(100, X, Y)
        CALL PGEND
      DO 20 J=1,5
      PRINT 15, X(J), Y(J)
 15   FORMAT(2X,2F8.3)
 20   CONTINUE
      STOP
      END
      SUBROUTINE TRIG(N,X,Y)
      DIMENSION X(1), Y(1)
      DO 10 I=1,N
      Y(I) = SIN(X(I))*EXP(-X(I))
 10   CONTINUE
      RETURN
      END
```

Subroutine **PGBEN** in the pgdemo program performs the plot initialization. Placing the **?** character in the **PGBEN** parameter list causes the program to query for which output device to use. The two calls to **PGSCRN** simply change the background and foreground colors (and there are plenty colors from which to choose). **PGENV** establishes the limits of the x- and y-axis, and **PGLAB** labels the axes and plot. (Math/Greek symbols and subscript/superscript capabilities are available.) The generated curve is plotted by **PGLINE**, and the plot is completed by **PGEND**.

This PGPLOT program is compiled by the command:

```
  $ f77 -o pgdemo pgdemo.f -pgplot -lX11
```

It is necessary to include the X11 library after the PGPLOT library because some of the PGPLOT subroutines create X windows and control their attributes.

Program execution results in the messages shown in Figure 1.

The pgdemo program first queried for which output device to use. I answered with a **?** to see the list of available devices (which is configured when the package is installed). From the list I then selected **/XWINDOW**, and the simple plot shown in Figure 2 was drawn in an X window. PGPLOT supports a long list of output devices, but not all are available for Linux users.

Now the fun can really begin by selecting font sizes, axis types, alphanumeric notations, and a host of other options. If you have had any previous experience with Fortran plotting routines, PGPLOT will be easy to learn and use.

Using Fortran with SciLab

SciLab is one of my favorite programs running under Linux. If you deal with matrix or vector data, signal analysis, nonlinear optimization, plotting, or other mathematical manipulations, you owe it to yourself to explore this feature-packed program. (SciLab was reviewed in *Linux Journal*, Issue 11, and it is available for a variety of other computer platforms, including Sun Sparc station, IBM RS 6000, HP 9000, DEC Mips, and DEC Alpha.)

Beyond the hundreds of built-in or supplied mathematical functions, SciLab users can also dynamically link their own Fortran and C subroutines to the SciLab binary without recompiling the SciLab source code. The linked subroutines are then available for calling from within SciLab using either interactive commands or by executing scripts.

This important feature allows Fortran users (and C programmers) to make use of “tried and true” source code without the trouble of converting the subroutines to equivalent SciLab macros and scripts that use built-in SciLab functions. Just as important, tedious debugging can be kept to a minimum (provided the linked subroutines have already been tested thoroughly).

Returning once again to our original trivial example code, let's link the subroutine **TRIG** into a SciLab session. First, we need to compile the **TRIG** object module using the f77 command:

```
  $ f77 -c trig.f
```

which produces trig.o in the present working directory. From within Scilab we link trig.o using the **link** command:

```
-->link('trig.o','trig')
 linking  "trig_" defined in "trig.o "
 lastlink 0,0
```

The first argument string in the **link** command is the name (case sensitive) of the Fortran object module. If this module is not in the current SciLab working directory, you must include the path. The second argument string must be the exact name of the Fortran subroutine being linked; however, case is not important. (Note that SciLab variables are case-sensitive, so subsequent use of **trig** within SciLab requires that you use lower case.)

Other subroutines also can be linked in the same way, and SciLab lists all linked subroutines when the **link** command is issued without arguments, i.e.:

```
-->link()
 ans  =
 trig
```

Now we are ready to use **trig** in our SciLab session, but first we need to specify the two input variables **(n,x)** for the **trig** subroutine. Below are the commands issued with Scilab echoing the results.

```
-->n=5
 n  =
    5.
-->x=[.1 .2 .3 .4 .5]
 x  =
!    .1     .2     .3     .4     .5 !
```

Actually calling **trig** is done using SciLab's **fort** command as illustrated below (along with the result) for our example:

```
-->y=fort('trig',n,1,'i',x,2,'r','out',[1,5],3,'r')
 y  =
! .0903330 .1626567 .2189268 .2610349 .2907863 !
```

Okay, now for the explanation. On the left-hand side of the entered expression is a list of the subroutine output variables (**y** in this example). The arguments inside the fort expression consist of three groups. First is the called subroutine name (**trig**) as a string variable. This is followed by a list of the subroutine input variables, given in this example as:

```
n,1,'i', x,2,'r'
```

The first three items are the input variable **(n)**, its position in the **trig** subroutine argument list **(1)**, and a string character representing the variable's type (**i** for integer). Likewise, the second input variable is the real array, **x**, positioned as the second argument in **trig**. This pattern continues until all the input variables are listed. Variables in the input list do not have to be listed in any particular order. In other words, we could have just as easily listed the inputs as:

```
x,2,'r', n,1,'i'
```

Once all the input variables are listed, we can specify the output variable or variables, denoted in this example by:

```
'out',[1,5],3,'r'
```

The key word **'out'** always appears, followed by a matrix notation informing SciLab that the output variable, **y**, is a 1x5 array. The **3** gives the position of **y** in the subroutine's argument list, and **r** states that the variable is type real.

Subroutines having more than one output variable simply need to list the parameters associated with each variable on the output side in the **fort** argument list and include each variable on the left-hand side of the expression. For example, assume we have a Fortran subroutine given by:

```
SUBROUTINE WIN95(IDEAS,BUGS,DOS,DELAY)
REAL*4 BUGS(1,1), DOS, DELAY(1)
INTEGER*2 IDEAS
.
.
.
RETURN
END
```

The input variables are **DOS** and **IDEAS**, and the outputs are **BUGS** and **DELAY**. After compiling and linking this subroutine into SciLab, it is called by:

```
-->bugs,delay]=fort('win95',dos,3,'r',ideas,1,'i',
                  'out'[99,99],2,'r',[1,10],4,'r')
```

It's as easy as that!

When coupling Fortran subroutines with SciLab, any **print** statements within your Fortran subroutines will not print to the SciLab session. Instead they print to the X window from which you started SciLab (or on the console monitor if you loaded Scilab from a window manager, such as FVWM). More importantly, the dynamically linked Fortran subroutines can **open, read, write**, and **close** disk files. Subroutines containing **COMMON** blocks must be restructured to accept all the variables and constants through the **fort** command line.

Conclusions

The Fortran programming language is reasonably well supported in the Linux environment. Furthermore, a variety of high-quality programming tools and libraries provide a capability that, when coupled with all the features of Linux, makes for a potent programming platform for engineers and scientists.

In the future we can expect a robust g77 compiler with debugger support, continued improvement in existing support libraries, and release of new Fortran tools. Perhaps even more exciting is work being done by the Linux-Lab Project. This project is developing drivers to support acquisition of laboratory and field data using Linux. Higher-level interface to most hardware devices will be via C language libraries, which (we hope) will also be callable from our Fortran programs.

So take the plunge, you Fortran fanatics! It will be an exciting adventure!

Thanks to Tony Dalrymple, Rod Sobey, and Gary Howell for helpful comments on this article.

Dr. Steven Hughes (s.hughes@cerc.wes.army.mil) is a senior research engineer at the Coastal Engineering Research Center in Vicksburg, Mississippi. His research activities focus on water wave kinematics, scour at breakwaters, and laboratory methodologies. He switched to Linux in October 1994 (kernel 1.1.54), but he admits to writing his first Fortran program over 25 years ago using FORTRAN 66 (or maybe that was FORTRAN 1.0?). Cycling and two teenage daughters keep Steve and his wife fit and frenzied, respectively.

Compile C Faster on Linux

Christopher W. Fraser

David R. Hanson

Issue #25, May 1996

Many people who love the GNU gcc compiler still think that it is too slow in normal use, or that it uses too much memory.

lcc is a small, fast C compiler now available on Linux. A perfectly good C compiler, gcc, comes with Linux. Why would anyone bother installing a second one? Because the two compilers make different tradeoffs, so they suit different stages of the development cycle. gcc has many targets and users, and it includes an ambitious optimizer. lcc is 75% smaller (more when counting source code), compiles more quickly, and helps prevent some porting bugs.

For those who have always wanted to customize or extend their compiler, our recent book, *A Retargetable C Compiler: Design and Implementation*, tours lcc's source code in detail and thus offers especially thorough documentation. Pointers to lcc's source code, executables, book, and authors appear at the end of this article.

Speed Tradeoffs

lcc is fast. gcc implements a more ambitious global code optimizer, so it emits better code, particularly with full optimization options, but global optimization takes time and space. lcc implements a few low-cost, high-yield optimizations that collaborate to yield respectable code in a hurry.

For example, lcc compiles itself in 36 seconds on a 90 megahertz Pentium running Linux. gcc takes 68 seconds to compile the same program (the lcc source) with the default compiler options, and 130 seconds with the highest level of optimization. Code quality varied less. gcc's default code took 36 seconds to reprocess this input, just like lcc's code. gcc's best code (that is, with optimization level 3) runs in 30 seconds, about 20% faster. This is only a single data point, and both compilers evolve constantly, so your mileage may vary. Naturally, one can save time by using lcc for development and optimizing with gcc for the final release build.

Porting Code

Indeed, compiling code with two different compilers helps expose portability bugs. If a program is useful and if the source code is available, sooner or later someone will try to port it to another machine, or compile it with another compiler, or both. With a new machine or compiler, glitches are not uncommon. Which of the following solutions will net you less unwanted e-mail? For you to find and erase these blots while the code is fresh in your mind? Or for the porter to get diagnostics much later, about non-standard source code?

lcc follows the ANSI standard faithfully and implements no extension. Indeed, one option directs lcc to warn about a variety of C constructs that are valid but give undefined results, and thus can behave differently on a different machine or with a different compiler. Some programmers use lcc mainly for its strict-ANSI option, which helps them keep their code portable.

Cross Compiling

Like gcc, lcc can be configured as a cross-compiler that runs on one machine and compiles code for another. Cross-compilers can simplify life for programmers with multiple target platforms. lcc takes this notion a step further than most cross-compilers: we can, and typically do, link code generators for several machines into each version of the compiler.

For example, we maintain code generators for the MIPS, SPARC, and X86 architectures. We both work on and generate code for multiple platforms, so it's handy to be able to generate code for any target from any machine. We usually fold all three code generators into all compiler executables. A run-time option tells lcc which target to generate code for. If you don't maintain code for multiple targets, you're free to use an lcc that includes just one code generator, saving roughly 50KB for each code generator omitted.

A Compact Compiler

lcc is small. lcc's Linux executable with one code generator is 232 KB, and its text segment is 192 KB. Both figures for the corresponding phase of gcc (cc1) exceed a megabyte. lcc's small size contributes to its speed, especially on modest systems. A compact program benefits those who wish to modify the compiler. Most developers will use pre-built executables for lcc; they will never examine or even recompile the source code. But the Linux community particularly prizes the availability of source code, partly because it allows users to customize their programs or adapt them for other purposes.

When configured with the Linux PC code generator lcc is 12,000 lines of C source code. gcc's root directory—without the target-specific description files—holds 240,000 lines. Surely, some of this material is not part of the compiler proper, but the separation is not immediately apparent to those who haven't browsed gcc's source recently. The machine-specific module is the part most often changed, because new target machines come along more often than, say, new source languages. The lcc target-specific module for the Linux PC is 1200 lines, and half of that repeats boilerplate declarations or supports the debugger, so the actual code generator is under 600 lines. The target-specific modules for gcc average about 3000 lines. These comparisons illustrate the fact that the two compilers embody different trade-offs and that neither beats the other at everything: gcc can emit better code and offers many options, while lcc is easier to comprehend but is otherwise less ambitious.

gcc and lcc use retargetable code generators driven in part by formal specifications of the target machine, just as a parser can be driven by a formal grammar of its input language. gcc's code generator is based in part on techniques that one of us (Fraser) originated in the late 1970's. lcc uses a different technique that is simpler but somewhat less flexible.

An Adaptable Tool

Its compact source code helps people adapt lcc. An early adaptation by one of us (Hanson) injected profiling code that counts executions for a profiler that comes with lcc. For example, the lines

```
for (<1965>r = 0; <15720>r < 8; <15720>r++)
   if (<15720>rows[r] &&
       <5508>up[r-c+7] &&
       <3420>down[r+c]) { ...
```

annotate source code from the lcc test suite's implementation of a program that finds all ways to place eight queens on a chessboard such that none of them attacks any other. The numbers in angle brackets report how many times the following code fragment was executed, which can differ within one line. This profiler would have been a big project without lcc, but it was modest one with it. Other adaptations of lcc include interpreters (www.cit.gu.edu.au/~sosic/papers/sigplan92.ps.Z), code generators for multiple targets (ftp://ftp.cs.princeton.edu/pub/lcc/contrib/), a C++ compiler, programmable debuggers that can debug across a network (http://www.cs.purdue.edu/homes/nr/ldb/), and a retargetable optimizing linker (http://www.cs.princeton.edu/~mff/mld/). A group at Stanford University has adapted lcc for use with a global optimizer (suif.stanford.edu/suif/suif.html). At least some of these efforts chose lcc over gcc because lcc's small size made it seem easier to comprehend and change. Many of these projects were begun before the lcc book was done; we expect even more adaptations now that extensive documentation is available.

Literate Programming

Most developers will use pre-built executables for lcc and never study the source code. The Linux community, however, expects source, and lcc provides an annotated version of most of its code in the form of a book. lcc's annotations, like its small size, are designed to help developers modify lcc.

lcc is written as what Knuth has termed a “literate program”, which interleaves the source code with prose explanations. Two programs process this input. One program extracts just the C source code, which can be compiled with any C compiler. The other program processes both the prose and the code and emits the typescript for the lcc book. We generate the book and the compiler from a single source because it's too easy for multiple sources to get out of sync with one another.

A brief fragment of the chapter on the X86 code generator demonstrates literate programming:

> Static locals get a generated name to avoid other static locals of the same name:
>
> ```
> <X86 defsymbol>=
> if (p->scope > LOCAL && p->sclass == STATIC)
>    p->x.name = stringf("L%d", genlabel(1));
> ```
>
> Generated symbols already have a unique numeric name. Defsymbol simply prefixes a letter to make a valid assembler identifier:
>
> ```
> <X86 defsymbol>+=
> else if (p->generated)
>    p->x.name = stringf("L%s",p->name);
> ```

Each of the two displays above consists of a “fragment label” in angle brackets and a “fragment” of C code. The fragment label names the piece of the C program being described (here the version of the routine defsymbol for the X86). The **+=** in the second fragment says that the second code fragment is appended to the first.

This example is necessarily tiny, but it shows how literate programming allows one to build up a complex program a bit at a time, explaining it on the way. The lcc distribution includes conventional C code that can be modified as usual, but when some explanation would help, one can easily get it from the annotated code in the book.

Not shown in this sample are page numbers in each fragment that point to adjoining fragments, and miniature indices in the page margin that point to the page that defines each identifier that's being used. Many readers have identified these mini-indices as especially helpful.

Availability

lcc's C source code and Linux executables are available for anonymous ftp at URL ftp://ftp.cs.princeton.edu/pub/lcc/. It's about a megabyte, so it can be downloaded using even, say, a 14.4kbaud modem in about 10 minutes. The package includes Dennis Ritchie's preprocessor for ANSI C, but lcc is also used with gcc's preprocessor. Like gcc, lcc emits assembler code for the standard Linux assembler, debugger, and C library, so the package does not include any of these. A sub-directory collects code generators and other companion software contributed by others. The package describes mailing lists for communicating with others working on, and with, lcc.

Our book about lcc, *A Retargetable C Compiler: Design and Implementation*, (ISBN 0-8053-1670-1) is available from Addison Wesley at 800-447-2226 and from other sources listed on lcc's home page (http://www.cs.princeton.edu/software/lcc/).

lcc is free for non-commercial use. The lcc book amounts to a single-user license for lcc, so some have arranged commercial use by simply including a copy of the book with their product (and charging for it); the publisher offers substantial discounts. Other arrangements are possible.

Chris Fraser (cwf@research.att.com) has been writing compilers since 1974. He earned a Ph.D. in computer science at Yale in 1977 and does computing research at AT&T Bell Laboratories in Murray Hill, New Jersey.

Dave Hanson (drh@cs.princeton.edu) is Professor of Computer Science at Princeton University. His research interests include programming language design and implementation, software engineering, and programming environments. His Web is at: www.cs.princeton.edu/~drh/.

Advanced search

Introduction to Gawk

Ian Gordon

Issue #25, May 1996

For many simple programming problems, awk is an excellent solution. Let Ian Gordon show you how to make your life easier.

How often have you thought to yourself, “I should write a program to do that!” only to realize that you will have to write more than just the code needed to solve the problem at hand? Your program will probably need to get the names of data files from the command line, open and read these files, and allocate and manage memory for data storage. This programming overhead can be a lot of effort to write and debug. To make this programming task even less appealing, what if you need this program “right now” and it may be used only once or twice? Does writing this program still seem worth all the effort? If you are using one of the more traditional languages, such as C or C++, perhaps not. However, the awk programming language may be just the right tool for writing the programs you need while minimizing the programming overhead.

gawk, the GNU version of the powerful awk programming language, lets you concentrate on writing the code to solve the problem at hand without worrying about all the overhead required to actually make your program do its job. gawk offers many features designed to help you quickly write useful and powerful programs. With features such as pattern-matching, associative arrays, automatic handling of command-line argument files, and no need for variable declarations, gawk is able to free you from many of the tiresome details that often get in the way of getting the job done.

gawk is suitable for a wide range of applications, from simple, one-line applications to complex applications that will be used on a regular basis. gawk is also a simpler, easier to use alternative to Perl. Although Perl programs will run faster than comparable gawk programs, the syntax and features of gawk are (in my opinion) easier to read and tend not to become quite so obfuscated.

C programmers will find that parts of gawk are already quite familiar to them. In many ways, the syntax of gawk looks very much like the syntax of C, with constructs such as pre- and post-increment and decrement operators, nestable if-else blocks, for loops which look exactly like those in C—even the familiar **{** and **}** defining sections of code. This close similarity to C is not such a surprise when you consider that one of the originators of the awk programming language, Brian Kernighan, was also one of the originators of C.

However, beyond this similarity in syntax, awk is a language quite unlike the traditional languages in most common use today.

In this article I will describe the more basic features of working with gawk, the GNU version of awk. There will be many parts of this language that I cannot cover here—for these you will need to consult one of the sources listed in the reference section at the end. Although I will be describing gawk, the features discussed here should be applicable to most versions of the awk programming language. As such, the names gawk and awk are often used interchangeably.

In keeping with the tradition set by countless authors writing about a programming language, here is the ever-popular “Hello World” program written in awk:

```
BEGIN { print "Hello World" }
```

Before I explain how to run this program, I will describe how a gawk program, or script, works.

Pattern Matching

A major difference between gawk and most other languages is that gawk is a pattern-matching language. That is, gawk scans its input looking for patterns which have been specified in the gawk program, and executes the block of gawk code associated with that pattern. A gawk program, or script, consists of one or more patterns which the programmer wishes to match against each line of input, and the corresponding action blocks (enclosed between **{** and **}**) which are to be executed when that pattern is found in an input line. So a gawk program has the form:

```
pattern1 { action1 }
pattern2 { action2 }
 .
 .
 .
patternN { actionN }
```

These patterns, which can consist of a simple expression, a regular expression, a combination of patterns, or even an empty pattern, can be as simple or as complex as needed. To print all lines in a file which contain the word “Linux”, the pattern is simply defined as **/Linux/** and the action block is **{print}**. Thus, the complete gawk program can be written as:

```
  /Linux/ { print }
```

Action blocks consist of one or more gawk statements enclosed between **{** and **}**. In this simple example, the print statement will print everything on each line which contains the pattern "Linux". However, this program will also match such words as "LinuxKernel"--the pattern does not have to be a discrete word. Also, since pattern matching is case-sensitive by default, it will not match the pattern "linux".

If you need to match both upper and lower case, the pattern can be changed to allow for this—it just becomes a more complex pattern. If you wanted the pattern to match both "linux" and "Linux", you could write the pattern as **/[Ll]inux/**. In this case, you are telling gawk to look for groups of characters that begin with any of the characters enclosed in the square brackets (here, either an upper or lower case "L") followed by the lowercase letters "inux". Other options for dealing with case sensitivity are to use the built-in functions **tolower()** or **toupper()** to change the case of the input line (or just parts of the line) before the pattern matching takes place, or you can set the built in variable **IGNORECASE** (in awk, built in variables are always written in upper case) to any non-zero value at the start of your program.

Patterns in gawk can be as simple or as complex as needed to match the desired item in the input line. If you do not specify a pattern, the action block will be executed for every line of input. This is known as an empty pattern. So if you do not explicitly put a pattern into your program, gawk treats the lack of a pattern as a pattern that will match everything in the input.

Alternatively, if you specify a pattern but no action, gawk will provide a default action—namely **{print}**--for you. So the simple program above can be rewritten as **/Linux/**, although it is usually better to define an action explicitly, since this results in more readable code.

gawk also defines several special patterns which do not match any input at all, the most commonly used being **BEGIN** and **END**. The action block associated with **BEGIN** will be executed only once, before gawk starts to read the input files, and allow you to take care of any setup and initialization details that may be needed. The action block for the **END** pattern will be executed after the processing of all input has been completed and is useful for printing any final results from your program. The **BEGIN** and **END** patterns are optional—you include them only when there is a need for them.

However, if you wish to write a gawk script that takes no input at all—say for example, the ever-popular “Hello World” program that was shown earlier—your gawk statements must be enclosed in the action block for the **BEGIN** pattern. Otherwise, gawk will see them as part of the main input loop block (described next) and wait for some input (or a Control-D) before printing—probably not what you want to happen in this case.

Main Input Loop

Work in almost any programming language and you will have to write code to get the names of any files from the command line, open these files, and read their contents. For most file access, gawk let you skip these steps entirely. If you pass one or more file names on the command line, after executing the code in the BEGIN block (if present), gawk will automatically get the name from the command line, open a file, read its contents line-by-line, try to match any pattern you have defined against these lines, close the file when it is finished, and move onto the next file listed. If the input is coming from standard input (i.e., you are piping the output of another program to your gawk program), the input process is equally transparent. However, if you find that you need to handle this file input in some different manner, gawk provides you with all the tools necessary to do this. But for most of the file handling you will need, it is better to let gawk's input loop do the work for you.

Running a gawk Program

Now that we have seen how a gawk program works the next step is to see how to make your program run. With gawk on Linux, we have three ways to do this. For those truly quick-and-dirty tasks, an entire gawk program can be written and executed on the command line, although this is really only practical for very small programs. Using our simple example from above, we can run it with the command:

```
gawk '/Linux/ {print}' file.txt
```

When running a gawk script from the command line, you must enclose the awk statements in single quotes and list any data files after the closing quote. If you need to use more that one gawk statement in an action block, simply separate each statement using the semicolon. For example, if you wanted to print each line that contained “Linux” and keep a count of how many input lines contain the pattern **/Linux/** you could write

```
gawk '/Linux/{ print; count=count+1 }
END { print count " lines" }' file.txt
```

You can list any number of data files on the command line and gawk will automatically open and read them, looking for any lines which match the pattern defined.

You can also use your favourite editor to write your gawk program and pass the name of the file to gawk using the **-f** option to tell gawk to try to execute the contents of that file. (For convenience, I like to use the extension ".awk" on these files, although this is not necessary.) So if the file linux.awk contains the pattern-action block:

```
/Linux/ {
    print
    count = count + 1
}
END {
    print count "lines found."
}
```

It can be executed by the command:

```
gawk -f linux.awk file.txt anotherfile.txt
```

Under Linux (and other versions of Unix) there is another, easier way to run your gawk program—simply put the line

```
#!/usr/bin/gawk -f
```

at the top of the program to indicate the path to the gawk interpreter. Make the file executable using the chmod command--**chmod +x linux.awk**. Then we can execute the gawk program by typing its name and any parameters. (Note: you will need to check the actual location of the gawk interpreter on your system and put this path in the first line.)

Input Fields

Another powerful and time saving feature of gawk is its ability to automatically separate each input line into fields, each referred to by number. The entire line is referred to as **$0** and each field within the current line is **$1**, **$2**, and so forth. So if the input line is **This is a line**,

```
$0 = This is a line
$1 = This
$2 = is
$3 = a
$4 = line
```

Likewise, the built-in variable **NF**, which contains the number of fields in the current input line, will be set to 4. If you try to refer to fields beyond **NF**, their value will be **NULL**. Another built-in variable, **NR**, contains the total number of input lines that awk has read so far.

As an example of the use of these fields, if you needed to take the contents of a file and print it out, one word per line (useful if you want to pipe each word in a file to a spell checker), simply run this script:

```
{ for (i=1;i<=NF;i++) print $i }
```

To separate the line into fields, gawk uses another built in variable, **FS** (for “field separator”). The default value of **FS** is **" "** so fields are separated by white space: any number of consecutive spaces or tabs. Setting **FS** to any other character means that fields are separated by *exactly one* occurence of that character. So if there are two occurences of that character in a row, gawk will present you with an empty field.

To get a better idea of how **FS** works with input lines, suppose we wanted to print the full names of all users listed in /etc/passwd, where the fields are separated by **:**. You would need to set **FS=":"**. If the file names.awk contains the following gawk statements:

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

and you run it with **gawk -f names.awk /etc/passwd**, the program will separate each line into fields and print field 5, which in this case is the full name of the user. However, the line **FS=":"** will be executed for each line in the data file—hardly efficient. If you are setting **FS**, it is usually best to make use of the **BEGIN** pattern, which is run only once, and rewrite our program as:

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

Now the line **FS=":"** will be executed only once, before gawk starts to read the file /etc/passwd.

This automatic splitting of input lines into fields can be used to make patterns more powerful by allowing you to restrict the pattern matching to a single field. Still using /etc/passwd as an example, if you wanted to see the full name of all users on your Linux system (field 5 of /etc/passwd) who prefer to use csh rather than bash as their chosen shell (field 7 of /etc/passwd), you could run the following gawk program:

```
# (in awk, anything after the # is a comment)
# change the field separator so we can separate
# each line of the file /etc/passwd and access
# the name and shell fields
```

```
BEGIN { FS=":" }
$7 ~ /csh/ {print $5}
```

The gawk operator **~** means “matches”, so we are testing if the contents of the seven field match **csh**. If the match is found, then the action block will be executed and the name will be printed. Also, remember that since patterns match substrings, this will also print the names of tcsh users. If a particular input line does not contain a seven field, no problem—no match will be found for this pattern. Similarly, the pattern **$7 !~ /bash/** will run its action block if the contents of the seven field do **not** match the pattern **bash**. (Unlike the match operator, this pattern will match if $7 does not exist in the current input line. Recall that if we try to access a field beyond NF, its value will be NULL, and NULL does not match /bash/, so the action block for this pattern will be executed.)

To further demonstrate the power of fields and pattern matching, let's go back to the problem of dealing with case sensitivity in pattern matching. By using a built-in function, **toupper()** or **tolower()**, we can change the case of all or selected parts of the input line. Suppose we have a data file containing names (the first field) and phone numbers (the second field), but some names are all lower case, some are all upper case and some are mixed. We could simplify the matching by modifing the pattern to:

```
toupper($1) ~ /LINUX/ {print $0}
```

This will cause the name in field 1 to be converted to upper case before awk tries to match it against the pattern. No other parts of the input line will compared against the pattern.

Control Structures

The control statements in the gawk language closely resemble those found in C, thus making gawk more easily written and understood by C programmers. gawk contains the pre- and post-increment and decrement operators **++** and **--**, as well as an if-else statement that looks very much like the one found in C. Also multi-line blocks of code are grouped within **{** and **}**. Even the for loop seems to have been taken right out of a C programming book.

This allows you to “mix and match” code which takes advantage of gawk's pattern matching with code that uses more traditional control structures, so if patterns are not sufficient for your task (or you are not sure how to use them to accomplish your task) you can use standard programming techniques as well. Conventional programming with gawk is not covered here; the gawk info page (run **info gawk**) documents this well, and the goal of this article is to demonstrate gawk's distinguishing features.

Variables and Arrays

Another timesaving feature of gawk is that there is no need to declare a variable before using it. A variable can be a string, an integer, or a floating point number depending on the value assigned to it. gawk will handle conversions for you automatically. As a result, an expression such as **total = 2 + "3"** is valid and will give the expected result, **5**. To make your job even easier, gawk will initialize each variable when it is used for the first time, setting it to **0** for an integer or **""** for an integer or a string, respectively. This takes away any worries about uninitialized variables.

gawk also carries this ease of use of variables to arrays. There is no need to declare an array before using it, or even to specify a maximum size for that array. To create an array, simply use it and gawk will allocate the required space for you. As you add more data to the array, its size will automatically expand to accomodate it.

However, the array indices in gawk differ from those in languages such as C, in that gawk indices are associative, rather than numeric.

In an associative array, the array index is associated with the value assigned to it. This means that you can write expressions such as **theArray["text"]="this is a line"**. If you wish, you can still use an integer as the index, as in **theArray[50] = "some value"**. It is also possible to use a mixture of strings, integers, and even floating point numbers as indices in the same array, since gawk treats all indices as strings. So the expression **theArray[50] = "some value"** is equivalent to **theArray["50"] = "some value"**.

To make working with arrays as easy as possible, awk provides the programmer with several powerful array operators. For example, to test whether a value is present in an array you can use the **in** operator. For example:

```
if (someValue in theArray) {
   # action to take if somevalue is in theArray
}
else {
   # an alternate action if it is not present
}
```

To perform an action on all values in an array, such as printing each value contained in it, you can use a variation of the for loop, for example:

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

gawk sets the variable **i** to the next value in theArray on each pass through the loop and then prints it.

To remove a value from an array, simply use the delete operator. For example, **delete theArray["word"]** will remove **"word"** from **theArray**.

With associative arrays, you can quickly build powerful applications without concern for the traditional overhead of declaring the array, allocating the memory, or searching for an item in the array. And size is not a factor—the following gawk program easily read and stored all 45,101 words from the file /usr/dict/words into an associative array (in this case, using the number of the current line as the array index):

```
   { words[NR] = $1 }
   END { print NR " words read" }
```

Such a task would be much more involved in C, as you would need to determine how you want to store all the words (An array declared with a size sufficient for all 45101 character strings? A linked list? A binary tree?). You may argue that with C you are free to choose a data structure which will provide much more efficient memory allocation and faster access speed than is possible with an associative array. While this may be true, it does not tell the whole story—it will certainly take you some time to write and test this C program (and very likely, more time to debug it). The power of the associative arrays and the simple, transparent memory management built into gawk means that you are free from dealing with such concerns—just tell gawk what you want and it handles much of the hard work behind the scenes.

Run-time Performance

It seems impossible to have such ease of use together with speed; there must be a trade-off. This is one area in which gawk suffers—run-time performance. However, this is not to say that gawk is a terribly slow language. Since gawk is interpreted rather than compiled, it cannot compete with compiled languages for speed of execution. (It also is somewhat slower than a comparable program written in Perl.) However, if your main concern is getting a working program written as quickly as possible, you probably do not want to wrestle with C or C++ for a week to perfect the most efficient algorithm. By trading off the speed advantages and control features of C (or another compiled language) for ease of use, gawk lets you get the job done quickly and relatively painlessly.

If, however, execution speed is a critical point, gawk makes an excellent tool for implementing and testing a prototype before you start to code in C. And when the prototype is complete you may find that the gawk version is fast enough to meet your needs.

Conclusion

gawk offers the programmer a simple, somewhat C-like syntax, automatic file handling, associative arrays, and powerful pattern matching—features which can help you to create a program much more quickly than with a more traditional language. gawk also has many other useful and powerful features such as user-defined functions, recursion, many built-in functions, regular expressions, multidimensional arrays, formatted output using printf and sprintf, even the ability to set variables on the command line. These features are beyond the scope of this article. Without doubt, gawk's interpreter will produce a slower running final product than a C compiler, or even a Perl interpreter. But this slower execution speed (it certainly is not slow!) is more than compensated for by the speed and ease of program development and testing. When you need a program to perform a task and you need it right now, whether it is a quick-and-dirty, use-once program or a program that will be getting plenty of use, gawk may prove to be the right language for the task.

Ian Gordon (iang@hyprotech.com) is a support programmer at Hyprotech Ltd. in Calgary, Alberta. He discovered the joys of Linux 15 months ago, a discovery which has taken up much of his free time.

Archive Index Issue Table of Contents

Advanced search

Creating A Linux Firewall Using the TIS Firewall Toolkit

Benjamin Ewy

Issue #25, May 1996

If you have a valuable or fragile network to protect, you may want to protect it with a very strong, well-proven firewall. In this article, Benjamin Ewy explains very thoroughly how to build your own 'bastion host' firewall with Linux.

As more and more companies try to develop a presence on the Internet, establishing a secure network perimeter is becoming a very important topic. There are many varieties of what are loosely referred to as firewalls. The general principle behind a firewall is that it serves as a choke point between an internal network and the outside world. The choke point only allows traffic through that is deemed safe.

IP-based filters are one common form of firewall that rely on the source and destination addresses to decide which kind of traffic to pass through. They have the advantage of flexibility in that they can easily be adapted to different types of traffic as needed. The primary disadvantage of IP-based filters is that they rely on IP addresses as the principle form of authentication, and they also lack the ability to look higher into the protocol layer to determine exactly what kind of traffic is being sent.

Application-level gateways are another form of firewall that often consist of a computer called a **bastion host**. The bastion host runs a set of firewall software which implements the policy “that which is not expressly permitted is prohibited”. This policy is implemented at the application level, which allows the bastion host to more completely control the traffic that passes through it.

Implementing the interface between the internal and external networks at the application level allows much more control over the authentication for particular services and, in particular, allows for many forms of **strong** authentication. The main disadvantage of application-level Firewalls is that they require interfaces for every specific application that is to pass through the gateway. If a new application interface is desired, either custom software must be written or the service cannot be provided. The Trusted Information Systems Firewall Toolkit (fwtk) is a very useful kit for creating bastion hosts.

The fwtk supports the functions of a bastion host by providing several small programs that can be pieced together as the site operator desires while simplifying management with a common configuration file. For each service the security policy allows to pass through the firewall, a specific application level **proxy** is required. The fwtk comes with proxies for telnet, rlogin, SMTP mail, ftp, http, X window, and a generic TCP plug-board server that works as a transparent pass-through proxy for many other services.

Additionally, the fwtk comes with a tool called **netacl**, which implements network level access control, and **authsrv**, which implements a network authentication service. This article focuses on preparing a generic Linux host to be a bastion host, obtaining and compiling the fwtk, and configuring its services to support a secure network environment.

Preparing Your Bastion Host

The first step is to prepare the Linux host to be a functional and secure bastion host. There are several principles that firewall builders should adhere to. The ideal bastion host should only provide proxy services and should not be a general purpose machine. Only administrative accounts should be allowed, and if possible, logins to the bastion host should be restricted to the console, although allowing strongly authenticated remote access for remote maintenance will be discussed. The bastion host should not rely on any network services such as NIS or any form of remote file access, such as NFS. Allowing either of these opens up numerous holes that can compromise your bastion host.

Next, it is necessary to verify that the required functionality is available with the Linux host. Every bastion host has at least two network interfaces, one connected to the internal network and the other connected to the external network access point. These interfaces should be configured and tested prior to any further modifications, and you should verify the accessibility of your bastion host from both the internal and external networks. Refer to the Linux NET2 HOWTO and the Linux Multiple Ethernet mini-HOWTO as necessary.

The kernel should be rebuilt, ensuring that IP forwarding (CONFIG_IP_FORWARD) is **disabled** when you do **makei config**. If IP forwarding were enabled, the kernel would automatically forward packets from one interface to the other interface if a route has been established. Controlling this forwarding is what building a bastion host is all about. Finally, if you want to provide a secure mechanism for SMTP mail service, it is necessary to first configure and test sendmail. Refer to the Linux Kernel HOWTO and the Linux Electronic Mail HOWTO, as appropriate.

The next task is to secure the bastion host so that only the proxy services are available. Begin by removing all unneeded services from the inetd configuration file, /etc/inetd.conf. Simply put a **#** in front of each unneeded service line, and when done editing the file, issue a **kill -HUP** to the process id of inetd (perhaps with **killall -HUP inetd**). Remove ftp, telnet, SMTP, nntp, shell, login, talk, stalk, pop, uucp, ftp, bootp, finger, systat, netstat and **every** other service you are not expressly sure you want to provide. We will be defining our proxy services in this file later.

Finally, prevent the startup of any stand-alone daemons by cleaning out the boot files in /etc/rc.d, removing unneeded programs. In particular, check the rc.inet2 file and comment out rpc.portmap, rwhod, rpc.mountd, rpc.nfsd, rpc.ugidd, and ypbind. After you are done removing services, reboot your bastion host and carefully examine the output from a **ps aux** and check that you didn't miss any unnecessary programs. It is also a good idea to run **rpcinfo -p** and the port scanner that comes with the fwtk in the tools directory to verify that all unnecessary services are dead.

Compiling the Firewall Toolkit

Obtain the toolkit, Linux patches, and, if desired, the S/Key package, as detailed in Obtaining Firewall Resources. Bellcore's S/Key provides one-time password support for network authentication if built into the toolkit. A number of commercial one-time password systems are also supported by the fwtk, but their use is not detailed in this article.

Put the fwtk archive in /usr/src/, and run **tar xfz fwtk-v1.3.tar.Z** to unpack it. If you prefer to build in a different place, modify the Makefile.config as appropriate. The Linux patches that we will be applying assume /usr/src/fwtk is the source code directory.

These patches are based on the work of Marco Pauck (pauck@wmd.de) and the firewall-users mailing list. In addition, there are some modifications and additions done by the author to allow the x-gw proxy to work and to support the S/Key authentication mechanism. Most of the patches work around Linux's select() function. Put the fwtkpatches.tgz file into /usr/src/fwtk. Then run **tar xfz fwtkpatches.tgz** which will create a patches directory. Go into the patches directory and run the INSTALL script or apply the patches by hand. If you do not want S/Key support, run the INSTALL.noskey script instead.

Assuming you want S/Key, put the S/Key archive in /usr/src and run **tar xfz skey-2.2.tar.gz** to un-archive it. It is necessary to compile S/Key first so that its libraries can be linked into the authentication service when we compile the fwtk. This S/Key is already ported to Linux, so all that is necessary to build it is to run **make** inside the /usr/src/skey-2.2 directory.

When the patches have been installed and S/Key has been compiled, you can modify the Makefile.config if you want to change any of the defaults, but the rest of this article will assume you have left the Makefile.config as patched. Go to /usr/src/fwtk/ and run **make** and **make install**. The firewall components will be installed in /usr/local/etc/ by default.

Configuring the Network Access Control Lists

The firewall toolkit is made up of three main components: netacl, authsrv, and the various service proxies. Netacl is similar to the tcp wrapper, tcpd, that is common on many Linux systems. Netacl is used to check rules on a per-service basis and take the defined action. You edit your /etc/inetd.conf file to call netacl, passing the normal service to netacl as its first parameter. An example entry for the finger service might look like Listing 1. inetd will invoke netacl when a connection is made on the finger port.

Netacl looks in the common configuration file to see what action to take. The common configuration file is called the netperm-table and is found in /usr/local/etc/netperm-table. A default netperm-table that has many examples, in addition to the ones presented in this article, is installed automatically. Netacl looks in the netperm-table and reads entries that start with **netacl**. Netacl understands the **permit-hosts** and **deny-hosts** options for defining access lists and requires **-exec** to be defined for each line as well, as shown in Listing 2.

A given *service* can have multiple lines of both the permit and deny varieties. *Address* can be a list of hosts addresses, and wildcards such as *.my.domain or 129.17.* are supported. The keyword *unknown* matches hosts that cannot be resolved using DNS. The first rule that matches is the one performed. Lines may not be broken.

In the examples given in this article, the following conventions will be used: .internal.net will represent your internal network. ftp.server.internal.net will represent a ftp server internal to your network. www.server.internal.net will represent a www server internal to your network. compute.server.internal.net is a compute server internal to your network. trusted.external.net is an external network that you trust. bastion.host.internal.network will represent your bastion host's IP address.

An important point is that these rules are not associated with a network interface. If you write rules for a service to allow your internal private network to have special access rights, you must ensure that those IP addresses could only have been received from your internal network. IP spooling can be used to pretend to be a host on your internal network and take advantage of your rules. This can be prevented by using screening routers to block packets claiming to be from the internal network when they arrive on the external network connection. Often your Internet provider can implement this type of rule in the router that feeds your site.

The first line of Listing 3 will exec in.fingerd if the requesting host is a member of trusted.external.network. The second line will match all others and cat a message of your choice. This might be a “fake” finger output to mislead attackers or a simple explanation that the service is unavailable. Netacl is often used for permissions on services that you are not proxying but can also be used to switch among the proxy or the real service based on the origin of the connection. This feature will be discussed in more detail later.

The netperm-table also contains configuration information for all of the proxies, as shown in Listing 4. The format is similar to the netacl format, and each rule can contain multiple lines. These options will be discussed in more detail for each proxy later.

Providing a Network Authentication Service

Authsrv is the authentication server for the firewall toolkit. The authentication server is optional but allows multiple types of authentication to be managed in a consistent manner. Support for the authsrv daemon's authentication by is built into all of the proxies in the firewall toolkit, and can be selectively enabled in the netperm-table on a per-proxy or even per-**permiti-hosts** basis.

Authsrv has support for many different types of authentication, including internal plain-text passwords, and several forms of **strong** authentication using one time passwords compatible with Bellcore's S/Key, Security Dynamic's SecurID, Enigma Logics' Silver Card, and Digital Pathways' SNK004 Secure Net Key. In our example, we have compiled S/Key and its support into authsrv, but the other mechanisms are similar and their details can be found by looking in the /usr/src/fwtk/auth directory. S/Key is a challenge-response one-time password system that will present you with a sequence number and a key at login time. You must give the sequence number, key, and your own private pass-phrase to an S/Key calculator, and it will return a 6 word password. That password will be valid only for that particular sequence number, and it has the property that it was created using a non-reversible algorithm, so it is not possible to easily calculate the next password even if the current one is known. This type of strong authentication is one of the best features of using the fwtk.

To configure the authsrv it must be added to the inetd.conf so that inetd will start it, as shown in Listing 5.

Since authsrv is not a well known service, an unused port must be selected and added to the /etc/services file. The fwtk configuration manual suggests port 7777. The corresponding /etc/services entry would then be:

```
# Example services entry
authsrv 7777/tcp
```

As before, when you change the /etc/inetd.conf file, you must send a **-HUP** signal to inetd to cause it to read in the changes.

Configure authsrv itself by setting up its options in the netperm-table. Authsrv recognizes the **database**, **permit-hosts**, **nobugus**, **userid**, and **badsleep** options. The database option tells authsrv where to find its database, and the **permit-hosts** can be used to restrict which hosts can query the authsrv. It is recommended that authsrv be run on the bastion host so that the database is protected from misuse. An example config might include the entries shown in Listing 6 in the netperm-table.

In our example, the bastion host is running the authsrv daemon, and the bastion host is the only host with proxies requiring authentication by our server. We restrict the authsrv requests to come *only* from the bastion host to prevent unauthorized probing of the database.

Next, we set up an S/Key-based admin account on our bastion host. First, the auth database needs to be initialized. The best way to do this is to run authsrv as root. Then add an admin user and enable logins for that user by entering the following at the authsrv prompt:

```
authsrv# adduser admin
authsrv# enable admin
```

Set the protocol to skey, and give the the user wizard privileges:

```
authsrv# proto admin skey
authsrv# superwiz admin
```

Then you need to set the admin password. S/Key allows your password to be several words long. Enter your phrase between quotes:

```
authsrv# password admin "my neat password phrase"
ID admin s/key is 664 wa56038
authsrv# exit
```

The output returned by authsrv after your pass-phrase is the next sequence number it will use for a challenge and its key. You can use them to generate one-time passwords, as needed, using the **key** program (read its man page, found in /usr/src/skey-2.2/key). For example, if you are challenged with:

```
  S/Key Challenge: s/key 663 wa56038
```

run **key 663 wa56038** and enter in your pass-phrase. It will respond with a six-word phrase that you can enter to authenticate yourself. If you are going to be traveling, there are Macintosh and DOS versions of the S/Key calculator, or you can have key print out a list of your next passwords by running **key -n *`number`* 663 wa56038**, and it will print out your next *number* passwords and their appropriate sequence numbers.

There is another program called authmgr which can be used to remotely administer the authentication database. There are many features in addition to those shown here, such as groups and group permissions for users, and the ability to specify authentication on a per-user, per-time basis. These additional features can be found in the authd man pages. Finally, there are two utilities called authdump and authload which allow you to take snapshots of the current database, for archival or administrative purposes, and then reload the database.

Configuring and Using the Proxy Services

This section focuses on the configuration and use of the service proxies for telnet, ftp, http, and SMTP mail. Common methods for configuring these services are discussed, but each of them has many options and are very flexible. The man pages for each of the services should be reviewed to determine if other configurations might suit your installation better. The proxies for rlogin and X-Windows are configured similarly to the telnet proxy. The Generic TCP plug-through proxy can be used for NNTP news transfers, talk sessions, or any other TCP service a site wishes to pass through the firewall.

The telnet proxy is called tn-gw. The tn-gw has many options, including the **permit-hosts** and **deny-hosts** lines as seen on other services, and a number of message options. The fwtk configuration manual discusses how to use netacl to allow both telnet service to the bastion host and the telnet proxy to coexist on the bastion host, although this is just one of many possible configurations.

First, use netacl to switch the service based on the origin, as shown in Listing 7, which shows an /etc/inetd.conf entry, and Listing 8, which shows the additions to the netperm-table.

When a telnet connection is started, inetd calls netacl. Netacl looks at the source IP address, and if it is not from the bastion host, it calls the tn-gw proxy. The tn-gw proxy prints a denial message and closes, if the source address is unknown, and will allow a non-authenticated connection to compute.server.internal.net only from .trusted.external.net. Connections from the internal net have no restrictions (the default) on their destinations, and users are allowed to change their bastion host passwords if they are coming from the internal network. Additionally, connections from the internal net to the bastion host itself are allowed. Finally, all other hosts are allowed to go to any destination other than the bastion host itself, after they authenticate with the specified authserver. You might not want to use this setup (allowing unauthenticated access from any external site is not a good idea) but it presents many of the options the toolkit offers.

If a user is on the internal network and wants to have telnet access to the external network, they **telnet bastion.host** and then type **c external.host** to connect to the external host.

If a user is on the external network and wants to connect to an internal host they will have to **telnet bastion.host**, enter their userid and authentication as required by specific authentication type, and then **c internal.host**.

Finally, if an administrator wants to connect to the bastion host from the internal network, they **telnet bastion host**, then **c bastion.host**, and netacl will start the real telnet service on the bastion host.

Next, we will configure the ftp proxy system. We will assume your site does not want to provide anonymous ftp service from the bastion host to the external network. The TIS configuration guide discusses in more detail how to configure a site that supports anonymous ftp and the ftp proxy on the same host. Our example will only have the ftp proxy on the bastion host.

The inetd.conf file will need to be modified to call the ftp-gw proxy. Netacl is not used since we are not switching the service being provided, as we were in the telnet example. The inetd.conf line is shown in Listing 9. Then establish the permissions in the netperm-table, as shown in Listing 10.

These lines will print the ftp-deny.txt file and close the connection if the reverse name lookup fails; internal nodes will be allowed to ftp through the gateway without authentication, but **RETR** and **STOR** transactions are logged. Additionally, it will allow external nodes to connect to the internal ftp server after authentication with the authserver, logging **RETR** and **STOR** transactions.

Users on the internal network will ftp to the bastion host and enter their destination at the username prompt. To be forwarded to the site big.archive as user bob, they need to enter bob@big.archive at the bastion host's username prompt, and it will forward the connection. Users on the external network will have to authenticate themselves first and then enter their destination. This is discussed in more detail in the ftp-gw man page.

Now, configure the proxy for http. This is similar to the other proxies and can have a relatively simple configuration. First, add the line in Listing 11 to the /etc/inetd.conf file, and then set up the netperm-table configuration entries shown in Listing 12.

This configuration will allow internal hosts to go out to wherever they want without authentication and allow external hosts to connect to the www.server.internal.net host without authentication. Check the http-gw man page for options involving specific authentication of particular http actions.

For users on the internal network, we can configure their proxy-aware web browser to transparently pass through the firewall http proxy. We will use the Netscape browser as our example. Find the Network Preferences menu under the Options menu. Under that, there is a section on proxy configuration. Select manual proxy configuration and enter the IP address of your bastion host for each of the proxied services and their respective ports. Now you can again use normal http addresses, and the browser will do all necessary requests automatically through the bastion host.

Our final service to be proxied is SMTP mail. The fwtk comes with two programs—smap and smapd—which serve to reduce problems with sendmail and insulate it from some attacks. They do use sendmail, however, so this section will assume that the sendmail configuration for the bastion host has already been setup and debugged. Sendmail configuration is well outside the scope of this article. The Linux Electronic Mail HOWTO can be consulted as necessary.

Smap is a minimal SMTP client that is invoked by inetd, accepts SMTP mail messages, and writes them to a special spool directory. Smapd is a daemon that replaces sendmail in rc.M (or whichever boot script starts sendmail in your distribution). Smapd will look at the spool directory and deliver messages using sendmail periodically. This time, three changes need to be made. Listing 13 shows an /etc/inetd.conf entry to smap, Listing 14 shows how to start smapd from a boot script, and Listing 15 shows the netperm-table entries.

Next, create the /var/spool/inspool directory and make it owned by uucp. Run **mkdir /var/spool/inspool; chown uccp /var/spool/inspool**. Finally, run sendmail from a cron job so that it can process any entries that could not be delivered. A line like:

```
   0,30 * * * * /usr/lib/sendmail -q >/dev/null 2>&1
```

should be added to root's crontab.

Final Points and Additional Resources

The TIS Firewall Toolkit is a very flexible and useful collection of programs for creating bastion hosts. A collection of examples of how to configure a Linux-based bastion host have been presented. Many of these programs have additional features, and the documentation that comes with the toolkit should be read to get the most out of these programs. Several additional tools, such as a portscanner and several log summary generators, come with the fwtk.

One final step before completing your bastion host is the removing of any unnecessary programs that may have been installed. In general, new holes are found every day, so the fewer programs installed, the better. This includes gcc! Without a compiler, many hackers are limited in what they can do if they should break in. It is a good idea to run Tripwire on your system after it is configured, to provide a safeguard against unauthorized modifications to the system. Tripwire verifies the checksums of files and alerts you to modifications. Finally, make a complete backup of your bastion host so that you have a "Day 1" copy to revert to in case of emergency.

There are many useful references for information on firewalls. The fwtk comes with an overview, an installation and configuration guide, a user manual that shows users how to access services through the firewall, and man pages for all of the programs associated with the fwtk.

Useful Linux resources include the **Linux NET-2 HOWTO**, the **Linux Firewall HOWTO**, the **Linux Multiple Ethernet mini-HOWTO**, and the **Linux Kernel HOWTO**. All of these are available on sunsite.unc.edu, tsx-11.mit.edu, and their mirrors.

These and other useful online information about firewalls can be found at TIS Resources.

Several excellent books on firewalls are:

- *Firewalls and Internet Security*. Cheswick & Bellovin, Addison Wesley.
- *Building Internet Firewalls*. Chapman & Zwicky, O'Reilly & Associates.
- *Internet Firewalls and Network Security*. Siyan & Hare, New Riders Publishing.

Benjamin Ewy (bewy@tisl.ukans.edu) has been involved in Unix system administration for 5 years and has used Linux professionally for 3 years. His professional interests include all aspects of network engineering, particularly network security. When not working, he enjoys designing loudspeakers and spending time with his new family.

Letters to the Editor

Various

Issue #25, May 1996

Readers sound off.

Sparc?

Just a short note to congratulate you and the staff on *Linux Journal*. I've subscribed from issue 1 and have enjoyed watching the magazine grow and mature much like Linux itself. Keep up the good work.

On to the suggestion—I was wondering if you plan to run an article on the Linux SPARC port with news and views from the developers. With a number of elderly SPARCs hanging around I'm looking forward to the possibility of running Linux on them.

—Neil Clifford n.clifford@physics.oxford.ac.uk www.physics.ox.ac.uk/sat/vsohp/satintro.html

Yes!

Progress on that front has been amazingly rapid lately, and we expect to run an article on the topic some time in the near future. While development continues, Linux/SPARC currently runs both native binaries and SunOS 4 binaries, including X-based ones. For example, the SunOS Netscape runs under Linux/SPARC. It can netboot with the recent NFS root support in Linux. Debian/SPARC and Red Hat/SPARC are both underway.

Missing Pieces?

I recently purchased the book *Linux Unleashed* and transferred the files on the CD-ROM to a SyQuest 270 MB cartridge. Linux (Slackware version 1.2.1) is now booting and running fine on my computer.

Since 1982, I've done hundreds of Xenix and Unix installations. The only drawbacks I see to Linux are as follows:

1. The termcap file is extremely sparse, not even including the setup for a Wyse-60 terminal!

2. The printcap file is also extremely sparse, not supporting either HPGL or PostScript or Epson dot matrix printers!

3. SCO binaries cannot be run!

My question to you is: Is there a newer version of Linux I could download or purchase that rectifies each of these problems? Unfortunately, I don't have the time to experiment or develop my own version; I need one already prepared that can run SCO binaries and has a comprehensive termcap and printcap.

Thanks for your help. —Ronald W SatzSystems Engineer

Old software

The version you have purchased is very old; Slackware has gone through two major revisions since. Most Linux distributions now include the standard BSD termcap and terminfo files, which do include support for the Wyse-60. Printing support under Linux is much richer than your question suggests. There is a document called the **Printing HOWTO** available as part of the **Linux Documentation Project**. Newer distributions include the LDP documents in electronic form, and several vendors sell the documents in printed form, as well. Finally, support for SCO binaries (through the “iBCS2” package) is now a part of every major Linux distribution. While it doesn't run *every* SCO binary, it does run the great majority. One caution: support for SCO OpenServer 5 has been added in only development versions of the iBCS2 package, and that support won't be added to distributions for a few months.

BogoMips on Pentia

In his “The Quintessential Linux Benchmark” Wim van Dorst writes: “The table also shows that the Pentium processor doesn't have the expected extrapolated multiplication factor. This is due to the fact that the specific busy-loop algorithm is not optimized for the parallelism of the Pentium processor.”

But won't every everybody ask, “What if it were optimized?” and “How would it be optimized?”

I have the answer: the optimization for the Pentium is to put the decrement and branch instructions apart by inserting a nop instruction between them.

This allows the branch to be predicted correctly, and the BogoMips number becomes clock * 1.00 +/- 0.003 (based on two measurements on Pentium 75 and 90).

One of the reasons it is not done in the distribution code is the idea to have one code per architecture (instruction set), and another reason is quite obvious as well—the loop code is only used for timing, and the slower each loop iteration, the less the energy consumption by the processor. You don't want to fry eggs on your chip while measuring BogoMips, do you?

--Leonid A. Broukhis leo@zycad.com

LJ is TOO short

I love *LJ* and finally got a subscription a few months ago because I missed out on a month at the news stand. But, there is just one little thing...

LJ is too short. I get it and two days later I have read it cover to cover twice. Do you think you could make *LJ* a little bit longer so I don't read it up so quickly? **smile**

--Joel M. Lindell linjoe@bethel.edu

Stop The Presses

Michael K. Johnson

Issue #25, May 1996

Apple and Linux, the Linux server, other architectures, Linux Matters and Linux 2.0 announced.

Apple was perhaps one of the last companies that most Linux developers and users expected to join the Linux effort—until Apple's recent joint announcement with the Open Software Foundation that they were supporting a port of Linux to the Mach microkernel running on PowerMac computers. This announcement was treated by Apple as if it were of equal importance to their announcement of a large price cut on most of their computers. The announcement was made on the fifth of February at the Freely Redistributable Software Conference, and made international news that day and the next.

Linux had already been ported to some PowerPC machines before Apple's work was started; see the Linux/PowerPC web page at http://www.linuxppc.org/linuxppc/ for the latest information. They have successfully run Linux/PPC on a Motorola PowerStack computer. The work that the OSF did is significantly different; instead of porting Linux directly to PowerMacs, they ported Linux to **OSF MK**, the Open Software Foundation Microkernel, based on Mach 3. In microkernel parlance, this is creating a Linux "server". While they did that, they also ported OSF MK to the PowerMac. After completing those projects, porting their Linux server to OSF MK on the PowerMac took less than three weeks. That doesn't mean that it is production quality yet, but that is significantly faster progress than most ports of Linux.

The Linux server creates other potential advantages, besides making Linux quickly available on Apple PowerMacs. For instance, OSF MK has been ported to platforms that Linux has not yet been ported to, such as Intel i860 and HP PA-RISC. Developers wishing to port to such platforms may find it easier to port the Linux server to OSF MK on those platforms than to do a complete port of Linux to those platforms, or may find it easier to start their port by first porting the Linux server, and then working on porting the standard monolithic Linux kernel.

Just as at DEC, Linux support at Apple is a serious part of company strategy supported by corporate management. By maintaining close contact and a friendly relationship with the developers working on the other Linux/PPC ports (Motorola, Be, FirePower, and IBM), Apple has assured the Linux community that it is not attempting to take over Linux development, but instead has become a contributing member of the community.

The work has not only contributed to Linux, but also to OSF MK. Several performance improvements and extensions were made to OSF MK to improve the performance of the Linux server, which may also improve the performance of other servers on OSF MK.

But That's Not Linux!

Linux is designed as a monolithic operating system, the antithesis of a microkernel-based operating system. Many people have remembered that Linus has said that he is not interested in making Linux be a microkernel operating system. Doesn't creating a Linux server violate the whole idea of Linux?

Not in the least. First of all, the great majority of the source files were unmodified. Of 909 original files in the Linux source code, only 43 had to be modified. Second, the source code to everything, including the drivers in the microkernel, is freely available, and can be used to develop a monolithic Linux/PowerMac if there are sufficiently interested developers. Finally, no one is suggesting that the Linux server should replace the monolithic version of Linux as the official version.

For more information, see the web site: www.osf.org/mall/os/mklinux.html

Other Architectures

Over several months, support for Linux/Sparc has progressed on the sun4c architecture to the point where it is now able to run all normal SunOS X-based binaries correctly, including Netscape, and work on the native user environment is proceeding at a rapid pace.

On February 17th, David Miller (the leader of the Linux/Sparc team) announced that he had achieved a shell prompt on the sun4m architecture; specifically a Sun SparcClassic. (if you don't know why that is significant, read the series of articles on porting Linux to the DEC Alpha in *Linux Journal* issues 18-21.) On the

20th, he “apologized” for that “lie,” announcing that it now “runs all the programs that work on the sun4c”.

Several other developers, including Miguel de Icaza, are hard at work developing a full user environment and distribution of Linux/Sparc. Keep an eye on *Linux Journal* and the comp.os.linux.announce newsgroup if this whets your appetite.

Linux in the News

Linux Journal is no longer the only magazine which covers of Linux. In February, *Byte* ran a cover story entitled **Linux Matters**, which explained that Linux isn't just inexpensive, it's also worth using. *Infoworld* has regularly given Linux reasonable coverage; In the February 19th issue, Nicholas Petreley's reaction “mimicked the attitude of my 2-year-old daughter at a toy store: `gimme have it.”' Linux has been regularly mentioned in *Dr. Dobb's* and *Unix Review* for over a year. Of course, we still think that *Linux Journal* is the best place to get your Linux information fix.

Recent Development

As we announced last month, Linus is preparing to release a new stable version of Linux, provisionally called Linux 2.0. More features that will be interesting to our readers include true NFS caching, made possible by the new “page cache” which debuted in the 1.3.51 kernel, SMP on Pentium Pros, VFAT filesystem (Windows 95 “long filename”) support, APM (Advanced Power Management, normally used on laptops) capabilities, and disk quotas.

Freely Redistributable Software is Alive and Well

Phil Hughes

Issue #25, May 1996

The first conference of its kind, this historic meeting was attended by a small but knowledgeable group of freeware enthusiasts.

I have just returned from spending four days at the Freely Redistributable Software Conference. It was held in Boston on February 2 thru 5 and sponsored by the Free Software Foundation. The first conference of its kind, it was a real treat to attend.

With less than 200 attendees, it wasn't a large conference but I feel it was very significant. For those of you who have attended Usenix in the early to mid-80s, this conference has much the same flavor. For those of you who didn't, *Nerdcon* might give you an idea of what I am talking about.

It is not that the attendees were all Nerds but they weren't marketing people or novices to computers.

Discussions tended to be serious and technical and the participants had the knowledge to back up their positions.

How it Started

The conference began on Friday night with a reception. It was a good chance for attendees to meet each other. While I didn't see any important decisions being made, it was my first chance to get a feel for how significant the Linux influence would be.

Greg Wettstein of the Roger Maris Cancer Center came over to talk to me and said he hoped to meet Linus. I pointed out Linus who, along with his girlfriend, Tove, were at the next table. Greg went over, introduced himself and hung out with Linus for quite a while. Greg commented to me later on how impressed he was with Linus, for his work on the system of course, but also as a person. Greg was not the first or only person to make a very positive personal comment about Linus. Many, including myself, think that his personality, that is, his willingness to listen to other opinions—good or bad—is one of the major reasons Linux has become so popular. I think that his sense of humor is another important contribution.

A Full Saturday

Saturday offered a collection of tutorials including the two half-day sessions I presented on Linux—Linux:An Open System for Everyone and Installing and Running Linux. They were well attended (about 30 students) and there were definitely some new converts to Linux. While the tutorials were intended for Linux newcomers, there were a few serious Linux users in attendance. The general conclusion was that we had a good time and everyone learned something.

The auxiliary meeting location was the Cambridge Brewing Company (CBC), a brewpub with good food and good brew about two blocks away from the conference hotel. Linus and a large group of Linux-followers ended up there on Saturday night; ten of us including Dan Quinlan, Jon “Maddog” Hall, Erik Troan, Donnie Barnes, Eric Raymond ended up there for lunch on Sunday and another group including Greg Wettstein, Steve Imlach, Tom Sargent and Bryttan Bradley and myself ended up there on Sunday night.

While there were other important ideas discussed at CBC such as convincing *60 Minutes* that they should do a segment on Linus and Linux, the quality of their stout was an important consideration. Personally I would call it a thumbs up.

The Sunday Conference

This was the only day of open sessions. The first keynote (by Linus) and five of the ten sessions were on an aspect or use of Linux. The sidebar outlines all the sessions and tells you how you can get a copy of the proceedings.

Sunday started off with a keynote by Linus (after I had fun waking up the crowd and attempting to make Linus sound like an important mainstream executive).

Linus originally planned his keynote around being the Dr. Ruth Westheimer of Software. With a fake German accent he intended to ask questions like “Do you go blind if you program alone?” “Is it ok to date different operating systems?” “How do I know when I met the right OS?”

He then intended to continue by answering the questions. But, that didn't work out so he went on to his second idea, “ Software is almost but not quite totally like sex”. He decided it just didn't have the same zing to it so he went on to present a talk titled “Write Free Software, Travel the World and Meet People”.

He said “Rather than writing a great operating system I wrote a small, not so great operating and then made it free. That turned it into a great operating system.” Certainly the right attitude to take at the conference and, in practice, it seems like he is right.

I personally found great interest in Victor Yodaiken's talk about using Linux at New Mexico Institute of Mining and Technology. Briefly, what he has done is slipped a real-time kernel between Linux and the hardware. This had made it possible to handle hard real-time processing (this is where processes must run at a required time, not just “soon” and yet still have the utility and versatility of a general purpose operating system running on the same hardware. Victor has promised *Linux Journal* an article on his work.

There is an interesting comment in the conference paper on the Yugoslav Experience that helps support the idea that Linux is viable in an open market: “even though the price of pirated commercial Unix-like operating systems is comparable to the price of Linux, and with practically no legal limitations to using pirated copies, Linux is being more widely used for Internetting in both [the] academic community and companies.”

The next presentation was on Linux on the OSF Mach3 Microkernel. This fits in with Apple's announcement of Linux on the PowerMac. Let's just say this was a surprise. Michael addresses this subject in *Stop the Presses*.

Greg Wettstein presented a talk on his work with Linux at the Roger Maris Cancer Center. An article on his work appeared in *Linux Journal* issue 5. To condense a serious effort into a sentence, the cancer center is using over 30 Linux workstations running a custom-built patient care information system using Perl and Tcl/Tk.

In the following session, Erik Troan presented a talk on the package management system developed at Red Hat software. This package, RPM, makes in possible to easily update an existing Linux system. The bottom line is that Red Hat is encouraging vendors of Linux software to use their packaging method. This will make it easier to distribute and maintain software for Linux.

Monday I decided to play the role of student instead of presenter and attended Tom Christiansen's all-day Perl tutorial. At lunch we had a lively discussion of Linux, NetBSD, software licenses, is Emacs the answer to the world's problems and all those other important subjects.

During the afternoon I mentioned to Tom that I was glad I had proofed Arnold Robbin's (excellent) Awk book before attending the Perl session because of the confusion that ensues when I am trying to think in Awk and Perl at the same time. This led Arnold and Tom into a “which is better” discussion at afternoon break. No bloodshed and I think they are both right but it was interesting to listen to them. As I have been an Awk user for about ten years and am just starting to work with Perl, I intend to write an article or two for *LJ* on how I see the two languages fitting into the world.

Words from Other Attendees

While I found the conference extremely interesting and valuable I wanted to get feedback from others. Here is what a few attendees had to say.

> **Russell NelsonCrynwr Software, nelson@crynwr.com**Cool stuff: Meeting people f2f (including Linux && girlfriend [Tove], Peter Deutsch, Tom Christiansen, Greg Wettstein and that fyl fellow. Badges with e-mail addresses on them, so you'd know who people are.
>
> Uncool stuff: The price relative to the length. ... Badges printed with a too-small font, so I had to squint at people's chests to know who they were. ... I like to call it freed software. It's a little ungainly, but it carries more information than the confusing appellation “free”.
>
> **L. Peter DeutschAlladin Enterprises, ghost@aladdin.com**Cool stuff: the two keynote talks. I found Stallman's quite inspiring in terms of getting a more spiritual/emotional understanding of the “free” software concept. Meeting people face to face. Having the opportunity to present my own twist on “free” software to a lot of people who didn't entirely agree with it. The gamelan concert. The free Linux CD-ROMs.
>
> Uncool stuff: I second Russell [Nelson]'s comment on price versus length. If I hadn't been presenting a paper, I probably wouldn't have come. The small attendance, presumably arising from the short notice.
>
> (Russ and Peter's comments continued from each other. Russ commented that he thinks the FPL (which is what his paper was on) encourages free software even more than the GPL. He points out that products live and die by market share.)
>
> **Steve ImlachSynergy, si@synergy.encinitas.ca.us**My eyes have been opened to some great opportunities in our area using Linux on our hardware and some contributions that we may provide to the continuing Linux development.

> First, I thought it was a good sign to see in the *Boston Globe* Business section about Apple and the Linux server on top of the Mach kernel. This is some good stuff. Synergy is already at work with the Mach 3 kernel and would be very interested in getting more information in this area.
>
> Since we are manufacturers of high performance Single Board Computers, we are interested in first, porting Linux and second, helping with the real-time development. Synergy is also highly interested because it would not only help us in performance but also improve the bottom line to our customers.
>
> The ideas keep coming. Linux can also provide a high performance testing environment for our boards because we would have the source to tweak in any areas we deem to be a snappy enhancement.
>
> It's just a matter of getting started. I see a lot of full weekends ahead. Luckily, unlike our FSF friends, I have a job that supports this development in an ugly capitalistic evil manner. You know, it's the profit mongers like Synergy that donate to the FSF year after year because we haven't learned to steal what is free yet.
>
> Ok, that was a little jab at RMS. His speech was a little tainted, I thought. His story about the “community” in college reminded me of how I felt when I played ball at school. I was sad it was ending and about the *camaraderie* I felt should never end. The thing is, you move on. That is not to say, however, that the same type of teamwork couldn't happen at the workplace. He shouldn't knock it. Has he tried it? I think he is exactly right when he says he doesn't like the competition. There is plenty of it out here in the real world. What could he do if he was pushed via that competition. Too bad. FSF stuff is used everywhere and has great admirers here at Synergy as reflected by the annual donations. I believe he should balance his remarks to reflect what is really allowing the FSF to survive. Cooperation with the business community. He seems to affect a lot of views in the FSF “community”.
>
> Linus was the type of guy I would like to work with. He certainly shows it in the product and the way he handles himself while juggling everyone's ideas, good and bad. It's no wonder that he is such a popular fellow. I hope I may be factor in the coming days along with other engineers at Synergy that are more experienced than I in the advancement of real-time Linux. First, there will be the Linux learning curve as we delve through the source ...
>
> Dr. Greg was inspirational and a great new-found friend. We talked about just about everything and he is just the type a person that you would want to help up

at Roger Maris cancer center in N.D. What was really amazing was watching him and Tom Sargent (Synergy-Tucson) talk forever about everything from Cancer research to spelunking to circus maneuvers in vintage aircraft to man-eating viruses found in the depths of Africa. These guys need to have a talk show on cable.

P.S. Linus Torvalds deserves great credit for the revolution of free OS. What is his status now and how does he get support for the continuing effort? I am also forwarding this to Dr. Greg. I would like to see how his efforts are paying off up there in what would otherwise be a drain on the spirit of a mere mortal.

Dr. G.W. WettsteinOncology Research Div. Computing FacilityRoger Maris Cancer Center, greg@wind.rmcc.com I enjoyed the opportunity we had to visit at the conference (and the beer). I think the Conference on Freely Redistributable Software was important for its implications about the entry of free software into commercial venues. This was probably demonstrated most appropriately by a sampling of the selected papers. Of particular interest was the paper from the Apple/OSF consortium as well as the papers from the group working through SBIR and Peter Deutsch's thoughts on marketing. Hopefully my discussion of our accomplishments at the RMCC were helpful in this area as well.

The impact of redistributable sources on the commercial arena was also underscored by the presence of commercial enterprises at the conference. Of notable interest was the presence of Apple and representatives from HP and Synergy Systems. Apple's presence was extremely important with their announcement that Linux would assume a role in their strategic presence and would be marketed as an Apple offering. Who would have ever thought that an organization such as Apple, which considers software as strategically central to their business plan, would consider free software as a viable marketing tool.

What I found extremely interesting were discussions with Steve Imlach and Tom Sargent from Synergy Systems. Their interest in Linux is specifically for the strategic advantage that it will confer on their hardware marketing efforts. Their group feels that Linux represents a high quality solution which will allow them to more effectively penetrate markets due to the cost advantage it imparts to their hardware platform.

The implications of this are extremely important for the freely distributable software movement. Steve Imlach indicated that the leadership of Synergy is committed to the philosophy of free software. This commitment includes monetary donations to the effort as well as a commitment to release sources back into the Linux kernel development stream. In my mind

> this is representative of the wonderful synergy that can and should develop between commercial enterprise and the free software movement.
>
> In business parlance this is a win-win situation for everyone involved. Commercial enterprises gain a strategic advantage and the free software movement reaps monetary advantage. An even more intrinsic advantage and one that must be carefully considered is that the software remains free. Capitalism flourishes and ultimately incubates the growth and development of freely distributable software.
>
> This conference came at a particularly important time for the free software industry. If the concept is to move forward, it is imperative that advocates join hands with commercial endeavors wishing to exploit the benefits of free software. Niche markets must be targeted and developed so as to promote commercialization of free software with the ultimate benefit of subsequent re-investment.
>
> Phil Hughes, publisher of *Linux Journal* probably phrased the imperative most correctly when he said, “It is time for Linux and free software to have applications and markets which allow it to DO something.”

Conclusion

Finally, here I am on a late night flight on my way back to Seattle so I guess I am done with what I need to tell you about the conference. We intend to keep up on plans for the next Freely Redistributable Software Conference and will let you know about it in plenty of time so you can attend. If you want to rub elbows with some of the important people in the free software movement, get genuinely inspired by the high interest level in Linux, learn something new and maybe drink a real or virtual beer or two, it is the right conference to attend.

—PH

Conference Proceedings

The following papers were presented at the Sunday conference. Note that a printed copy of the proceedings is available for $25 (postpaid in the U.S., add $10 for foreign airmail) from Free Software Foundation, 59 Temple Place, Suite 330, Boston, MA 02111. They can be reached by phone at +1 617 542-5942 or fax at +1 617 542-2652.

Automated Management of a Heterogeneous Distributed Production Environment by Ph. Defert; CERN, European Laboratory for Particle Physics, Geneva, Switzerland

Freely Redistributable Software across the Internet—Current Practice and Future Directions to Overcome the Bandwidth Crisis by Neil Smith; HENSA Unix, University of Kent at Canterbury, UK

Cheap Operating Systems Research and Teaching with Linux by Victor Yodaiken, New Mexico Tech

Freely Redistributable Instead of Commercial Software—Yugoslav Experience by Radivoje Zonjic; Department of Electrical Engineering, Belgrade University, Yugoslavia

Linux on the OSF Mach3 Microkernel by Francois Barbou des Places; OSF Research Institute, Grenoble and Cambridge

Internationalization in the GNU project by Ulrich Drepper; University of Karlsruhe

Perceptions—An Implementation of a Medical Information Support Environment with Freely Distributable software by Dr. Greg W. Wettstein; Oncology Research Division Computing Facility, Roger Maris Cancer Center

The RPM Packaging System by Erik Troan; Red Hat Software

Coordination Joint Cost/No-Cost Rights for Software Developed with SBIR Funding by Philip A. Wilsey; Computer Architecture Design Laboratory, Department of ECECS, Cincinnati

Licensing Alternatives for Freely Redistributable Software by L. Peter Deutsch; Aladdin Enterprises

Linux in the Real World

Todd Lewis

Issue #25, May 1996

If you have ever considered starting your own Internet Services Provider business with Linux, Todd gives you both an introduction to the subject and a quiz to help you decide if you really want to start.

Ever since the opening of the Internet to commercial development a few years ago, many an individual has made a living for himself by leasing a connection to the global network, establishing a local network, and tacking up his siihingle as an Internet Service Provider (ISP). ISPs, from the mighty MCI down to the most humble Mom and Pop basement organizations, offer exactly the same thing—a direct connection to the Internet. Smaller businesses do this mostly via PPP connections over modems, whereas larger ISPs tend to stick to leased-line dedicated connections.

Although this might sound daunting at first, the actual process is not impossible, even for a single individual of limited resources. The most pressing obstacle in the past has been that the resources needed to offer ISP-like services—not merely PPP connections but also e-mail, Usenet, and FTP/WWW hosting—have only been available in the form of high-end (and high-cost) Unix workstations. This translated into considerable cost for the budding entrepreneur.

The other traditional problem has been that the system- and network-administration skills needed to offer such services could only be gained on one of these high-priced workstations, creating a chicken-or-the-egg dilemma, whereby if you did not already have access to these machines through work or school, your main capital investment would have to sit idle for six months to a year while you tried to learn how to run it.

The ubiquitous nature of Intel-x86-based personal computers, the strength of their networking capabilities and, most of all, free operating systems such as Linux brought this sad state of affairs to an end. With its traditions of laissezfaire development and absolutely free access, Linux offers two qualities essential to the upstart ISP. The first of these is minimal cost. The second, and the more important, is the ability to learn *all* the essentials of network administration on an operating system which refuses to withhold any secrets from you.

It is hard to reinforce this latter aspect of Linux enough. As an independent Internet Service Provider, your job is network management. If your machines are not routing traffic properly, *you* must understand why and be able to fix it. If your Usenet feed is clogging your system, *you* must be able to diagnose the problem and fix it. If your DNS is not spitting out IP numbers properly, it is *your* problem, and yours alone. Microsoft technical support will not come down and help you figure out why. Sprint or MCI will not drop what they are doing to help you out, even if you are their customer. Good consultants (other than the author) are notoriously hard to find and (including the author) are horribly expensive.

Linux is *the* operating system of choice for ISPs, because it fulfills the number one requirement for an ISP's OS. You must be able to diagnose problems, and you must know enough about your system's operation to be able to fix the problems which *will* plague you, no matter what OS you choose. This, combined with the huge range of software available for Linux, make it the hands-down winner.

This is not to say that Linux is perfect, and indeed, there are several problems with Linux in an ISP environment. This article should help by offering a roadmap, showing both obstacles and bypasses to help you in your journey. It does not offer advice on how to connect to the Internet, but rather concentrates on using Linux to offer ISP services.

Getting Started: Hard(ware) Choices

Your first step is deciding on a hardware base. What kind of machines will you run? How many of them will you have? What peripheral equipment will you need? You should have good answers to *all* of these questions before you buy any hardware.

In the past, the platform decision was easy. If you ran Linux, you used an Intel-based or Intel-clone 386 or greater machine. With the recent enhancements to Linux for the Sparc, MIPS, and DEC Alpha, this choice is a little more clouded. Red Hat and others are coming out with full-fledged CD-ROM distributions for these higher-powered machines. While these are in the early stages of development and I do not recommend them (yet), the price-to-performance edge of these machines, especially the DEC Alpha, means you should keep them in mind as a real option, especially for upgrading your system down the road. For startups, though, x86 machines are usually the way to go.

If you are going to concentrate on dial-up business, you will need modems. Modems can be one of your worst nightmares, and you should shop carefully. However, the modem issue is not Linux-specific. With V.34 finally having settled down to a well-established standard, most modems will do the job, and many people suggest shopping for price. Nonetheless, doing your homework by reading reviews and soliciting feedback on Usenet can pay off many times over. Just think about dedicating hundreds of man-hours and thousands of dollars to modems that refuse to work, and ask yourself if the extra initial effort might be worth it.

Of course, standard PCs can handle a pair of modems at best. How do you connect all of these modems to your PC? The answer is through one of the many port-servers (also known as terminal servers) available on the open market. Port servers connect to your modems with built-in serial ports and forward data back and forth between them and your computer, usually over a network. Alternatively, you can use an “intelligent serial board” which connects the modems directly to a PC. A good review of intelligent serial boards can be found in the June, 1995 issue of *Linux Journal* on page 46. Remember to include a port server or intelligent serial board in your plans.

Then there is your network itself. Unless you plan to fulfill all of your requirements with one machine (usually not a good idea), you will need a network to connect your machines. While Fast Ethernet, FDDI, and ATM are all options, plain vanilla 10 Mbps Ethernet is usually the way to go. I recommend 10-base-T (twisted-pair 4-conductor cable with a hub) over 10-base-2 (coaxial wire with T-connectors and terminators). 10b2 is simply not reliable enough to avoid service disruptions as you remove machines from the network, rearrange your network with expansion, and trip over the network cable. If you decide on the 10bT route, remember to equip all of your machines with Ethernet cards (anywhere from $30 to $300 each) and buy an Ethernet hub (between $400 and $1500). The NET-2-HOWTO includes an excellent discussion of the merits of the various ethernet cards; ISA bus ne2000 clones offer an easy and very inexpensive solution.

Presumably, you have arranged an Internet connection, either through one of the nationwide services such as Sprint, MCI, and ANS, or through another regional ISP. Traditionally, this connection is done via a dedicated router. Cisco Systems makes the best high-speed routers available, but they are usually overkill for connections of T1 speed or less. Livingston routers are a particular favorite among the Linux community, and other options also exist. An increasingly popular option is to make your own router out of a Linux-based

PC, using a T1-interface card from a manufacturer like Enhanced Technologies. If you have an ISDN connection, then you can use your Linux box as a router too, through an ISDN terminal adapter from 3COM, Boca, Motorola, and others.

Finally, you have to decide how many Linux machines you will need. This all depends on how you have designed your network. There are two big mistakes that people make. On the one hand, some assume that every little network function needs a dedicated server. “We have to have a dedicated DNS, a dedicated web server, a dedicated FTP server, a dedicated mail server...” This can become very expensive, very quickly, and unnecessarily so. On the other hand, I have seen people make the mistake of cramming news, mail, PPP, and web service all on one machine, which is so slow that their customers leave as fast as they come in the door.

What hardware will younee and how should it be connected? How much will it cost? Through the rest of this article, we'll look at the service you can offer as an ISP and what kind of setup you'll need to do provide it.

Let's Linux!

The Linux kernel only offers the basics of an operating system. You must chose one of the many Linux distributions to use on your system. ISPs have a number of specific concerns which might not coincide with the general user's, and foremost among these are **upgradeability**, **coherence**, and **network integrity**.

Upgradeability is essential to the ISP because of two competing demands. First, you will need to offer the latest and greatest solutions. The Internet protocol suite of services is constantly growing, and the reason we have distributions in the first place is so that we all don't have to port, clean up, and install the various software packages on our systems by ourselves. Secondly, your service must be reliable. This means that when you do want to upgrade your system, taking it down, formatting the hard drive, and installing a new distribution is not a very good option.

By coherence, I mean how well the various components of your system fit together. For example, installing Wietse Venema's TCP wrapper (/sbin/tcpd) is not a trivial exercise, mostly because you must coordinate it with the various weird features of your network daemons. Slackware and most other Linux distributions come with tcpd built in and ready to go. A coherent Linux distribution saves you a lot of effort, as you need not fix the mistakes in the distribution.

Finally, you will live or die by the quality of your network functions. Sloppy and/or buggy compilations of network utilities, non-functioning daemons, or generally inferior network services are unacceptable in your distribution.

New distributions are usually plagued by the latter two of these problems, and this was the case with the early releases of Red Hat. However, Red Hat is very easily upgradeable and has been steadily improving the general quality of their product. Additionally, if you are brave enough to try Linux on a Sparc or Alpha, then Red Hat is probably the way to go.

Coherence and strength of network services have always made Slackware a particular favorite of mine, and I have gotten a lot of mileage out of it. It saddens me, therefore, that the persistence of bugs in the distribution and, most of all, the impossibility of upgrading a Slackware system are causing it to drop out of favor.

My own bets are laid on Debian, a package maintained much as the kernel is developed—by a team of people over the Internet. Debian is still gearing up, and there have been problems with one CD-ROM version of it, but I think that it holds the most promise.

Speaking of CD-ROM problems, get ready for them. Many times I have run into people who are having networking problems with a CD-ROM version of Slackware, to whom I recommend reinstalling the “N” section (networking) from an Internet source. More than 90% of the time, this solves the problem. We have a T1 connection to the Internet, and although we keep a CD-ROM version of Linux lying around for safety reasons, I prefer installing directly from the network, because if one vendor's CD-ROM version of Slackware is broken, only their customers will know it, but if SunSITE's copy of Slackware is broken, the whole world starts yelling.

Really, choosing a distribution depends more on what you're familiar with than anything else. If you have experience with Red Hat, for example, you will want to think seriously before selecting a different distribution, as the pain of learning a new distribution might be greater than the advantage to be gained.

Networking—It's Not Who You Know, It's What You Know

Your major task after installing your new distribution is to set up your network and get packets flowing. This is not the topic of another article—it is the topic of a full-length book! Fortunately, the Linux community is very responsive to the needs of its customers (i.e., itself), and one of the main reasons to use Linux is because of the documentation. You should read the Net-2-HOWTO as an absolute minimum, and a copy of Olaf Kirch's *Linux Network Administrator's Guide* (the “NAG”) should sit on your desk within easy reach.

Of course, getting packets flowing is just your first concern in establishing your network. There are 6 main topics which you should consider: connecting customers, DNS, mail, news, Web/FTP, and network security. We will examine each of these in turn.

PPP or SLIP? Coke or Shasta?

The Serial-Line Internet Protocol (SLIP) was a ground-breaking development in computing. Then again, so was the Apple II. While still in favor in certain circles, SLIP is dead and should be recognized as such. Its successor, the Point-to-Point Protocol, (PPP) is the wave of the future. (Okay, it was the wave of a year ago, but you get the point.)

SLIP is firmly entrenched in second place to PPP for several reasons. First of all, SLIP is only capable of serving IP traffic, whereas you can run virtually anything over a PPP link, including IP, AppleTalk, IPX, and others. Second, PPP's Link Quality Management (LQM) functions give you a solid connection by running it at the proper speed, regardless of electrical interference or other line noise. Finally, the newer versions of PPP will have world-class authentication, enhanced LQM, and other features, such that PPP will continue to pull away from SLIP in terms of quality.

Linux comes with both SLIP and PPP built in. Your kernel needs to be compiled with *both* SLIP and PPP support. Remember this when you install your system, or you will rack your brains trying to figure out why one or the other is not working when you try to use the missing one.

You will want to read the PPP-HOWTO, as it is invaluable in understanding how to make PPP work. You will need to decide a number of things, such as whether to give customers shells from which to invoke PPP or make PPP their default login program.

Furthermore, how will you do the accounting for their time on-line? The answer to the previous question might affect this one. Adam McKee's BBS-Util might be helpful, as might a number of other packages. Ask around, look around, or do what we did—write your own.

Getting PPP/SLIP to work is a medium-sized job, and you will want to test your setup via modem with as many different OSs as possible. Win 3.11, Win95, OS/2 and OS/2 warp systems need to be able to dial in, and you need to test all of these systems to make sure that you can explain to your customers how to do it. Writing a HOWTO for each OS as you test it is a good idea. And don't forget your Macintosh customers! They suffer under the oppressive boot of hegemony as much as MS Windows users do, and they usually make good customers, as they are willing to pay good money for a product that works.

A good decision might be to put your PPP customers on the same machine as your mail server, making it easier to maintain only one /etc/passwd file for the entire operation. pppd will use username/password pairs from the /etc/passwd file, so throw that pap-secrets file away!

PPP is fairly time-consuming to set up, and if you decide to offer non-IP services, this, too, will take time. However, usually once it's up, it's up, and you can go on to other problems. Just be sure you have backups!

DNS: What's in a Name?

While computers understand 32-bit IP addresses, you and I don't. My e-mail is tlewis@cheney.net, not tlewis@204.214.16.150 (although that works, too). The Domain Name System is the glue that holds together domain names and IP numbers.

You do have an Internet domain name, don't you? You will need one, and once you have it, you will need a Domain Name Server (DNS) in order to use it. DNS setup is fairly straightforward, and I have written a mini-HOWTO on the subject, which will be available from the LDP archives by the time you read this. [FIXME: More information coming here]

DNS places a very light load on a machine, and your DNS server can serve in other capacities, too. Some people put it on their PPP server, some on their news server. We have two: one on our WWW server and the other on a 386-40 dedicated to DNS. To each his own.

E-mail: It Makes the World Go 'Round

E-mail is probably the most useful and the least appreciated aspect of the Internet. Once you have tired of the pretty pictures of the WWW and want to get some work done, e-mail is the tool of choice. Nonetheless, people neglect it too often, at their peril.

If your news server goes down, you will get some gripes, and if your web server goes down, you will get some calls, but if mail goes down, Annie bar the door! Your POP server (the program that lets customers use programs like Eudora to read their mail) usually works out of the box, but your SMTP daemon (such as sendmail) is what does the heavy lifting, and this is where your efforts will be directed.

The NAG includes a good section on mail, and the Mail-HOWTO is also a good starting point. Mail is another of those services with which, once you have it working, your problems are over, but you will spend time getting it working properly. If customers want UUCP, mailbots, or mailing-list support, this means additional work, and you should consider the amount of additional work before knee-jerking into a “Sure, we can do it!” response.

News: In a Word… ARRGGHH!

Usenet, the global electronic news system, is distributed mainly over the Internet, although certain UUCP networks and others also offer it. Simply put, any message written by anyone anywhere submitted to public Usenet groups (1000, 4000, 10000—how many are there?) will end up on every Usenet server on the planet. Last I heard, a full Usenet feed (i.e., accepting every group) runs about 400MB per day, so you probably don't want a full Usenet feed coming in over your 56kbps connection.

The News-HOWTO goes over the various options for news servers (programs which accept deliveries, organize the articles, and spit them back out to customers as they request them). Pick one, learn it, and stick with it. At 400MB per day, news is an inherently dangerous thing, kind of like having a water main feeding into your sink, and you can lose control of your news server without a whole lot of effort, filling disks and ruining your day.

News also exacts a heavy toll on the machine that serves it, consuming large amounts of RAM, CPU cycles, and disk space. At the very minimum, you will want your /var/spool/news directory on a separate physical disk. Usually, it is a good idea to have a separate news server, which perhaps also serves as a DNS. Our 486-DX66 groans slightly under a full news feed, and a Pentium might not be a bad choice. Then again, you might not want to run a full news feed, in which case a 486DX-33 might do the trick.

Web and FTP: The Fun Stuff

Slackware and most other distributions come with Washington University's wu-ftpd anonymous FTP server built-in, preconfigured, and ready to go. Go to /home/ftp/pub, start throwing stuff in there, and you're off to the races. If your customers want to be able to put up files for FTP on your server, then you might have some work to do.

The WWW, despite all the hype, is not especially difficult to implement, either. Apache, a derivative of NCSA's httpd, is a favorite Linux tool and comes with plenty of neat gizmos. Virtually all web servers are capable of serving information from a public_html directory in a user's home directory, accessible as www.foobar.com/~username/. If you decide to do this, you will want to put your web daemon on the same machine as your users' home directories, which may also be your mail machine. If this, along with PPP, is too much, break out your PPP server and leave mail and web on the one machine. Plus, you can have users access this one central machine via multiple PPP servers (even via multiple remote PPP servers across your greater metro area), instead of having each server duplicate these functions.

Network Security: I Could Tell You, But Then I'd Have to Kill You

ISPs offer access to their networks to people they usually have never met. As such, you will need to keep a constant eye on the security aspects of your system. Start with a good introduction to network security, such as Cheswick and Bellovin's *Firewalls and Internet Security*, which served as my introduction to networking in general.

A good place to start is with a packet filter on the router connecting you to the Internet. A good packet filter and careful password management are two small steps that will put to rest 90% of your security concerns. The most important step you can take with Internet security is to understand it and to use the tools (like TCP wrapper and packet filters) at your disposal.

Conclusion

The above descriptions cover the high spots of all the issues which you will face in starting up an ISP business using Linux. It *is* doable; we at Cheney Communications and countless other ISPs are living proof of this.

You will have problems with your network. You should get used to that fact now. Fortunately, the good people on Usenet in comp.os.linux.networking (myself included) are always ready to help. The Linux Documentation Project is an invaluable resource when trouble arises. It is even more useful **before** trouble arises!

By now you should have a pretty good idea of what equipment you will need in order to start dishing out IP services. Are you going to offer limited Usenet and e-mail to a few businesses? An ISDN connection and a pair of 486DX-66s should do the trick. Are you starting up a full-service ISP for dial-up and leased-line services, with a full news feed and commercial web hosting? Three Pentiums (dedicated news server, dedicated PPP server, and a mail/WWW machine) and a router might be a good start.

You need to make sure that you have all of your ducks in a row as far as the business end of the operation. How will you keep track of billing? What will you charge your customers? For what services will you charge extra?

Is Linux the right operating system for you? If you have experience with Berkeley-style systems, maybe NetBSD would be a better choice in the short run. If you are in a corporate environment and are setting up a network for your business, maybe you can spend the extra money for the technical support of Solaris or SCO. Then again, Linux has, in my opinion, a better range of services than BSD, and technical support for Linux is available from SSC and others, including several *LJ* advertisers. Instead of being locked into technical support from one vendor, you have a choice. And why would anyone run SCO?

You should not be under the illusion that becoming an ISP is easy. News will go haywire on you again and again if you are not an expert (you will become one or die trying). The business can be very competitive in different regions, and your dreams of wealth and glitzy nerd-dom might die the hard death of too much work and too little money.

Before investing the time and money in starting up an ISP business, you should be sure of your ability to do it. If in reading through the documentation mentioned above, you had trouble understanding it and are not confident in your ability to pull it off, maybe jumping right in is not the best decision. Again, Linux comes to the rescue. You can get a dial-up account with another ISP and set up a trial system on your PC at home. If you cannot handle a partial news feed, your own mail server, a DNS, and a web daemon, keep hacking at them until you can, and then reconsider starting up. I can think of few deaths worse than being condemned to run a network when you don't know what you're doing, especially if your life's savings are riding on it.

Finally, to all of you startup ISPs out there, I wish you good luck. With skill and hard work, it can be a rewarding business, and you get the satisfaction of meeting interesting customers and introducing them to the Internet. With the tools which Linux provides, there is no reason why you cannot build a first-rate network (and hopefully this article will help, too). Simply be aware that you are not alone in the ISP market, and your competitors will always be breathing down your neck. We might be one of them.

Happy Linuxing!

Todd Graham Lewis (tlewis@cheney.net) is Vice President of Networking at Cheney Communications Company, an ISP in Birmingham, AL. In his spare time he reads 19th-century literature and Linux Documentation. He is working on another HOWTO and Dostoevsky's *The Brothers Karamazov*.

The Devil's in the Details

Georg V. Zezschwitz

Alessandro Rubini

Issue #25, May 1996

This article, the third of five on writing character device drivers, introduces concepts of reading, writing, and using ioctl-calls.

Starting from the clean code environment of the two previous articles, we now turn to all the nasty interrupt stuff. Astonishingly, Linux hides most of this from us, so we do not need a single line of assembler...

Reading and writing

Right now, our magic **skel**-machine driver can load and even unload (painlessly, unlike in DOS), but it has neither read nor written a single character. So we will start fleshing out the **skel_read()** and **skel_write()** functions introduced in the previous article (under **fops and filp**). Both functions take four arguments:

```
Static int skel_read (struct inode *inode,
                      struct file *file,
                      char *buf, int count)
Static int skel_write (struct inode *inode,
                       struct file *file,
                       const char *buf,
                       int count)
```

The **inode** structure supplies the functions with information used already during the **skel_open()** call. For example, we determined from **inode->i_rdev** which board the user wants to open, and transferred this data—along with the board's base address and interrupt to the **private_data** entry of the file descriptor. We might ignore this information now, but if we did not use this hack, **inode** is our only chance to find out to which board we are talking.

The **file** structure contains data that is more valuable. You can explore all the elements in its definition in **<linux/fs.h>**. If you use the **private_data** entry, you find it here, and you should also make use of the **f_flags** entry—revealing to you, for instance, if the user wants blocking or non-blocking mode. (We explain this topic in more detail later on.)

The **buf** argument tells us where to put the bytes read (or where to find the bytes written) and **count** specifies how many bytes there are. But you must remember that every process has its own private address space. In kernel code, there is an address space common to all processes. When system calls execute on behalf of a specific process, they run in kernel address space, but are still able to access the user space. Historically, this was done through assembler code using the **fs** register; current Linux kernels hide the specific code within functions called **get_user_byte()** for reading a byte from user address space, **put_user_byte()** for writing one, and so on. They were formerly known as **get_fs_byte**, and only **memcpy_tofs()** and **memcpy_fromfs()** reveal these old days even on a DEC Alpha. If you want to explore, look in **<asm/segment.h>**.

Let us imagine ideal hardware that is always hungry to receive data, reads and writes quickly, and is accessed through a simple 8-bit data-port at the base address of our interface. Although this example is unrealistic, if you are impatient you might try the following code:

```
Static int skel_read (struct inode *inode,
                      struct file *file,
                      char *buf, int count) {
    int n = count;
    char *port = PORT0 ((struct Skel_Hw*)
                        (file->private_data));
    while (n--) {
        Wait till device is ready
        put_user_byte (inb_p (port), buf);
        buf++;
    }
    return count;
}
```

Notice the **inb_p()** function call, which is the actual I/O read from the hardware. You might decide to use its fast equivalent, **inb()**, which omits a minimal delay some slow hardware might need, but I prefer the safe way.

The equivalent **skel_write()** function is nearly the same. Just replace the **put_user_byte()** line by the following:

```
        outb_p (get_user_byte (buf), port);
```

However, these lines have a lot of disadvantages. What using them causes Linux to loop infinitely while waiting for a device that never becomes ready? Our driver should dedicate the time in the waiting loop to other processes, making use of all the resources in our expensive CPU, and it should have an input and output buffer for bytes arriving while we are not in **skel_read()** and corresponding **skel_write()** calls. It should also contain a time-out test in case of errors, and it should support blocking and non-blocking modes.

Blocking and Non-Blocking Modes

Imagine a process that reads 256 bytes at a time. Unfortunately, our input buffer is empty when **skel_read()** is called. So what should it do—return and say that there is no data yet, or wait until at least *some* bytes have arrived?

The answer is **both**. *Blocking* mode means the user wants the driver to wait till some bytes are read. *Non-blocking* mode means to return as soon as possible —just read all the bytes that are available. Similar rules apply to writing: *Blocking* mode means "Don't return till you can accept some data," while *non-blocking* mode means: "Return even if nothing is accepted." The **read()** and **write()**calls usually return the number of data bytes successfully read or written. If, however, the device is non-blocking and no bytes can be transferred, **-EAGAIN** is typically returned (meaning: *"Play it again, Sam"*). occasionally, old code may return **-EWOULDBLOCK**, which is the same as **-EAGAIN** under Linux.

Maybe now you are smiling as happily as I did when I first heard about these two modes. If these concepts are new for you, you might find the following hints helpful. Every device is opened by default in blocking mode, but you may choose non-blocking mode by setting the **O_NONBLOCK** flag in the **open()** call. You can even change the behaviour of your files later on with the **fcntl()** call. The **fcntl()** call is an easy one, and the man page will be sufficient for any programmer.

Sleeping Beauty

Once upon a time, a beautiful princess was sent by a witch into a long, deep sleep, lasting for a hundred years. The world nearly forgot her and her castle, twined about by roses, until one day, a handsome prince came, kissed her, and awakened her --and all the other nice things happened that you hear about in fairy tales.

Our driver should do what the princess did while it is waiting for data: sleep, leaving the world spinning around. Linux provides a mechanism for that, called **interruptible_sleep_on()**. Every process reaching this call will fall asleep and contribute its time slices to the rest of the world. It will stay in this function till another process calls **wake_up_interruptible()**, and this "prince" usually takes the form of an interrupt handler that has successfully received or sent data, or Linux itself, if a time-out condition has occurred.

Installing an Interrupt Handler

The previous article in this series showed a minimal interrupt handler, which was called **skel_trial_fn()**, but its workings were not explained. Here, we introduce a "complete" interrupt handler, which will handle both input to and output from the actual hardware device. Figure 1 shows a simple version of its concept: When the driver is waiting for the device to get ready (blocking), it goes to sleep by calling **interruptible_sleep_on()**. A valid interrupt ends this sleep, restarting **skel_write()**.

Figure 1 does not include the double-nested loop structure we need when working with an internal output buffer. The reason is that if we can perform only writing within the **skel_write()** function there is no need for an internal output buffer. But our driver should catch data even while not in **skel_read()** and should write the data in the background even when not in **skel_write()**. Therefore, we will change the hardware writing in **skel_write()** to write to an output buffer and let the *interrupt handler* perform the real writing to the hardware. The interrupt and **skel_write()** will now be linked by the “Sleeping Beauty” mechanism and the output buffer.

The interrupt handler is installed and uninstalled during the **open()** and **close()** calls to the device, as suggested in the previous article. This task is handled by the following kernel calls:

```
#include <linux/sched.h>
int request_irq(unsigned int irq,
                void (*handler)
                     (int, struct pt_regs *),
                unsigned long flags,
                const char *device);
void free_irq(unsigned int irq);
```

The **handler** argument is the actual interrupt handler we wish to install. The role of the **flags** argument is to set a few features of the handler, the most important being its behaviour as a *fast* handler (**SA_INTERRUPT** is set in **flags**) or as a *slow* handler (**SA_INTERRUPT** is *not* set). A fast handler is run with all interrupts disabled, while a slow one is executed with all interrupts except itself enabled.

Finally, the **device** argument is used to identify the handler when looking at /proc/interrupts.

The handler function installed by **request_irq()** is passed only the interrupt number and the (often useless) contents of the processor registers.

Therefore, we'll first determine which board the calling interrupt belongs to. If we can't find any boards, a situation called a *spurious* interrupt has occurred, and we should ignore it. Typically interrupts are used to tell whether the device is ready either for reading *or* writing, so we have to find out by one or more hardware tests what the device wants us to do.

Of course, we should leave our interrupt handler quickly. Strangely enough, **printk()** (and thus the **PDEBUG** line) is allowed even within fast interrupt handlers. This is a very useful feature of the linux implementation. If you look at kernel/printk.c you'll discover that its implementation is based on wait queues, as the actual delivery of messages to log files is handled by an external process (usually klogd).

As shown in figure 2, Linux can handle a timeout when in **interruptible_sleep_on()**. For example, if you have are using a device to which you send an answer, and it is expected to reply within a limited time, causing a time-out to signal an I/O error (**-EIO**) in the return value to the user process might be a good choice.

Certainly the user process could care for this, too, using the alarm mechanism. But it is definitely easier to handle this in the driver itself. The timeout criteria is specified by **SKEL_TIMEOUT**, which is counted in *jiffies*. Jiffies are the steady heartbeat of a Linux system, a steady timer incremented every few milliseconds. The frequency, or number of jiffies per second, is defined by **HZ** in **<asm/param.h>** (included in **<linux/sched.h>**) and varies on different architectures (100 Hz Intel, 1 kHz Alpha). You simply have to set

```
#define SKEL_TIMEOUT timeout_seconds * HZ
/* ... */
current->timeout = jiffies + SKEL_TIMEOUT
```

and if **interruptible_sleep_on** timed out, **current->timeout** will be cleared after return.

Be aware that interrupts might happen within **skel_read()** and **skel_write()**. Variables that might be changed within the interrupt should be declared as **volatile**. They also need to be protected to avoid race conditions. The classic code sequence to protect a critical region is the following:

```
unsigned long flags;
save_flags (flags);
cli ();
critical region
restore_flags (flags);
```

Finally, the code for the “complete” error handler:

```
#define SKEL_IBUFSIZ 512
#define SKEL_OBUFSIZ 512
/* for 5 seconds timeout */
#define SKEL_TIMEOUT (5*HZ)
/* This should be inserted in the Skel_Hw-structure */
typedef struct Skel_Hw {
    /* write position in input-buffer */
    volatile int ibuf_wpos;
    /* read position in input-buffer */
    int ibuf_rpos;
    /* the input-buffer itself */
    char *ibuf;
    /* write position in output-buffer */
    int obuf_wpos;
    /* read position in output-buffer */
    volatile int buf_rpos;
    char *obuf;
    struct wait_queue *skel_wait_iq;
    struct wait_queue *skel_wait_oq;
    [...]
}
#define SKEL_IBUF_EMPTY(b) \
 ((b)->ibuf_rpos==(b)->ibuf_wpos)
#define SKEL_OBUF_EMPTY(b) \
 ((b)->obuf_rpos==(b)->obuf_wpos)
#define SKEL_IBUF_FULL(b) \
 (((b)->ibuf_wpos+1)%SKEL_IBUFSIZ==(b)->ibuf_rpos)
#define SKEL_OBUF_FULL(b) \
 (((b)->obuf_wpos+1)%SKEL_OBUFSIZ==(b)->obuf_rpos)
Static int skel_open (struct inode *inode,
                      struct file *filp) {
    /* .... */
    /* First we allocate the buffers */
    board->ibuf = (char*) kmalloc (SKEL_IBUFSIZ,
                                   GFP_KERNEL);
    if (board->ibuf == NULL)
        return -ENOMEM;
    board->obuf = (char*) kmalloc (SKEL_OBUFSIZ,
                                   GFP_KERNEL);
    if (board->obuf == NULL) {
        kfree_s (board->ibuf, SKEL_IBUFSIZ);
        return -ENOMEM;
    }
    /* Now we clear them */
    ibuf_wpos = ibuf_rpos = 0;
    obuf_wpos = obuf_rpos = 0;
    board->irq = board->hwirq;
    if ((err=request_irq(board->irq>
                         skel_interrupt,
                         SA_INTERRUPT, "skel")))
        return err;
}
Static void skel_interrupt(int irq,
                    struct pt_regs *unused) {
    int i;
    Skel_Hw *board;
    for (i=0, board=skel_hw; i<skel_boards;
         board++, i++)
         /* spurious */
        if (board->irq==irq) break;
    if (i==skel_boards) return;
    if (board_is_ready_for_input)
        skel_hw_write (board);
    if (board_is_ready_for_output)
        skel_hw_read (board);
}
Static inline void skel_hw_write (Skel_Hw *board){
    int rpos;
    char c;
    while (! SKEL_OBUF_EMPTY (board) &&
        board_ready_for_writing) {
        c = board->obuf [board->obuf_rpos++];
        write_byte_c_to_device
        board->obuf_rpos %= SKEL_OBUF_SIZ;
    }
    /* Sleeping Beauty */
    wake_up_interruptible (board->skel_wait_oq);
}
Static inline void skel_hw_read (Skel_Hw *board) {
    char c;
    /* If space left in the input buffer & device ready: */
    while (! SKEL_IBUF_FULL (board) &&
        board_ready_for_reading) {
        read_byte_c_from_device
        board->ibuf [board->ibuf_wpos++] = c;
        board->ibuf_wpos %= SKEL_IBUFSIZ;
    }
    wake_up_interruptible (board->skel_wait_iq);
}
Static int skel_write (struct inode *inode,
                       struct file *file,
                       char *buf, int count) {
    int n;
    int written=0;
    Skel_Hw board =
        (Skel_Hw*) (file->private_data);
    for (;;) {
        while (written<count &&
               ! SKEL_OBUF_FULL (board)) {
            board->obuf [board->obuf_wpos] =
                get_user_byte (buf);
            buf++; board->obuf_wpos++;
            written++;
            board->obuf_wpos %= SKEL_OBUFSIZ;
        }
        if (written) return written;
        if (file->f_flags & O_NONBLOCK)
            return -EAGAIN;
        current->timeout = jiffies + SKEL_TIMEOUT;
        interruptible_sleep_on (
            &(board->skel_wait_oq));
        /* Why did we return? */
        if (current->signal & ~current->blocked)
        /* If the signal is not not being
           blocked */
            return -ERESTARTSYS;
        if (!current->timeout)
            /* no write till timout: i/o-error */
            return -EIO;
    }
}
Static int skel_read (struct inode *inode,
                      struct file *file,
                      char *buf, int count) {
    Skel_Hw board =
        (Skel_Hw*) (file->private_data);
    int bytes_read = 0;
    if (!count) return 0;
    if (SKEL_IBUF_EMPTY (board)) {
        if (file->f_flags & O_NONBLOCK)
            /* Non-blocking */
            return -EAGAIN;
        current->time_out = jiffies+SKEL_TIMEOUT;
        for (;;) {
            skel_tell_hw_we_ask_for_data
            interruptible_sleep_on (
                &(board->skel_wait_iq));
            if (current->signal
                & ~current->blocked)
                return -ERESTARTSYS;
            if (! SKEL_IBUF_EMPTY (board))
                break;
            if (!current->timeout)
                /* Got timeout: return -EIO */
                return -EIO;
        }
    }
    /* if some bytes are here, return them */
    while (! SKEL_IBUF_EMPTY (board)) {
        put_user_byte (board->ibuf
                          [board->ibuf_rpos],
                       buf);
        buf++; board->ibuf_rpos++;
        bytes_read++;
        board->ibuf_rpos %= SKEL_IBUFSIZ;
        if (--count == 0) break;
    }
    if (count) /* still looking for some bytes */
        skel_tell_hw_we_ask_for_data
    return bytes_read;
}
```

Handling select()

The last important I/O function to be shown is **select()**, one of the most interesting parts of Unix, in our opinion.

The **select()** call is used to wait for a device to become ready, and is one of the most scary functions for the novice C programmer. While its use from within an application is not shown here, the driver-specific part of the system call is shown, and its most impressive feature is its compactness.

Here's the full code:

```
Static int skel_select(struct inode *inode,
                       struct file *file,
                       int sel_type,
                       select_table *wait) {
    Skel_Clientdata *data=filp->private_data;
    Skel_Board *board=data->board;
    if (sel_type==SEL_IN) {
        if (! SKEL_IBUF_EMPTY (board))
            /* readable */
            return 1;
        skel_tell_hw_we_ask_for_data;
        select_wait(&(hwp->skel_wait_iq), wait);
        /* not readable */
        return 0;
    }
    if (sel_type==SEL_OUT) {
        if (! SKEL_OBUF_FULL (board))
            return 1;  /* writable */
        /* hw knows */
        select_wait (&(hwp->skel_wait_oq), wait);
        return 0;
    }
    /* exception condition: cannot happen */
    return 0;
}
```

As you can see, the kernel takes care of the hassle of managing wait queues, and you have only to check for readiness.

When we first wrote a **select()** call for a driver, we didn't understand the **wait_queue** implementation, and you don't need to either. You only have to know that the code works. **wait_queue**s *are* challenging, and usually when you write a driver you have no time to accept the challenge.

Actually, **select** is better understood in its relationships with read and write: if **select()** says that the file is readable, the next read must not block (independently of **O_NONBLOCK**), and this means you have to tell the hardware to return data. The interrupt will collect data, and awaken the queue. If the user is selecting for writing, the situation is similar: the driver must tell if **write()** will block or not. If the buffer is full it will block, but you don't need to tell the hardware about it, since **write()** has already told it (when it filled the buffer). If the buffer is not full, the write won't block, so you return 1.

This way to think of selecting for write may appear strange, as there are times when you need to write synchronously, and you may expect that a device is writable when it has already accepted pending input. Unfortunately, this way of doing things will break the blocking/nonblocking machinery, and thus an extra call is provided: if you need to write synchronously, the driver must offer (within its **fops**) the **fsync()**call. The application invokes **fops->fsync** through the **fsync()** system call, and if the driver doesn't support it, **-EINVAL** is returned.

ioctl()--Passing Control Information

Imagine that you want to change the baud-rate of a serial multiport card you have built. Or tell your frame grabber to change the resolution of an image. Or whatever else... You could wrap these instructions into a series of escape sequences, such as, for example, the screen positioning in ANSI emulation. But, the normal method for this is to make an **ioctl()** call.

ioctl() calls as defined in **<sys/ioctl.h>** have the form

```
ioctl (int file_handle, int command, ...)
```

where ... is considered to be one argument of the type **char *** (according to the **ioctl** man page). Strange as it may be, the kernel receives these arguments in **fs/ioctl.c** in the form:

```
int sys_ioctl (unsigned int fd, unsigned int cmd,
               unsigned long arg);
```

To add to the confusion, **<linux/ioctl.h>** gives detailed rules how the commands in the second parameter should be built, but nobody in all the drivers is actually following these ideas yet.

In any case, rather than cleaning up the Linux source tree, let's concentrate on the general *idea* of **ioctl()** calls. As the user, you pass the file handle and a command in the first two arguments and pass as the third parameter a pointer to a data structure the driver should read and/or write.

A few commands are interpreted by the kernel itself—for example, **FIONBIO**, which changes the blocking/non-blocking flag of the file. The rest is passed to our own, driver-specific **ioctl()** call, and arrives in the form:

```
int skel_ioctl (struct inode *inode,
                struct file *file,
                unsigned int cmd,
                unsigned long arg)
```

Before we show a small example of a **skel_ioctl()** implementation, the commands you define should obey the following rules:

1. Pick up a free MAGIC number from /usr/src/linux/MAGIC and make this number the upper eight bits of the 16-bit command word.
2. Enumerate commands in the lower eight bits.

Why this? Imagine “Silly Billy” starts his favorite terminal program minicom to connect to his mailbox. “Silly Billy” accidentally changed the serial line minicom uses from /dev/ttyS0 to /dev/skel0 (he is quite silly). The next thing minicom does is initialize the “serial line” with an **ioctl()** using **TCGETA** as command. Unfortunately, your device driver, hidden behind /dev/skel0, uses that number to control the voltage for a long-term experiment in the lab...

If the upper eight bits in the commands for **ioctl()** differ from driver to driver, every **ioctl()** to an inappropriate device will result in an **-EINVAL** return, protecting us from extremely unexpected results.

Now, to finish this section, we will implement an **ioctl()** call reading or changing the timeout delay in our driver. If you want to use it, you have to introduce a new variable

```
unsigned long skel_timeout = SKEL_TIMEOUT;
```

right after the definition of **SKEL_TIMEOUT** and replace every later occurrence of **SKEL_TIMEOUT** with **skel_timeout**.

We choose the **MAGIC '4'** (the ASCII character 4) and define two commands:

```
# define SKEL_GET_TIMEOUT 0x3401
# define SKEL_SET_TIMEOUT 0x3402
```

In our user process, these lines will double the time-out value:

```
/* ... */
unsigned long timeout;
if (ioctl (skel_hd, SKEL_GET_TIMEOUT,
           &timeout) < 0) {
    /* an error occurred (Silly billy?) */
    /* ... */
}
timeout *= 2;
if (ioctl (skel_hd, SKEL_SET_TIMEOUT,
           &timeout) < 0) {
    /* another error */
    /* ... */
}
```

And in our driver, these lines will do the work:

```
int skel_ioctl (struct inode *inode,
                struct file *file,
                unsigned int cmd,
                unsigned long arg) {
    switch (cmd) {
    case SKEL_GET_TIMEOUT:
        put_user_long(skel_timeout, (long*) arg);
        return 0;
    case SKEL_SET_TIMEOUT:
        skel_timeout = get_user_long((long*) arg);
        return 0;
    default:
        return -EINVAL; /* for Silly Billy */
    }
}
```

Georg V. Zezschwitz a 27-year-old Linuxer with a taste for the practical side of Computer Science and a tendency to avoid sleep.

XXXXXXXXXXXXXXXX (XXXXXXXXXXXXXX) Like Georg, is also 27-years-old and has the same interest in the practical side of Computer Science and the same tendency to avoid sleep.

Archive Index Issue Table of Contents

Advanced search

Keeping Track of Change

Michael K. Johnson

Issue #25, May 1996

Have you ever wanted to turn back time after making a mistake and irrevocably damaging a file you were editing? You can do so with minimal effort after reading this article.

For years, software developers have been keeping track of all their changes to programs they are developing with what they call “version control” software. However, even competent developers have sometimes found it too troublesome to use these tools except where they find it absolutely necessary, and so it is not surprising that very few non-developers have used these tools at all.

The reason that they find these tools so difficult to use is that *the rules for how to use them are written with the most complicated situations in mind.* For most personal use of these tools, that is like learning to fly a jet aircraft in order to drive a car. There is an easier way.

The assumption that makes everyone do so much extra work is that *more than one person will be trying to modify the file at the same time.* For your own personal files, or for keeping track of changes to system files when you are the only system administrator, that is not an issue.

The standard program for doing version control under Linux (and most other versions of Unix) is GNU's **RCS**, which stands for **R**evision **C**ontrol **S**ystem. It has many options, but you hardly need to know about any of them to make good use of it. You can treat RCS as one of the simplest tools on your system.

In fact, you can keep track of all the changes that you ever make to a file with one command. Run the command **ci -l *`filename`*** each time you make a change to the file (“ci” stands for “**c**heck **i**n”; you are “checking in your changes”). The first time you do this, you will be given the option to describe the file:

```
$ ci -l foo
foo,v  <--  foo
enter description, terminated with single '.'
  or end of file:
NOTE: This is NOT the log message!
>
```

You are not required to describe the file, but you can if you like. Then type a **.** character on its own line, or press **^D** on an empty line.

When you have made a change to the file, you need type only one command to tell RCS to keep track of that change for you:

```
$ ci -l foo
foo,v  <--  foo
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.'
  or end of file:
>
```

Here, you may wish to describe the change you have just made, especially if you think you will want to examine the change at some future time, but it is not required.

Remembering that single command is enough to prepare you for disaster. The rest of this short article will show you a few easy tricks that make RCS a *little* more comfortable to use, and help you learn how to recover from disaster.

Even Easier?

If you think running one command each time you finish editing a file is *still* too much work, we can make it even easier for you! (Laziness is a *virtue* if it causes you to work smarter instead of harder.)

We'll use a *shell script*. You can create a file in your own “bin” directory (if your account is on a computer shared with other people) or in /usr/local/bin (if you are using your own computer). Whichever directory you choose must be listed in your **PATH**. To make it easy to type, we'll call it **et**, short for edit text.

```
#!/bin/bash
# et - Edit Text file, while keeping
#      track of changes with RCS
[ -z "$1" ] && {
  # No file specified on the command line
  echo "Edit what file?"
  exit 1
}
${EDITOR-vi} $1
# Only check in the file if it exists.
[ -f $1 ] && ci -l $1
```

Type that into a file named **et**, without making any typos. Then run the command **chmod +x et** to make the file *executable*. Now, you will be able to run **et *filename*** to edit and then automatically check in your changes.

The **#!/bin/bash** line tells Linux this script is a shell script that is interpreted by the bash shell, which comes as a standard part of every Linux distribution. All the other lines starting with a **#** character are comments, and are ignored by bash.

If you decide that you will never want to include a description of the changes you have made, you can change the last line of the script to

```
[ -f $1 ] && echo "." | ci -l $1
```

The reason for the **[-f $1]** part is so that if you type **et** ***`mistake`***, and then quit the editor without saving the file, you won't get error messages when ci finds that there is no file for it to check in.

The **${EDITOR-vi}** part runs your favorite editor, or if you have not chosen a favorite editor, it defaults to the old standard vi editor. You can choose a default other than vi; pico, jed, joe, Emacs, and other editors are all possibilities. For example:

```
${EDITOR-jed} $1
```

will run jed, unless you have chosen some other favorite editor.

To choose your favorite editor, which will apply to all programs that want you to choose an editor, not just **et**, you will need to set the **EDITOR** environment variable to the name of the editor you want to use. If you are using a bourne shell, such as bash, zsh, pdksh, ksh, or sh, you will want to add a line like the following to the file .profile in your home directory:

```
EDITOR=jed ; export EDITOR
```

If you use the C shell (csh or tcsh), you will want to add a line like the following to the file .login in your home directory:

```
setenv EDITOR jed
```

How Does It Work?

RCS doesn't keep a whole new copy of the file each time you check in changes. Instead, it records only the lines with changes in them, along with the descriptions of the changes (if you choose to provide them). It does this in a separate file. Changes to the file *filename* are kept in the file *filename,v*. If you find this to be too much clutter, you can tell RCS to stash the *filename,v* file out of the way, in a subdirectory called RCS. Simply create the subdirectory, and RCS will automatically use it.

Remember what it looked like when the *foo* file was checked in, above? Let's create a *bar* file whose corresponding *bar,v* file is kept out of sight in an RCS directory:

```
$ mkdir RCS
$ vi bar
$ ci -l bar
RCS/bar,v  <--bar
enter description, terminated with single '.'
  or end of file:
NOTE: This is NOT the log message!
>
$ vi bar
$ ci -l bar
RCS/bar,v  <--  bar
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.'
  or end of file:
>
```

If you did not originally create an RCS directory, and you get tired of seeing the *,v* files, you can use these commands to get them out of the way safely:

```
$ mkdir RCS
$ mv *,v RCS
```

RCS will know where to search for them.

Turning Back the Clock

So what do you do when you screw up? You have carefully kept copies of your changes, but how do you retrieve yesterday's version, or last year's version, or the second most recent version?

RCS keeps track of versions by **version numbers**. The first version you check in is assigned the number **1.1**, the second is assigned the number **1.2**, the third **1.3**, and so on.

When you find that you have made a mistake, you can compare what you currently have with previous revisions. To compare against the previous revision, use this command:

```
$ rcsdiff -u filename
```

The **-u** tells rcsdiff to use the “**u**nified diff” format to show you the changes, and it compares the current version of the file *filename* with the most recent version that was checked in. Here's an example. The previously checked in version of the file foo, version 1.3, consisted of:

```
This is a test of the emergency
RCS system.  This is only a test.
```

I have since edited the current version to read:

```
This is a test of the emergency
RCS version control system.
This is only a test.
```

Before checking this new version in, I can check the differences between the current contents of the file and the previous version that was checked in, giving me this:

```
$ rcsdiff -u foo
=============================================
RCS file: foo,v
retrieving revision 1.3
diff -u -r1.3 foo
--- foo 1996/02/01 00:34:15     1.3
+++ foo 1996/02/01 00:34:31
@@ -1,2 +1,3 @@
 This is a test of the emergency
-RCS system.  This is only a test.
+RCS version control system.
+This is only a test.
```

After showing what versions are being compared, the differences are shown. Lines that have not been changed are printed with a single space in front of them. Lines that have been removed in the newer version have a **-** in front of them, and lines that have been added have a **+** in front of them. As you can see, lines that are changed are considered to be deleted from the older version, and changed replacements added to the new version. Around any section with changed line, up to 3 lines of unchanged “context” will be shown to help you understand where in the file the change has been made.

This mechanism can be used to compare any two versions. After making a few more changes to the file, I can compare revision 1.6 to revision 1.3:

```
$ 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.
```

Note that it is good to list the earlier revision first. Otherwise, the sense of **+** and **-** are reversed.

Your changes are likely to be much more significant than these examples, and may take up more than a screen listing. This is not a problem; use a pager to view your output one screenful at a time:

```
$ rcsdiff -u -r1.3 -r1.6 foo | less
```

Once you see the changes you have made, you can usually figure out where you made your mistake and fix it by hand. Sometimes you might have accidentally done damage that is large and confusing, but more often you have changed a phrase or a paragraph, are dissatisfied with your change, and simply can't remember exactly what it used to look like.

If you want to, you can store a copy of the changes in a file with a command like:

```
$ rcsdiff -u -r1.3 -r1.6 foo > filename
```

or print the changes like:

```
$ rcsdiff -u -r1.3 -r1.6 foo | lpr
```

Recovering from Massive Errors

If the time comes when you would rather simply revert to an older version, choose the version to which you wish to revert (call it **1.***x*) and run these commands:

```
$ ci -l foo
$ co -r1.x foo
RCS/foo,v  -->  foo
revision 1.x
writable foo exists; remove it? [ny](n): y
$ chmod +w foo
```

At this point, you have reverted to version 1.*x* of your file. From this point, you can continue making changes as if you had simply edited the latest version until it was identical to version 1.*x*.

Note that you should almost never have to do this; many people never have.

Looking for Old Changes

How do you figure out which versions to look at, when you are looking for one particular change? One command provides an instant summary of the changes that have been made since revision 1.1. This is where providing descriptions for important changes comes in handy. To see the log of all the changes, use the rlog command:

```
$ rlog bar
RCS file: RCS/bar,v
Working file: bar
head: 1.3
branch:
locks: strict
        johnsonm: 1.3
access list:
```

```
symbolic names:
keyword substitution: kv
total revisions: 3;     selected revisions: 3
description:
----------------------------
revision 1.3    locked by: johnsonm;
date: 1996/02/01 02:40:16;  author: johnsonm;
    state: Exp;  lines: +1 -0
Added different text.
----------------------------
revision 1.2
date: 1996/02/01 02:39:59;  author: johnsonm;
    state: Exp;  lines: +1 -0
Added some text.
----------------------------
revision 1.1
date: 1996/01/31 21:22:36;  author: johnsonm;
    state: Exp;
Initial revision
=============================================
```

The most important parts for you to understand are the revision descriptions at the end. For example, under the **revision 1.2** section, there is a comment **Added some text.** This is the comment that you are allowed to type in after every revision. If you choose not to enter a comment, it says instead:

```
*** empty log message ***
```

which isn't particularly helpful when you are looking for a change you made.

On the other hand, if you find entering a comment to be so much work that you are tempted not to use RCS, you are better off when things go wrong to be able to go looking for the version you want than to have no information whatever. Don't avoid using RCS because you feel you ought to describe every revision...

These logs become very long, very quickly. In order to look at the log one screen at a time, use a pager program:

```
$ rlog bar | less
```

Again, you can send the log to a file or print it, if you prefer.

Conclusion

There is a lot more to RCS than can be found in this article, since this is a tutorial intended to make it easy for you to use RCS. If you are interested, RCS comes with a full complement of manual pages, as well as papers about how to use RCS in a development environment. In addition, *Linux Journal* had an earlier article on RCS, aimed more at developers, in the February 1995 issue. But don't think that you have to know everything about RCS in order to use it effectively.

Michael K. Johnson (XXXXXXXXXXXXXXX) is the Editor of *Linux Journal*, and uses RCS in the way described in this article to keep track of all the changes that he makes while editing articles for *Linux Journal*.

IPv6: The New Internet Protocol

Danny Yee

Issue #25, May 1996

Anyone interested in the technical details of IPv6 will want a copy—even if you are prepared to wade through the relevant RFCs, IPv6 provides annotated references to these and other important papers at the end of each chapter.

IPv6: The New Internet Protocol

Author: Christian Huitema

Publisher: Prentice Hall 1996

ISBN: 0-13-241936-X

Price: $44.00

Reviewer: Danny Yee

Huitema's *IPv6* is a concise but comprehensive description of the new Internet protocol. It begins with a very brief account of the motivation for a new protocol and the background to its selection (the competition between the different contenders), then plunges straight into the technical details: a chapter describing the basic packet format; one on routing and addressing; chapters on auto-configuration, security, and support for flows; a chapter on transition issues; and a final chapter in which Huitema offers his personal opinion of the major decisions made in the protocol design.

Each of the chapters goes into some detail. The chapter on security, for example, describes the Photuris key exchange system quite thoroughly, while the chapter on flows enters a little into the issues of fair queuing. Each chapter also discusses the points which were controversial in the decision process: such things as the length of the addresses, the mandation of potentially unexportable security support, the relationship between IP and ATM, and the choice of a dual-stack approach to IPv4-IPv6 integration rather than use of header translation. I felt that *IPv6* had much more meat to it than Bradner and Mankin's longer *IPng* (Addison-Wesley 1995), but the two books are really complementary, with the latter dealing more with the historical context and the framework within which the decision was made than with IPv6 itself (the difference in titles is appropriate).

IPv6 is a very nice little volume, marred only by poor proof-reading—there were far too many simple grammatical mistakes, and at least one spelling error which any automated spell-checker should have found. Anyone interested in the technical details of IPv6 will want a copy—even if you are prepared to wade through the relevant RFCs, *IPv6* provides annotated references to these and other important papers at the end of each chapter.

Disclaimer: I received a review copy of IPv6: The New Internet Protocol *from Prentice Hall, but I have no stake, financial or otherwise, in its success.*

Danny Yee (danny@cs.su.oz.au) All book reviews by Danny Yee are available via anonymous FTP: anatomy.sy.oz.au in /danny/book-reviews (index INDEX).

Archive Index Issue Table of Contents

Advanced search

New Products

LJ Staff

Issue #25, May 1996

Linux Internet Archives, Visual SlickEdit Available For Linux and more.

Linux Internet Archives

Yggdrasil Computing, Inc. has announced the availability of its new Linux CD-ROM set. The Winter 1996 edition of Linux Internet Archives contains six CDs. These CDs include the free Linux software from Tsx-11 and Sunsite, the GNU archive on prep.ai.mit.edu, the X11R6 archives including the free contributed X11R6 software from ftp.x.org, the Internet RFC standards, and a total of nine non-Yggdrasil Linux distributions. Price: $22.95 plus shipping.

Contact: Yggdrasil Computing, Inc., 4880 Stevens Creek Blvd., Suite 205, San Jose, CA 95129-1034. Phone: (800) 261-6630. Fax: (408) 261-6631. E-mail: orders@yggdrasil.com. URL: www.yggdrasil.com.

Visual SlickEdit Available For Linux

MicroEdge, Inc. has announced the release of Visual SlickEdit for X-Windows. Visual SlickEdit is available on several Unix platforms including AIX RS6000, HP-UX, Solaris-Sparc, Solaris-Intel, SunOS, SGI Irix, Digital Unix, and Linux. The release of Visual SlickEdit for X-Windows signals the first time that a single graphical programming editor has achieved true cross-platform status. Visual SlickEdit is available for OS/2, Windows, Windows95,and Windows NT. Per-user prices: Linux, $195; X-Windows, $395; Windows, $295; OS/2, $219.00.

Contact: MicroEdge, Inc., P.O. Box 18038, Raleigh, NC 27619-8038. Phone: (800) 934-EDIT. Fax: (919) 831-0101.

Emulus—Terminal Emulation Software

SAS Institute has announced the availability of Emulus, its popular 3270 terminal emulation software, for the Linux operating system on Intel-based hardware. Emulus is an X-Windows/Motif application which uses TCP/IP to establish a connection to an IBM mainframe host, emulating a 3270 terminal. Included with Emulus is Helplus, an interactive hypertext help system modeled after Microsoft's WinHelp. Helplus is used to provide on-line Help for Emulus, but can also be used to build Help files for use with other X applications. It accepts the same input as WinHelp (help project files and RTF source), reads either XPM or GIF graphics files, and also supports standard WinHelp features such as secondary windows, pop-up topics, topic annotations, bookmarks, and most WinHelp macros. Price: $99.00 on CD-ROM for Linux.

Contact: SAS Institute, Book Sales division, (919) 677-8000.

Archive Index Issue Table of Contents

Advanced search

Consultants Directory

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

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

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

http://www.acay.com.au

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

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

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

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

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

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

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

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

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

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

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

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

Ian T. Zimmerman
Linux consulting.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Archive Index Issue Table of Contents

Advanced search