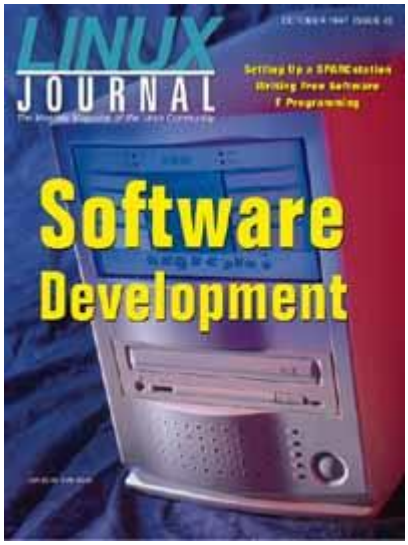
Linux Journal Issue #42/October 1997



Features

Literate Programming Using Noweb *by Andrew Johnson and Brad Johnson*

An introduction to Noweb, a tool designed to aid the programmer in producing understandable and easy to maintain code.

Remote Procedure Calls in Linux *by Ed Petron*

An introduction to this vital software development technique.

Xmotd: Writing Free Software *by Luis Fernandes*

This message-of-the-day browser was written to ease the burden of the local system administrator.

Portability and Power with the F Programming Language *by Walt Brainerd, David Epstein and Dick Hendrickson*

The authors combine over forty years of language-design committee experience to create the world's most portable, yet efficient, powerful, yet simple programming language.

News & Articles

Setting up a SPARCstation *by John Little*

LJ Interviews Thomas Roell *by Marjorie Richardson*

PostScript: The Forgotten Art of Programming *by Hans DeVreught*

Linux and the Alpha *by David Mosberger*

Reviews

Product Review SpellCaster DataCommute/BRI ISDN Adaptor *by Jay Painter*
Book Review Internet Programming with Python *by Dwight Johnson*
Book Review Unix Programming Tools *by Andrew L. Johnson*
Book Review Advanced Programming in the Unix Environment *by David Bausum*
Book Review Apache: The Definitive Guide *by Luca Cott Ramusino*

WWWsmith

Linux as an Internet Kiosk *by Kevin McCormick*
At the Forge Integrating SQL with CGI, Part 1 *by Reuven Lerner*

Columns

Letters to the Editor
From the Publisher Internet Changes/Linux Changes *by Phil Hughes*
Stop the Presses What Price High-Performance I/O? *by Phil Hughes*
Linux Apprentice DDD—The Data Display Debugger *by Shay Rojansky*
Take Command cat *by Patrick Hill*
Linux Means Business Grundig TV-Communications *by Ted Kenney*
New Products
System Administration Pgfs: The PostGres File System *by Brian Bartholomew*
Kernel Korner Kernel-Level Exception Handling *by Joerg Pommnitz*
Linux Gazette The Dotfile Generator *by Jesper K Pedersen*
Best of Technical Support

Archive Index

Advanced search

LINUX JOURNAL

Literate Programming Using Noweb

Andrew L. Johnson

Brad C. Johnson

Issue #42, October 1997

Noweb is a tool designed to enable a programmer to write documentation and code at the same time, with the goal of producing code that is easy to understand and maintain.

In essence, the purpose of literate programming (LP) can be found in the following quote:

> "Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *humans* what we want the computer to do."—Donald E. Knuth, 1984.

Such an environment reverses the notion of including documentation, in the form of comments within the code, to one where the code is embedded within a program's description. In doing so, literate programming facilitates the development and presentation of computer programs that more closely follow the conceptual map from the problem space to the solution space. This, in turn, leads to programs that are easier to debug and maintain.

When writing literate programs, one specifies the program description and the program code in a single source file, in the order best suited to human understanding. The program code can be extracted and assembled from this file into a form which the compiler or interpreter can understand—a process called "tangling". Documentation is produced by "weaving" the description and code into a form ready to be typeset (most often by TeX or LATeX).

Many different tools have been created for literate programming over the years. Most of the more popular are based, either directly or conceptually, on the WEB system created by D. E. Knuth ("Literate Programming", *The Computer*

Journal (27)2:97-111, 1984). This article focuses on Norman Ramsey's **noweb**—a simple to use, extensible literate programming tool that is independent of the target programming language.

Overview of the Noweb System

When you write a literate program using noweb, you create a simple text file (which by convention has a **.nw** extension) in which you provide all of the technical documentation for the various parts of the program, along with the actual source code for each part of the program.

February 13, 1997 autodefs.perl.nw 2

Processing the code chunk To process the code chunk we need to perform a few housekeeping tasks. First, we only want to consider lines that begin with $code_line_pat and second, we want to stop when we find a line that matches $end_code_pat. The following loop will suffice for this purpose.

2a ⟨process_code_chunk *subroutine* 2a⟩≡ (1d)

```
sub process_code_chunk {
    while ( ($_ = <>) && !/$end_code_pat/o ) {
        print $_;
        if( /$code_line_pat/o ) {
            ⟨Find and print any definitions 2b⟩
        }
    }
    print $_; # make sure we print the ``@end code'' line
}
```

Defines:
process_code_chunk, used in chunk 1d.
Uses $code_line_pat 1b and $end_code_pat 1a.

When checking for definitions we first strip off any comments since sub or package may also occur in a comment. We then build a list @def_list which contain all of the sub and package definitions on the line and print out an @index defn line for each.

2b ⟨Find and print any definitions 2b⟩≡ (2a)

```
$_ =~ s/#.*//o;
@def_list = (/sub\s(\w+)/go, /package\s(\w+)/go);
foreach $item (@def_list) {
    print "$index_prefix $item\n";
}
```

Uses $index_prefix 1c.

Defined Chunks

⟨autodefs.perl 1d⟩ 1d
⟨Find and print any definitions 2b⟩ 2a, 2b
⟨Global variables 1a⟩ 1a, 1b, 1c, 1d
⟨process_code_chunk subroutine 2a⟩ 1d, 2a

Index

$begin_code_pat: 1a, 1d
$code_line_pat: 1b, 2a
$end_code_pat: 1a, 2a
$index_prefix: 1c, 2b
process_code_chunk: 1d, 2a

Figure 1. Typeset Version of Perl Script

This file (Listing 1), which we call the nw source file, is then processed by **noweave** to create the documentation in a form ready for typesetting (the typeset version of the program is shown in Figure 1), or by **notangle** to extract the code chunks and assemble them in their proper order for the compiler or interpreter (the executable version of the program is in Listing 2). These two processes are not stand-alone programs, but a set of filters through which the nw source file is piped. It is this pipeline system that makes noweb both flexible and extensible, since the pipelines can be modified and new filters can be created and inserted in the pipelines to change the behavior of noweb.

Like most literate programming tools, noweb depends on TeX or LATeX—(LA)TeX to refer to either—for typesetting the documentation (although it has options for producing HTML output as well). However, one need not be a (LA)TeX guru to produce good results. All of the hard work of cross-referencing, indexing and typesetting the code is handled automatically by noweave.

The Typeset Documentation

The best way to get a feel for the capabilities of noweb is by reference to the finished product: the typeset version of a program. Figure 1 represents the typeset version of a Perl script that actually extends noweb's functionality by providing a limited “autodefs” filter. This filter will recognize and mark package and subroutine names for automatic cross-referencing and indexing.

When looking at this example, one can quickly see how chunks of actual code are interspersed throughout the descriptive text. Each code chunk is uniquely identified by page number and an alphabetic sub-page reference. For example, in Figure 1, there are four code chunks on the first page labeled in the left margin as 1a, 1b, 1c and 1d.

Besides the marginal tag, the first line of each code chunk also has its name and a chunk reference enclosed in angle brackets at the left margin and perhaps cross-reference information at the right margin. Lets examine chunk 1b more closely—a reasonable facsimile of its first line is:

```
1b <
```

This line tells us that we are now in chunk 1b. The **<*Global variables* 1a>+=** construct tells us we are working on the chunk named *Global variables* whose definition begins in chunk 1a. The **+=** indicates that we are adding to the definition of *Global variables*. At the right margin we encounter **(1d) <1a 1c>**, which means that the chunk we are defining is used in chunk 1d, and that the current chunk is continued from chunk 1a and will be further continued in chunk 1c. It should be noted that all of these visual cross-referencing clues—

with the exception of the chunk name itself—are provided automatically by noweb.

At the end of any chunk there are two optional footnotes—a “Defines” footnote and a “Uses” footnote. A user can manually specify, in the nw source file, a list of identifiers (i.e., variables or subroutines) which are defined in the current chunk. In addition, some identifiers may be automatically recognized, if an “autodefs” filter for the programming language is used. There are autodefs filters available for many languages including C, Icon, TeX, yacc and Pascal).

These identifiers are listed in the “Defines” footnote below the chunk where their definition occurs, along with a reference to any chunks which use them. Any occurrence of a defined identifier is referenced in a “Uses” footnote below the chunk that uses that identifier.

For example, in Figure 1, we see that chunk 1c defines the term **$index_prefix** which is used in chunk 2b. A quick peek at chunk 2b verifies that, indeed, this term is used and appears in the “Uses” footnote for that chunk.

Chunk 1d, autodefs.perl, represents the top level description of our entire program. This chunk is referred to as a “root” chunk in noweb and is not used in any other chunk. Our example has but one root chunk, although as many as you wish can be defined in your nw source file, and notangle can extract each of them into separate files.

The first line of code in chunk 1d is the obligatory **#!/usr/bin/perl** line which must begin all Perl scripts intended to be invoked as an executable program. However, the next two lines are not lines of Perl code at all but instead are references to other named chunk definitions. The code from those referenced chunks will be inserted at this point in the executable program extracted by notangle. Thus, we have a broad overview of our program, uncluttered by the specific global variable initializations and subroutine definitions.

Looking at chunk 2a, which is included in our root chunk, we see that it also includes another chunk, chunk 2b. This demonstrates that the inclusion of chunks can be nested to practically any level and can occur in any order in the documentation (definitions need not precede uses).

Our documentation ends with two optional indices provided by noweb—an index of code chunks and an index of identifiers.

Writing the Program in Noweb

With the knowledge of what comes out the end of the pipeline in hand, we can now describe the structure of the nw source file itself. The nw source file for our example program is given in Listing 1.

When you write a noweb program, you alternate between explaining some piece of code and providing the formal definition of that piece of code. You must indicate whether you are entering documentation or code by using one of two noweb tags.

To begin writing documentation, one starts with an @ symbol in the left column, followed by either a space or a newline. This indicates that all of the text that follows, at least up to the next tag, is documentation text. All documentation text is passed through the filtering process to the (LA)TeX file. Thus, the author is responsible for providing any special formatting such as sections, tables, footnotes and mathematical formulae which may be desired or needed in the documentation.

In addition to the standard (LA)TeX command set, noweb provides three additional control sequences. Any text surrounded by double square brackets in the text is typeset in the same fashion as literal code, and the **\nowebindex** and \nowebchunks commands expand into the two types of indices shown at the end of our example in Figure 1.

To indicate the beginning of a code chunk, you use double angle brackets surrounding a name for the code chunk followed by an equal sign:

```
<<code_chunk_name>>=
```

Everything following this construct is considered to be literal code or a reference to another chunk name. You reference another chunk name by placing its name in double angle brackets with no trailing equal sign. As with documentation, a code chunk is terminated when another tag is encountered. To continue a code chunk definition, you start a new code chunk using the same name within the brackets as the chunk to be continued.

The special formatting and cross-referencing of code chunks is handled automatically by noweb and requires no special input by the user—with the one exception of manually specifying identifier definitions.

To manually list identifiers which are defined in a given chunk, you terminate that chunk with a line of the form:

```
@ %def
```

The identifiers given on the line will be placed in a “Defines” footnote for that chunk and will automatically be cross-referenced and indexed by noweb as described in the previous section.

The process by which notangle extracts the code into a form suitable for the compiler or interpreter follows just a few simple rules. A root chunk is specified on the command line as the chunk to be extracted and assembled. This chunk is then printed line by line until a reference to another chunk is encountered. At this point, the referenced chunk is output line by line—and similarly for any chunks referenced therein. When the referenced chunk has been output, notangle continues the process of outputting the root chunk.

When dealing with continued chunks—two or more chunks sharing the same name—notangle concatenates their definitions in order of appearance into a single, named chunk. The extracted code for our example program is in Listing 2, and it can be seen that all spacing and indentation is preserved appropriately in the executable version.

Because of the way notangle extracts and assembles its input, the program can be presented and explained in the best order for human understanding. notangle will make sure that the program chunks are in the right order for the compiler or interpreter.

The Incantations

Now that we know how to create a program in noweb, we can examine the methods of generating our typeset and executable versions of the program. The noweb distribution provides a general shell script called, remarkably, noweb which drives the notangle and **noweave** processes. However, this method of invocation, though simple, is somewhat limited. We will focus here on using each tool separately as this method provides a more flexible approach.

When you invoke notangle, you specify a chunk name (a root chunk) to extract and assemble from the nw source file. If you fail to specify a chunk, notangle searches for a chunk named * to extract (this is the default root chunk in a noweb program). The notangle tool writes to stdout, so you must redirect this to a file of your choice. The general form of the command is:

```
notangle [-R
        [-filter
```

Thus, to extract the executable version of our example program we use:

```
notangle -Rautodefs.perl autodefs.perl.nw >
         autodefs.perl
```

The **-R** option specifies which root chunk to extract. The **-L** option is used to embed line directives, if they are supported by the compiler/interpreter you are using. The line directives refer to locations in the nw source file. Thus, when debugging your code, you never need to refer to the executable version. Rather, you can edit the code in the nw source file. The default format is for use with the C preprocessor, but it also works well with Perl, with one catch. The line directives are emitted whenever a chunk is entered or returned to, and refer to the next line of code. Therefore, in a script such as ours, a line directive winds up as the first line of the executable version before the **#!** line, rendering it non-executable. The fix for this is to delete the first line directive, or to move it below the first line and increment the line number by one.

One can write filters for use with either notangle or noweave that manipulate the source once it is in the pipeline. The pipeline representation of the nw source file in noweb is beyond the scope of this article (see the “Noweb Hacker's Guide” included in the documentation of the distribution). However, it should be noted that a filter could easily be constructed which automates the solution to the line directive problem.

The typeset version of the program is generated with the noweave tool. There are several useful options for noweave, all detailed in the man pages. We will only consider a few of the most important options here.

The first option of general interest concerns the desired output: you can specify **-latex** (default), **-tex** or **-html** as the formatting language to be used for the final documentation. Each of these options will supply an appropriate wrapper (which can be suppressed with the **-n** option) for the typeset version. You can write your nw source file intended for LATeX typesetting and still have the option of producing an HTML document by invoking noweave with the **-html** option and the LATeX-to-html filter (**-filter l2h**) included with the distribution.

The **-x** option enables cross-referencing and indexing of chunk names, as well as any identifiers which are automatically recognized by an “autodefs” filter. Using the **-index** option implies **-x** and provides cross-referencing and indexing for manually defined identifiers—those mentioned in **@ %def** statements in the nw source file.

Normally, noweave inserts additional information, such as the filename for use in page headers, as part of its wrapper. The **-delay** option causes noweave to suspend the insertion of this information until after the first documentation chunk. This is most useful when you wish to provide your own (LA)TeX wrapper to specify additional packages or to define your own special formatting commands. This implies a **-n** (omission of wrapper) option and requires that you make sure to include a **\end{document}** control sequence in a

documentation chunk at the end of the file to complete the wrapper. Our example nw source file is written in this fashion.

Our typeset version (Figure 1) was produced by first extracting the autodefs.perl root chunk with notangle and making it executable using the **chmod** system command. We then placed this executable in the noweb library directory and invoked noweave as:

```
noweave -autodefs perl -delay -index
        autodefs.perl.nw > autodefs.tex
```

We can then run LATeX on the resulting file—twice, to resolve page references—to create the dvi file, then use dvips to create the postscript version for inclusion with this article.

Additional options allow you to have the index created from an external file, to expand tabs and to specify alternative formatting options provided by the included noweb.sty file. The latter includes options to omit chunk numbering in the left margins, change text size in code chunks and switch from the symbolic cross-referencing of code chunks occurring at the right margin to simple footnote style cross-referencing similar in style to the “Defines” and “Uses” footnotes.

Conclusion

In general, a literate program takes more time and effort to initially produce. However, since much of this initial effort is devoted to explaining each part of the program, the author is likely to produce a better quality program in the end, because she has put more thought into the program's design at each stage of the game. Additionally, by investing in the extra effort of creating a well-documented program, the time spent later in maintaining and upgrading the program is considerably lessened.

In terms of documentation and explanation, the ability to describe components as they come into play in the design of the program—rather than in the order they must occur for the compiler or interpreter—is a vast improvement over traditional commented code. In addition to the benefits of improved code and easier maintenance, literate programs can also serve well as teaching tools.

Availability and Notes

Andrew Johnson is currently a full time student working on his Ph.D. in Physical Anthropology and a part time programmer and technical writer. He resides in Winnipeg, Manitoba with his wife and two sons, and he enjoys a good dark ale whenever he can. He can be reached at ajohnson@gpu.srv.ualberta.ca.

Brad Johnson is currently pursuing a degree in Statistics at the University of Manitoba.

Archive Index Issue Table of Contents

Advanced search

Remote Procedure Calls

Ed Petron

Issue #42, October 1997

A thorough introduction to RPC for programmers of distributed systems.

As any programmer knows, procedure calls are a vital software development technique. They provide the leverage necessary for the implementation of all but the most trivial of programs. Remote procedure calls (RPC) extend the capabilities of conventional procedure calls across a network and are essential in the development of distributed systems. They can be used both for data exchange in distributed file and database systems and for harnessing the power of multiple processors. Linux distributions provide an RPC version derived from the RPC facility developed by the Open Network Computing (ONC) group at Sun Microsystems.

RPC and the Client/Server Model

In case the reader is not familiar with the following terms, we will define them here since they will be important in later discussion:

- **Caller**: a program which calls a subroutine
- **Callee**: a subroutine or procedure which is called by the caller
- **Client**: a program which requests a connection to and service from a network server
- **Server**: a program which accepts connections from and provides services to a client

There is a direct parallel between the caller/callee relationship and the client/server relationship. With ONC RPC (and with every other form of RPC that I know), the caller always executes as a client process, and the callee always executes as a server process.

The Remote Procedure Call Mechanism

In order for an RPC to execute successfully, several steps must take place:

1. The caller program must prepare any input parameters to be passed to the RPC. Note that the caller and the callee may be running completely different hardware, and that certain data types may be represented differently from one machine architecture to the next. Because of this, the caller cannot simply feed *raw* data to the remote procedure.
2. The calling program must somehow pass its data to the remote host which will execute the RPC. In local procedure calls, the target address is simply a machine address on the local processor. With RPC, the target procedure has a machine address combined with a network address.
3. The RPC receives and operates on any input parameters and passes the result back to the caller.
4. The calling program receives the RPC result and continues execution.

External Data Representation

As was pointed out earlier, an RPC can be executed between two hosts that run completely different processor hardware. Data types, such as integer and floating-point numbers, can have different physical representations on different machines. For example, some machines store integers (C ints) with the low order byte first while some machines place the low order byte last. Similar problems occur with floating-point numeric data. The solution to this problem involves the adoption of a standard for data interchange.

One such standard is the ONC external data representation (XDR). XDR is essentially a collection of C functions and macros that enable conversion from machine specific data representations to the corresponding standard representations and vice versa. It contains primitives for simple data types such as int, float and string and provides the capability to define and transport more complex ones such as records, arrays of arbitrary element type and pointer bound structures such as linked lists.

Most of the XDR functions require the passing of a pointer to a structure of "XDR" type. One of the elements of this structure is an enumerated field called **x_op**. It's possible values are **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**. The **XDR_ENCODE** operation instructs the XDR routine to convert the passed data to XDR format. The **XDR_DECODE** operation indicates the conversion of XDR represented data back to its local representation. **XDR_FREE** provides a means to deallocate memory that was dynamically allocated for use by a variable that is no longer needed. For more information on XDR, see the information sources listed in the References section of this article.

RPC Data Flow

The flow of data from caller to callee and back again is illustrated in Figure 1. The calling program executes as a client process and the RPC runs on a remote server. All data movement between the client and the network and between the server and the network pass through XDR filter routines. In principle, any type of network transport can be used, but our discussion of implementation specifics centers on ONC RPC which typically uses either Transmission Control Protocol routed by Internet Protocol (the familiar TCP/IP) or User Datagram Protocol also routed by Internet Protocol (the possibly not so familiar UDP/IP). Similarly, any type of data representation could be used, but our discussion focuses on XDR since it is the method used by ONC RPC.

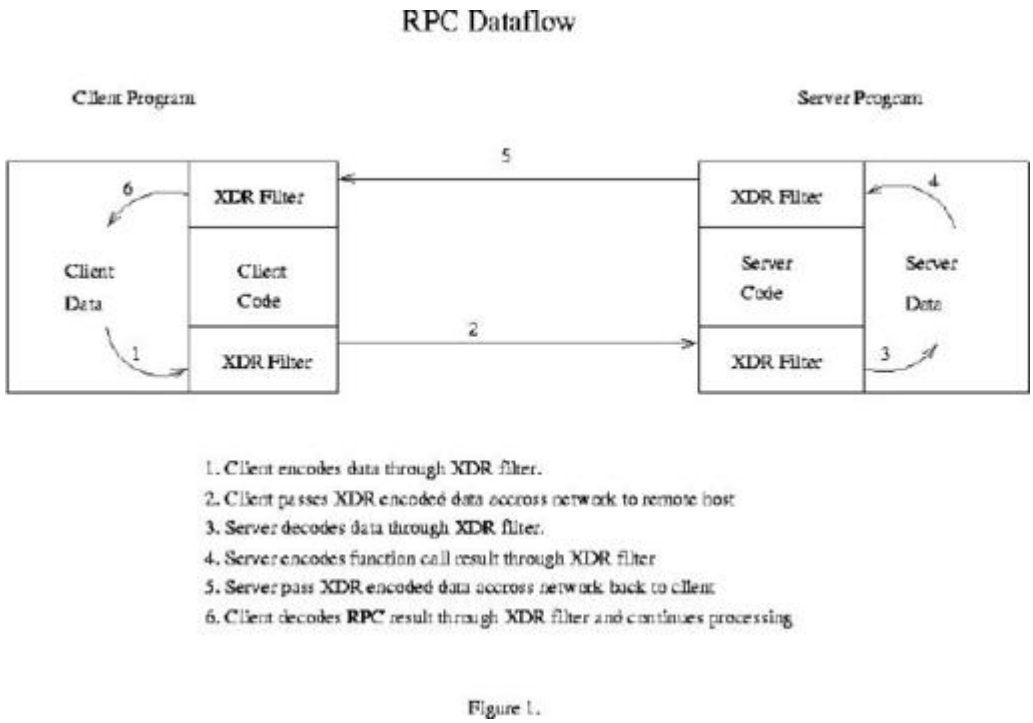


Figure 1. RPC Data Flow

Review of Network Programming Theory

In order to complete our picture of RPC processing, we'll need to review some network programming theory.

In order for two processes running on separate computers to exchange data, an **association** needs to be formed on each host. An association is defined as the following 5-tuple: *{protocol, local-address, local-process, foreign-address, foreign-process}*

The *protocol* is the transport mechanism (typically TCP or UDP) which is used to move the data between hosts. This, of course, is the part that needs to be common to both host computers. For either host computer, the *local-address/*

process pair defines the endpoint on the host computer running that process. The *foreign-address/process* pair refers to the endpoint at the opposite end of the connection.

Breaking this down further, the term *address* refers to the network address assigned to the host. This would typically be an Internet Protocol (IP) address. The term *process* refers not to an actual process identifier (such as a Unix PID) but to some integer identifier required to transport the data to the correct process once it has arrived at the correct host computer. This is generally referred to as a **port**. The reason port numbers are used is that it is not practical for a process running on a remote host to know the PID of a particular server. Standard port numbers are assigned to well known services such as TELNET (port 23) and FTP (port 21).

RPC Call Binding

Now we have the necessary theory to complete our picture of the RPC binding process. An RPC application is formally packaged into a *program* with one or more *procedure* calls. In a manner similar to the port assignments described above, the RPC program is assigned an integer identifier known to the programs which will call its procedures. Each procedure is also assigned a number that is also known by its caller. ONC RPC uses a program called **portmap** to allocate port numbers for RPC programs. It's operation is illustrated in Figure 2. When an RPC **program** is started, it registers itself with the portmap process running on the same host. The portmap process then assigns the TCP and/or UDP port numbers to be used by that application.

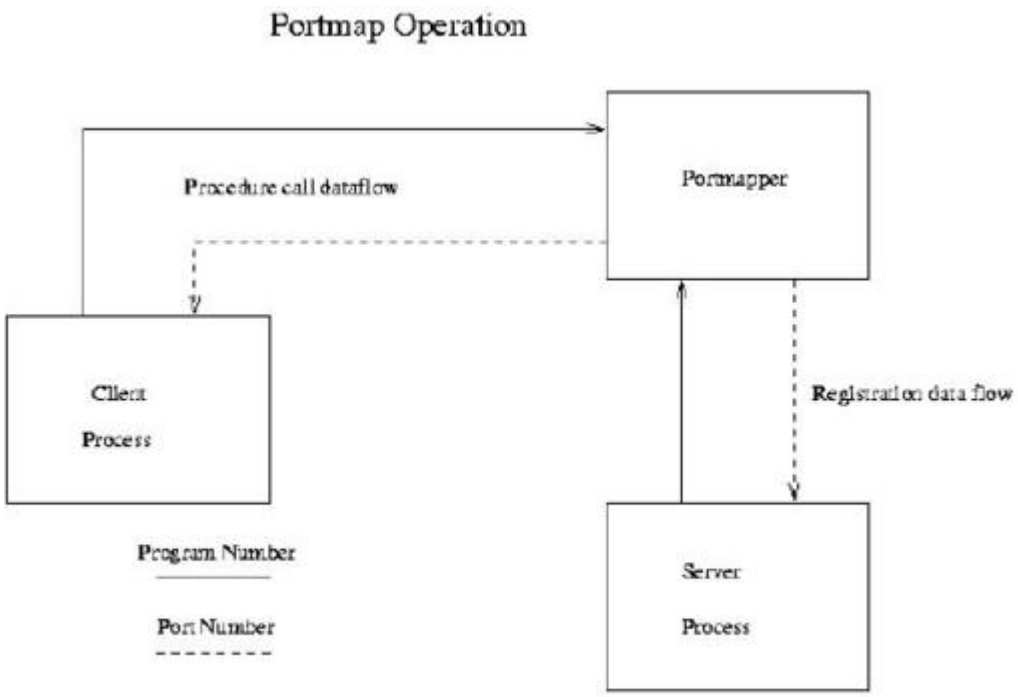


Figure 2

Figure 2. Portmap Operation

The RPC application then waits for and accepts connections at that port number. Prior to calling the remote procedure, the *caller* also contacts portmap in order to obtain the corresponding port number being used by the application whose procedures it needs to call. The network connection provides the means for the caller to reach the correct program on the remote host. The correct procedure is reached through the use of a dispatch table in the RPC program. The same registration process that establishes the port number also creates the dispatch table. The dispatch table is indexed by procedure number and contains the addresses of all the XDR filter routines as well as the addresses of the actual procedures.

RPCGEN: The Protocol Compiler

Listing 1. Source for avg.x

If the discussion of the mechanisms supporting RPC sounds complex, that's because it is. Fortunately, the development of RPC applications can be greatly simplified through the use of **rpcgen**, the protocol compiler. rpcgen has its own input language which is used to declare programs, their procedures and the data types for the procedures' parameters and return values. This is best illustrated by an example. The source code for an average procedure is shown in Listing 1. If we store this source code in a file called avg.x and invoke rpcgen with the following command:

```
rpcgen avg.x
```

Obtain the header file avg.h shown in Listing 2. This file contains all of the function prototypes and data declarations needed for the development of our application. It will also generate three other source files:

1. **avg_clnt.c**: the stub program for our client (caller) process
2. **avg_svc.c**: the main program for our server (callee) process
3. **avg_xdr.c**: the XDR routines used by both the client and the server

These sources are to be used “as is” and must not be edited.

Listing 2. Header File avg.h

To complete the application at the server end, we need code to provide the actual “smarts” required to correctly process the input data. This must be created manually. The code for the sample application presented here is shown in Listing 3. This code takes the XDR decoded array from the client and

separates and averages the values. It returns the result which is then XDR encoded for transmission back to the client.

Listing 3. Server Code for Average Application

To complete the application at the client end, the input data must be packed into XDR format, so that it can be sent to the server. The client program is also generated manually and is shown in Listing 4. The Makefile shown in Listing 5 can be used to build the application.

Listing 4. Client Code for Average Application

Listing 5. Makefile

Testing and Debugging the Application

The best way to test the RPC application is to run both the client and the server (the caller and callee) on the the same machine. Assuming that you are in the directory where both the client and the server reside, start the server by entering the command:

```
avg_svc &
```

The **rpcinfo** utility can be used to verify that the server is running. Typing the command:

```
$ rpcinfo -p localhost
```

gives the following output:

```
program vers proto   port
 100000    2   tcp    111  portmapper
 100000    2   udp    111  portmapper
  22855    1   udp   1221
  22855    1   tcp   1223
```

Note that 22855 is the program number of our application from avg.x and 1 is shown as the version number. Since 22855 is not a registered RPC application, the rightmost column is blank. If we add the following line to the /etc/rpc file:

```
avg        22855
```

rpcinfo then gives the following output:

```
program vers proto   port
 100000    2   tcp    111  portmapper
 100000    2   udp    111  portmapper
  22855    1   udp   1221  avg
  22855    1   tcp   1223  avg
```

To test the application, use the command:

```
$ ravg localhost $RANDOM $RANDOM $RANDOM
```

and the following values are returned:

```
value   = 9.196000e+03
value   = 2.871200e+04
value   = 3.198900e+04
average = 2.329900e+04
```

Since the first argument to the command is the DNS name for the host running the server, localhost is used. If you have access to a remote host that allows RPC connections (ask the system administrator before you try), the server can be uploaded and run on the remote host, and the client can be run as before, replacing localhost with the DNS name or IP address of the host. If your remote host doesn't allow RPC connections, you may be able to run your client from there, replacing localhost with the DNS name or IP address of your local system.

A Brief Look at DCE RPC

The ONC implementation of RPC is not the only one available. The Open Software Foundation has developed a suite of tools called the Distributed Computing Environment (DCE) which enables programmers to develop distributed applications. One of these tools is DCE RPC which forms the basis for all of the other services that DCE provides. Its operation is quite similar to ONC RPC in that it uses components that closely parallel those of ONC RPC.

Application interfaces are defined through an Interface Definition Language (IDL) which is similar to the language used by ONC RPC to define XDR filters. Network Data Representation (NDR) is used to provide hardware independent data representation. Instead of using programmer-defined integer program numbers to identify servers as does ONC RPC, DCE RPC uses a character string called a universal unique identifier (UUID) generated by a program called **uuidgen**. A program called **rpcd** (the RPC daemon) takes the place of portmap. An IDL compiler can be used to generate C headers and client/server stubs in a manner similar to rpcgen.

Although the entire DCE suite is commercially sold and licensed, the RPC component (which is the basis for all the other services) is available as freeware. See the references section for more information on DCE RPC.

Further Study

The sample application presented here is certainly a naive one, but it serves well in presenting the basic principles of RPCs. A more interesting set of applications can be found in the Network Information System (NIS) package for Linux (see the references section). Also, the Linux kernel sources contain an implementation of Sun's Network File System (NFS), an excellent example of the use of RPC applied to the problem of distributed file access.

In addition to distributed data access, RPC can also be used to harness the unused processing power present on most networks. The book *Power Programming with RPC*, listed in the references section, presents an image processing application that uses RPC to distribute CPU intensive tasks over multiple processors. With RPC, you have the capability to boost the performance of your applications without spending a dime on additional hardware.

References

Ed Petron is a computer consultant interested in heterogeneous computing. He holds a Bachelor of Music in keyboard performance (piano, harpsichord and organ) from Indiana University and a Bachelor of Science in computer science from Chapman College. His home page, The Technical and Network Computing Home Page at http://www.leba.net/~epetron, is dedicated to Linux, The X Window System, heterogeneous computing and free software. Ed can be reached via e-mail at epetron@wilbur.leba.net.

xmotd: Writing Free Software

Luis A. Fernandes

Issue #42, October 1997

One programmer's experience developing freely available software for Linux.

> Lucy: “What happens if you practice the piano for 20 years and then end up not being rich and famous?”
>
> Schroeder: “The joy is in the playing.”
>
> —Peanuts

xmotd is a message-of-the-day browser that runs under the X Window System. It was written to ease the burden of the local systems administrator in making day-to-day announcements of general interest to students and faculty using the computer network in the Electrical and Computer Engineering Department at Ryerson Polytechnic University, Toronto, Canada. This is a chronicle of my experiences developing a program that started as a trivial, single purpose tool and evolved into one with numerous features suggested by its users.

Development

It was a late August afternoon in 1993 when Nick Colonello, the department's system administrator, and I were discussing the past school year and how it would have been nice to be able to let students logging in to the system know important information. With the following school year less than a month away, I boasted that I could write a simple message browser in about 10 minutes that could be run automatically by **xdm** (the X display manager) when students logged in. He agreed that it was a good idea and I proceeded to complete version 0.1 in under 30 minutes; it was called **xbanner**.(1) Visually, it consisted of a text widget where the message would be displayed, and a “dismiss” button, in the top-left corner of the window, used to close the window. For a brief moment, it looked supremely functional. But I was not satisfied—it looked utilitarian and ugly. (See Figure 1.)

Figure 1. xmotd 0.1 Screen Dump

In another half hour, version 0.2 was completed with an aesthetic re-fit consisting of a label widget that displayed a title, a separator widget (purely decorative) to separate the title from the text and another label widget to display a bitmap (a logo of our computer network). This was the version that was first installed on our system.

A few weeks later, Luis Lopes, the systems administrator in the Computer Science department dropped by for a visit and expressed interest in installing xbanner on his system. I was duly impressed that someone I knew would actually find something I wrote useful.

In March of the following year I read a post in comp.windows.x asking how a message of the day could be displayed when running X and xdm. Not knowing where it would be taking me, I innocently followed up saying that I had hacked together such a program and was willing to share it. Within a few hours I received two e-mail messages, and the next morning had another asking for a copy of xbanner. Now I was truly impressed.

Its popularity was never anticipated. It was as if no one had thought the world needed such a program. Indeed, a cursory investigation, prior to beginning work on xbanner, would have revealed the existence of a similar tool that, with minor modifications, could easily fit whatever new specifications were demanded. However, at this time the Web was in its infancy and **archie** was the *de facto*, parochially-minded, search tool of choice. Obscurity and ignorance condemned useful software to languish on ftp sites around the world.

Re-inventing the Wheel

I first learned of **xmessage**'s existence when Vivek Khera e-mailed me a shell-script that used xmessage, in conjunction with other Unix programs, as a message-of-the-day browser. **xmessage**, written by Stephen Glidea of the eX Consortium (2), is a generic tool that can be used to browse a text file or display a message. It consists of a widget in which the text is displayed and an “okay” button in the bottom left corner of the window. (See Figure 2.)

Figure 2. Xmessage Screen Dump

Criteria for Useful Software

Notice the remarkable aesthetic similarities between xmotd 0.1 and xmessage. In a market with several similar, feature-rich tools, aesthetics are often the deciding factor for a user. In a recent interview, the late Seymour Cray noted the importance of aesthetics: “I've enjoyed the aesthetics part of building

computers because it's any extra little thing you add that is clearly your own personality being projected in the product."

In addition to aesthetics, flexibility also plays a major role in selection. Experience has shown that users have a preference for application software that allows them to solve their problems, rather than overpowers by solving the programmer's problems. Usually, a programmer writes a program to solve a particular problem, then gives it away with the hope that others will benefit from it. Typical end-users do not find much use for this kind of software because it was written "by programmers, for programmers". It performs a single job—nothing more, nothing less. Common shortcomings of this type of software include:

- idiosyncratic interfaces, with little thought put into design (the principle of least astonishment does not apply)
- poorly chosen default configurations giving the software an ugly appearance
- a lack of customization (an aspect that emphasizes the idiosyncrasies even further because the user cannot replace the settings with something more suitable)

Software that allows infinite customization (3) endears itself to users dedicated enough to read the documentation because it gives them complete power—the same sense of creative power the developer felt when designing the software.

For example, xmotd's customizable bitmap was a relatively minor coding burden (an additional 10 lines of code to read in and validate a bitmap file) that affords the end-user a personal touch to what would otherwise be very bland software. The EE department's logo was originally hard-coded as the default image in the internal version of xmotd (now suitably renamed). When I uploaded xmotd to the eX Consortium's archive, I hard-coded in the X logo. One of the first requests I received was for the ability to change the logo. A less-than -optimal (and therefore temporary) solution I implemented was to amend the README with instructions on modifying the source code so a different bitmap could be compiled in. However, I found that people were very reluctant to modify code; they preferred something simpler. It then occurred to me to use the X logo as the default image and add a **-bitmap** option to allow the user to override the logo with one of his choice.

Evolution

> "Every program in development at MIT expands until it can read mail." --Unknown

Users invariably suggest enhancements, because the developer cannot anticipate all the uses for the software. A good rule of thumb is that for each person who asks for an enhancement, assume ten do not. My decision to implement the suggested enhancements was based on coding time versus how many users I thought would benefit from the feature. For example, I would guess that the **-usedomainnames** (4) option is rarely used other than at the site of the user who suggested it. I added it only because the overhead was one extra line of code in the function that generated the timestamp.

Another example: the **-wakeup** option initially specified an hourly period using an integer argument. When someone requested finer granularity (15 minutes sleep time) the argument was changed to a floating-point number with the fractional portion used to represent minutes.

The change was simple and didn't break backwards compatibility with **-wakeup** operation in the previous version. Users tend to be frustrated (to the point of not bothering to upgrade) when successive versions introduce incompatibilities with previous versions.

If a particular feature is implemented, it is wise to add an option to disable that feature because it is very likely that someone else may not like it. I added **-timestamp** to allow the default timestamp name ".xmotd" to be overridden after a user request. Put simply—provide lots of options.

Feedback

There is a certain amount of personal gratification in writing software, but developers are happier when people actually *use* the software. Many features now incorporated into xmotd were suggested by users (systems administrators) at other sites. Unsolicited e-mail from users is a good indicator of popularity and is very inspiring to developers. If you enjoyed using some software (regardless of whether it's free) and know the author's address, I would urge you to write him a brief note. The euphoria from reading a note of thanks is ineffable. As a benchmark, I have received 107 e-mail messages regarding xmotd, versus 10 messages for **xabacus** and **xsecure** combined (a pair of X applications I had authored prior to xmotd).

The e-mail I have received about xmotd can be categorized as follows (ordered from least encouraging to most encouraging; too many from the beginning of the list may make you want to give up writing free software):

1. Your program didn't compile. Do you know why?
2. I didn't read the README file, how do I install your software?
3. I'm too lazy to read the documentation, how do I use this feature?

4. The documentation for using this feature is unclear.
5. Can you add this feature?
6. Your program doesn't work.
7. Thanks for writing this cool software.
8. The documentation for using this feature is unclear, here is a fix.
9. There's a bug in your cool software, and here's the patch.
10. Here's some code that implements a new feature and makes your software even cooler.

Motivations

Wasteful duplication is common in the free software world. Given that Unix has been around for more than a quarter century, it is very likely that a solution to a problem has already been developed and you need to find it. On the other hand, commercial software has a colossal advertising budget and therefore mediocre software is often guaranteed commercial success. The ubiquitous and global nature of the Web is altering the balance. (When digital cash becomes a common transaction medium, it will become easy to charge for software.)

The Web allows software to have its own home page, providing inexpensive, world-wide advertising that precludes a marketing department. (5) New versions can be announced with varying amounts of fanfare. In some cases, major undertakings like the GIMP paint program or the Gnus newsreader for Emacs evolve around a mailing list subscribed to by a dedicated following interested in discussing and implementing enhancements. A catalog of free software would be a great benefit to both the developers and the users of free software. [The Linux Software Map on Sunsite is easily searched and functions as a catalog; however, it is not always up to date—Editor]

The rewards for commercial software are money and, often, fame. The rewards for writing famous, free software are recognition, a reputation and, at best, fanatic adoration. Many developers of free software share the sentiment illustrated in the opening epigraph. We write software because it's fun; it's more fun when people actually use it. Making it freely available guarantees that the greatest number of people will use it. Very often, free tools are the only alternative for those administrators who administer vast networks with minimal resources.

Conclusions

> “A work of art is never complete; the artist merely abandons it.” --Unknown

Three years (and 14 releases) later, I thought xmotd was approaching completion; i.e., every option that could possibly have been added had been. Since December 1996, however, more suggestions (which will be incorporated into version 1.15 of xmotd) were submitted, including one for running the HTML version within an Intranet and having it spawn a web browser when an URL was clicked. I should note that xmotd gained popularity when it began supporting HTML. (See Figure 3.)

Figure 3. HTML Version of xmotd

My advice to eager developers is, first, not to begin writing software without ascertaining whether a similar package exists (web space can be effortlessly searched these days); if it exists, consider extending it rather than starting from scratch. Second, write the software with full intent of incorporating user suggestions (don't ignore your users) and bug fixes following the first release (version 1.0 is never the last release). Finally, write software that keeps pace with advances in technology (e.g., Java) or face obsolescence. People will begin to use software only if it satisfies their needs and will continue using it only if it evolves to accommodate their changing needs. Persevere until the day you write software that receives the world-wide acclaim we all dream about.

Acknowledgements

Per Abrahamsen, Derek Fedak and Lars Magne Ingebrigtsen shared insights that helped shape an early draft of this article. The author's photograph comes to you courtesy of Jason Naughton whose perseverance got the Sun video camera working. The *Peanuts* quote is by Charles Schulz, copyright United Feature Syndicate.

1. *The significance of the short coding time is that I use a template file, which contains a skeleton X program including the necessary include statements, application defaults, accelerator and X resource structures, a main() function with XtAppInitialize() calls and more.*
2. *eX refers to the defunct X Consortium (disbanded in December 1996).*
3. *The Emacs editor embodies this principle to perfection: every character on the keyboard can be re-mapped.*
4. ***-usedomainnames** may be used at sites where home directories are shared (NFS mounted) across various domains and the message of the day applies across the whole network.*
5. *Insanely popular software like the Emacs editor, GNU/Linux, the GNU C compiler, Perl and the X Window System does not need a home-page—its popularity makes it ubiquitous. It will be archived on any major ftp site.*

Resources

Luis Fernandes (elf@ee.ryerson.ca) is a hacker. Winning the honourable mention in Sun's Alpha Java Programming Contest was an indescribable moment of his life. His definition of fun includes hacking X and Java; listening to music (composed before 1800 and played on period instruments, in particular); reading unusually good books and generally doing anything creative.

Archive Index Issue Table of Contents

Advanced search

Portability and Power with the F Programming Language

Walt Brainerd

David Epstein

Dick Hendrickson

Issue #42, October 1997

This article describes some of the design goals of F, a new programming language, and introduces most of the language specifics.

With the F programming language, the authors combine over forty years of language-design committee experience to create the world's most portable, yet efficient, powerful, yet simple programming language. The recent attention commanded by the portability and power of Java is well-timed, as we show in F that efficiency and readability need not be made victims of cross-platform development.

Before diving into the F programming language definition, this article begins with some biased-but-almost-factual opinions of the authors. We admit it—we are not fans of C and C++.

Some Driving Opinions

Listing a few facts and myths about programming languages will help set the stage for the discussion of F. These opinions may communicate some of the ideas behind the F programming language design, allowing one to better understand the motivations of the authors and thus the language.

Fact 1: Programs are read more often than written. From your first programming assignment throughout your professional career, characters are entered *once*, following some sort of syntax and logic, and read and reread anywhere from twice to hundreds to thousands of times. Programs that cannot be read are simply poor programs.

Myth 1: Abbrev.R++ abbreviations are good. A programming language with the overall design of abbrev.R++ are quite popular among thinking/creating/coding/debugging speedsters. Afterall, most programmers learned how to program before learning how to type. Abbreviations, however, ranging from a "}" instead of the word "end", "int" instead of "integer" and **i++** or **++i** instead of **i=i+1** only add pieces to an already complicated puzzle. As with a piece of abstract art, one day someone may look at your code and ask, "That's nice, but what is it?"

Fact 2: Educational languages are dead or dying. As some instructors around the world are searching for a suitable replacement for Pascal, the majority are going-with-the-professional-flow and switching from Pascal to C, C++ or Java for introductory programming courses. There is no telling where computer science would be today if a whole generation of programmers who were brought up on Pascal in the '70s and '80s were presented with the sink-or-swim situation of C++ or Java as a beginning programming language. If Pascal did not exist, the odds are that there would be fewer of us reading this article (if it or the magazine even existed). Surely, a major factor in the rapid evolution of computer science was the once nurturing environment presented by Pascal.

Myth 2: A modern educational replacement for Pascal offers no advantages to the potential professional programmer. Many professionals, particularly those working on large projects, benefit from the advantages of the strict style enforcement that a small programming language offers. A small language can also offer reliable tools (compilers, debuggers, profilers), reliable customer support, reliable error messages and reliable references (textbooks and on-line documentation). As F is a language based on existing practice, professionals can make use of the large amount of existing debugged code.

Fact 3: Choosing the wrong implementation programming language affects the overall design, portability and maintainability of large projects. Many companies have been dealt an expensive blow attempting to keep up with a fast-moving multi-platformed industry with slow-moving software. Whether the software is being enhanced with efficiency and new features or being ported to the latest hardware, a poor choice for the original programming language can result in a serious loss of company resources. Until feeling the headache, it appeared that C was an appropriate, powerful and portable choice. In the early '90s, C++ promised more power and possibly safer features. Today, Java proves safer and portable, but sacrifices efficiency.

Myth 3: The software crisis has been solved. With no solution to the software crisis in sight, focus has been shifted towards "market-driven" distractions like the *hot,new programming language* filled with more promises than an election-year politician. Meanwhile, most large software projects are still written in C and continue to be delivered late, under-functioned or unstable. As long as a

smaller and simpler language does not sacrifice power, it is time for programmers and their management to wake up to the possibility of shipping stable, complete software on schedule. This starts with the decision of an appropriate programming language. An appropriate choice does not emphasize the potential salary of the programmer leading the project, but rather:

1. **C**: Do we need to access system information, trading off portability?
2. **C++**: Do we need objects and run time binding as well as accessing system information, trading off readability and portability?
3. **Java**: Do we need objects, run time binding and portability, trading off efficiency?
4. **F**: Do we need portability, efficiency and maintainability trading off access to system information (unless calling C from F) and run time binding? **Fact 4:** Most statements in most programming languages fit on one line. In the average program, a minority of the statements are split across many lines. Requiring a semicolon at the end of every statement means requiring a semicolon at the end of almost every line. **Myth 4:** Semicolons are a fact of life. Given that the end of a line is most often the end of a statement, the trivial programming language design decision is to use a special character in the rarer case of needing more than one line for a statement. Requiring a semicolon at the end of a statement is tedious and error prone. Languages requiring a semicolon ought to be required to present a nice error message when the semicolon is forgotten. In F, the end of line is the end of statement. If a statement requires more than one line, an ampersand (“&”) is used at the end of a line.

The F Programming Language

Starting with an internationally standardized programming language as a base, we set out to create the world's best programming language. Any lesser goal would result in an interesting but not a challenging exercise.

A Language Design

Designing a programming language involves thousands of ideas and decisions. Tradeoffs are constantly weighed between efficiency (both compile time and run time), readability, flexibility, familiarity, brevity, redundancy, implementation (compilers and tools), style, elegance, completeness, internationalization, standardization, marketability and target audience, to name just a few. The above facts and myths and the principles listed below

helped us to avoid personality conflicts (mostly) and reach decisions based on these goals:

- Readability
- Learnability without loss of professional power
- Portability and maintenance of large programs
- Minimizing unimportant syntax
- Requiring words instead of relying on defaults
- Eliminating redundancy

A pleasant surprise to the biased authors is the pure elegance of F.

F Statements

Except for assignment (=) and pointer assignment (=>), the first word of every F statement identifies the statement. All keywords are reserved words, allowing for specific error messages for incorrect syntax or misspelled keywords. Table 1 categorizes all the F statements. The diagram shows that every F procedure, either a subroutine or a function, is contained in a module.

Functions Are Not Subroutines

In F, a distinction is made between functions and subroutines. Functions are not allowed to have "side effects" such as modifying global data. All function arguments must be intent(in); subroutine arguments can be intent(in), intent(out) or intent(inout). The intent is required on all procedure arguments, allowing the compiler to check for misuse and forcing both the beginner and professional to document the intentions.

Intrinsic and User Defined Types

The intrinsic types in F are integer, real, complex, character and logical. User-defined types can be constructed from the intrinsic types and user-defined types. For example, a *person* can be constructed to have a name, height, phone number and pointer to the next person. Users can define operators which operate on intrinsic and user-defined types.

Entity Attributes

The attributes of an intrinsic or user-defined type in F are shown in Table 2. Pointers are strongly typed. That is, pointers can point only to objects that are targets. Although this idea makes solid pedagogical sense, the words pointer and target originated for the purpose of better compiler optimization.

Array Language

A sophisticated array language facilitates operations on whole arrays, contiguous and noncontiguous sections and slices of arrays. For example:

```
arr(5:1:-2, 3, 6:)
```

is a reference to the two-dimensional array created by taking the elements 5, 3 and 1 in the first dimension of **arr** and elements from 6 to the upper bound of the third dimension of **arr**, all in the third plan of the array. If **arr** is a 5 by 6 by 7 array, the referenced elements would be (5,3,6), (3,3,6), (1,3,6), (5,3,7), (3,3,7), (1,3,7).

A simpler example that calculates the sum inner product of a row and a column is shown here:

```
A(i,j) = sum(B(i,:)*C(:,j))g
```

sum is one of the more than one hundred intrinsic procedures found in F.

Modules

Modules are at the core of all F code. Modules are a data encapsulation mechanism that allows data to be grouped with the procedures that operate on that data. Modules can use other modules. As well, programs and procedures can use modules. Using a module makes the public entities of that module available. Examples of modules are found in Table 3.

One does not instantiate an instance of a module as one does with a class in C++ or Java. Instead, the concept of an object is best viewed as a module that defines a public, user-defined type together with the public procedures that operate on that type. The user of such a module can then declare a scalar or array of the defined type and have access to its procedures.

A public, user-defined type can be defined to have private components, so that the type and its procedures can be referenced; however, the parts that make up the type are private to the defining module.

Module Oriented Programming

Programming in F can be called module-oriented programming. Much like Java's requirement that all procedures appear in classes, all F procedures appear in modules. An F program that does not use any modules cannot call any subroutines or reference any functions. Modules can use other modules to access their public entities. A module, however, is not allowed to use another module for the purpose of exporting the public entities in the used module

unless the sole purpose is to collect a group of modules and make all their public entities available from one module.

This simple yet powerful method of module inheritance allows for an involved hierarchy of modules without complicating the investigation required to understand somebody else's code. Any reference to the function **foo** is known at compile time to be specifically a reference to a public function named foo in a specific module. Even without the aid of compiler tools, F is designed so a quick search (with the aid of **grep**) for the words “function foo” will most likely show function foo's definition line on your screen.

A nice educational feature of F is that every procedure must be declared as either public or private. The result is that a student writing a program that calls a subroutine must learn (or at least enter) the words program, use, call, module, subroutine and public. The public and private list also aids the professional as the first occurrence of a procedure name in a module will tell you if it is private and thus isolated to this module.

Overloading Procedures and Operators

F allows overloading procedure names as well as overloading operators. Every reference, however, is resolved at compile time. Thus, the statement

```
left = swap(int1, real2) * "hello"
```

displays an overloaded multiplication operator operating on the result of the int1/real2 swap and the character string “hello”. Also, swap can be a generic name, but it is also resolved to a specific function at compile time. Finally, the assignment operator (=) may also be overloaded; a mouse click on the = could conceivably direct you to the specific subroutine that would be called if this was not an intrinsic assignment statement.

More About F … A Surprise?

Before reading this section, you may want to view the example F program found in Listing 1,1 the Sieve of Eratosthenes. to see if you can guess what once-popular programming language F is based on. The name of the base language is often deceiving as the little known 1995 standard of this language is far more modern than the popular 1977 version. As the standards team is working on making the 2000 version even more object oriented, compilers for the 1990 version have become available from most vendors only in the last few years. If you have not guessed yet, you may be surprised to find out that today's best structured programming language is based on the the world's first structured programming language—FORTRAN.

Listing 1. Sample F Program

Now over 40 years old, more programmer energy has gone into the evolving definition of FORTRAN than any other programming language. Every F program is a FORTRAN program. With stronger object-oriented features scheduled for the year 2000 and continued support for the numerically intensive programmer, this recently forgotten programming language is poised for a strong comeback during the next decade.

A strength of FORTRAN is that the standard is constantly being updated with new features. Vendors are relying on the standards efforts and announcing new compilers after the specifications have been accepted. This is a strong portability statement when compared to languages that are attempting to standardize after various compilers are already in the market. Another push for portability is being made with the addition of Part 3 of the FORTRAN standard regarding conditional compilation expected within a year.

Free For You

The Linux educational version of F is freely downloadable. The Imagine1 web page (http://www.imagine1.com/imagine1/) contains the free Linux version, and free trail versions for Windows, PowerPC Macintosh and Unix. You will also find the BNF for F, many example programs, descriptions of F textbooks, and an invitation to join the f-interest-group. As a point of reference, nonLinux users pay $101US for an F compiler and book.

Acknowledgments

Many thanks belong to Numerical Algorithms Group, Inc. (NAG) for helping to make the Linux version of F available at no cost. Making F available on Windows, Unix and Macintosh was made possible with the help of Fujitsu Limited, NAG, Absoft Corp. and Salford Software, Inc. Thanks also goes out to the FORTRAN community for providing immediate interest in the F programming language.

Walt Brainerd is co-author of about a dozen programming books. He has been involved in FORTRAN development and standardization for over 25 years and was Director of Technical Work for the FORTRAN 90 standard. walt@imagine1.com

David Epstein is the project editor of Part 3 of the FORTRAN standard regarding conditional compilation. He is the developer of the Expression Validation Test Suite (EVT) for FORTRAN compilers and author of Introduction to Programming with F. david@imagine1.com

Dick Hendrickson has worked on FORTRAN compiler development in both educational and industrial environments since 1963. He currently is a consultant on compiler optimization and one of the developers of SHAPE, a test suite for FORTRAN compilers. dick@imagine1.com

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Setting Up a SPARCstation

John Little

Issue #42, October 1997

A gentle introduction to Sun Workstations and installing Red Hat Linux SPARC 4.0.

Many PC users have taken the plunge into the Unix world with Linux and, I imagine, quite a few Unix die-hards like me have taken their first, faltering steps with PC hardware, using Linux, too. I have to say that for a long-time Sun user, the attempted transition to PCs was not without its trials. The mysterious workings of BIOS and extended-vs-expanded memory still elude me, so it was with more than a little relief that I noted, towards the end of last year, the release of Red Hat 4.0 for the SPARC architecture. Now we Sunnites can run our favourite OS on our favourite hardware, and the PC folks have a chance to run a familiar OS on what many may have, until recently, considered rather exotic hardware. Many institutions are selling off, or even scrapping, their first generation SPARCs as they become less and less useful when running the relatively “heavy” OS which Sun now ships. Many of these workhorses are becoming available to the individual user at little or no cost, and Linux provides an ideal solution to the question of which OS to use.

SPARCstations generally offer better performance than a PC of equivalent age and come with a relatively standard set of peripherals and interfaces. Among the benefits are a much higher screen resolution than the old VGA-based 386 machines (the standard resolution for Suns is 1152x900) and built-in audio, Ethernet and SCSI controllers.

In this article I hope to introduce those readers unfamiliar with Sun hardware to a few of the peculiarities of the breed, provide pointers to which systems are good buys and which should be avoided, and to provide enough information for you to get Red Hat's SPARC distribution up and running on your machine.

Shopping For SPARCs

What systems are out there and what should you look for when shopping for a Sun workstation on which to run Linux?

First of all, several groups of machines are not currently supported under Red Hat Linux/SPARC and will not work under the 4.0 or 4.1 releases.

- None of the VME-based deskside or server machines are supported (i.e., the 400 and 600 series).
- The SS1000 and SS2000 servers are not supported.
- The newer “Ultra” based machines are not supported.

I should emphasize here that SPARC Linux is one of the platforms where changes are happening very rapidly (kudos to the dedicated band of portmeisters), and I expect that by the time that you actually read this an Ultra version will be available. However, the Red Hat version 4.0 and 4.1 releases discussed here do not support that architecture.

More recent desktop machines, the Classic and SPARCstations 10 and 20 (all “4m” architecture), run SPARC Linux quite happily, and the testing and boot details given later in this article are generally valid for them. However, it's unlikely that the average home user will come across one of these machines on the market at a price which he'd be willing to pay for an unfamiliar piece of hardware, so I'd like to concentrate here on the machines which you're most likely to encounter, the older, “4c” architecture systems.

First, you need to be aware that several of Sun's chassis will accommodate several different types of CPU. A good example of this is the original “pizza box” SPARCstation enclosure (about the same size as a large size pizza delivery box, with a distinctive “dimple” pattern on the front and cooling holes in the same pattern on the sides). This chassis was virtually unchanged between several different models, the SPARCstation 1, 1+ and 2 (the SPARCstation 2 had a small fan and a floor grille added for disk cooling).

A Sheep In Wolf's Clothing

Unfortunately for potential buyers, this chassis also accommodates the Sun 3/80 CPU. The 3/80 is 68030 based (not SPARC) and, at the moment, no Linux port is available for it. Unless you're intent on joining the band of volunteers working on the Sun3 porting project, you do not want to buy a 3/80 in SPARCstation clothing. Don't accept that the logo on the front of the machine actually reflects what is inside. Take a good close look at the business end of the CPU. Even if you can't open the case to check the CPU chip, a dead give-away that the machine is actually a 3/80 is the presence of a 9-pin D-type serial

port on the CPU. None of the desktop SPARCstations has this type of connector, although most, including the 3/80, have a 15-pin D-type Ethernet connector, so count those pins.

Figure 1. Red Hat Release & SPARCstation 1+

Any of the other machines mentioned above, the SPARCstation 1, 1+ or 2, will run Linux quite happily. They generally come configured with a floppy drive and one or two internal hard disk drives. They don't have an on-board frame-buffer though and one of the three available SBus card slots must be sacrificed to add one. Note: on the 1 and 1+ machines, the third SBus slot is marked as a “slave” slot and is only suitable for frame-buffer boards. It will not support I/O cards such as SCSI or Ethernet.

Table 1

Two other models appearing on the second-user market more frequently nowadays are the SPARCstation SLC and ELC systems. Both of these machines are easily mistaken as being nothing more than ordinary monochrome monitors, as the CPU is built into the housing of a 17 inch grey scale monitor. The clues are the SCSI, RS232 and keyboard connectors on the back panel of the machine. Keep an eye open for these two, and you could pick up a bargain.

Although both models first saw the light of day after the original SPARCstation 1, they were explicitly targeted at the bottom end of the market and mostly ended up as diskless workstations. The ELC is the more powerful of the pair, though not by much, and it can be recognized by the CPU access panel at the top rear of the monitor housing. However, as with the previously mentioned machines, CPUs and chassis are interchangeable, so the only way to tell for sure which machine you have is to power it on.

Another two systems with a common chassis are the IPC and the IPX. The enclosure for these two is smaller in width and depth and a little taller than the “pizza box” machines, and it is designed to have the same desktop “footprint” as the (separate) monitor. This enclosure can be installed under the monitor or stood on edge using an “L” shaped, blue-grey plastic stand supplied by Sun along with the system.

The compact design of the enclosure emphasizes the fact that these systems were targeted at the desktop user. The chassis is limited to a single 3.5 inch hard drive bay (plus a floppy), and non-standard mini-DIN style sockets are used for the RS232 connectors on the CPU back-panel.

Both machines have a built-in frame-buffer. The on-board frame-buffer supplied with the IPC is only monochrome, although it is possible to add colour by utilizing one of the two available SBus slots. The IPX sports a colour frame-buffer and uses a different type of SIMM, making it possible for the IPX to have a greater overall memory capacity while actually having only one third of the number of SIMM sockets of the IPC. Of the two, the IPX is also the more powerful.

Which To Choose?

Of all of the “4c” architecture machines mentioned above, probably the best choice of all is the SPARCstation 2. It offers the versatility of three SBus slots and a two-disk chassis, and has a much more powerful CPU than it's two look-alikes, the 1 and 1+. One of the SBus slots is needed for a frame-buffer during the install stage—Red Hat Linux/SPARC 4.0 needs a bit-addressable console device for install. It can be removed later and one of the RS232 ports used for the console connection instead, freeing up the slot for another SCSI controller, for instance. The extra fan between the floppy and hard disk mountings means that this model runs the drives a little cooler. Although the IPX might be a more suitable choice if you're looking for a colour desktop machine, the SPARCstation2 is probably the ideal candidate for use as a low-cost, high-capacity server.

The SPARCstation 1 and 1+ really are the elderly spinsters of this older generation of machines. While both have mounting points for two 3.5 inch SCSI disks, most of the drives available on the market today require much greater cooling than this chassis can provide. It's probably best to avoid these two systems unless you plan to use only the Sun-installed internal drive and make all additions and expansions on the external SCSI bus. *Do NOT* install high speed, high capacity, hot running drives such as the Seagate Barracuda into either of these machines. The airflow around the drive bays is inadequate and you'll seriously reduce the reliability and lifetime of the drive.

While the SLC and ELC appear, on face value, to be the least attractive, they still have their good points. There's no cooling fan (or disk) in either, so operation is quiet. Many people still like monochrome systems for their crisp, steady displays, especially when using the machine for extended hours for simple tasks such as text processing. Both of these systems need an external hard drive to run Red Hat Linux/SPARC. As already noted, the ELC is the best choice of the two.

The IPC and IPX are more compact than the other models in the Sun4c range and both have the advantage of a built-in frame-buffer. However, with only one internal disk and two SBus slots, they are probably best suited as desktop

machines, rather than for server applications. A few people on the SPARC Linux mailing list have mentioned install difficulties with the IPC, but whether this is simply because it is such a common system or due to a specific system problem is an issue which remains unresolved at the time of writing. The IPX is the best choice between these two anyway, as it is the more powerful, has larger memory capacity and a colour frame-buffer as standard.

Getting there… Red Hat's SPARC Linux Release

There are two basic options available for obtaining the Red Hat Linux/SPARC distribution. For those of us without a direct connection to the Internet, buying the Red Hat CD-ROM set is probably the best choice, as the package includes a soft-cover manual and technical support via e-mail or fax. This being Linux, there are already some other CD-ROM collections available which include the Red Hat distribution (but without the hard copy manual or technical support options). On the other hand, if you have a fast Internet connection, you might want to download the distribution directly from the Red Hat ftp server or one of it's many mirrors. Because the Red Hat installation program has an ftp option, you can even limit your initial download to the install image and use that option to download only the package groups you specify during the interactive install process. The ftp option also has an additional benefit. The folks at Red Hat usually make their latest release available in the “devel” tree, so it's possible to download the “latest and greatest” version using this method.

No matter how you obtain the distribution, you must visit Red Hat's web site and pick up the very latest version of the errata file before you begin to install. This will save you some wasted time and frayed nerves.

Despite the general excellence of SPARC Linux, several problems with Red Hat's initial 4.0 release, which can spoil your whole day if you don't know about them, came to light only after the CD-ROM set was created. There are two files which you should look at, the Linux/SPARC errata and the general errata for whichever version you're loading. The latter covers problems common to all architectures. Table 2 contains the URLs for the list of ftp mirror sites carrying the release and for the errata.

Boot (And Booting Problems)

There are several methods for booting into the Red Hat installation program. Unfortunately, many of the worst bugs in the initial release were directly related to the boot process, so 4.0 could be quite a struggle to install, especially for anyone not familiar with the peculiarities of a SPARCstation to begin with. While the 4.1 release addressed many of these problems, there are still a few remaining bugs with the boot floppy images on the CD-ROM, and you'd be much better off ftp-ing the latest images from the Red Hat server.

Booting from the CD-ROM is the first and most obvious method. Note that while the official Red Hat CD-ROM is bootable, copies of the distribution on other vendors' “compilation” CD-ROM sets will usually not be. The drawback with a boot from the 4.0 CD-ROM is a bug preventing installation across multiple partitions, on the target disk. If you boot from the 4.0 CD-ROM you *must* install everything into one, large partition. If you attempt to install across multiple partitions, the install process fails shortly after creating the file systems on your hard disk. I need to emphasize here that this problem is restricted to booting from CD-ROM and 4.0 only. Any of the other methods of installation outlined below will circumvent this problem, as will booting from a 4.1 CD-ROM.

The command for booting from the CD-ROM is **bsd(0,6,0)** (from the **>** prompt) for those machines with a boot PROM revision of less than 2, or **boot cdrom** (from the **ok** prompt) for those machines with a boot PROM revision of 2 or greater. The single partition problem can be overcome by net booting your machine (see below) and then continuing with the install from the local CD-ROM.

Creating boot and root floppies is the obvious method for anyone who has downloaded the distribution across the Internet or who must boot from CD-ROM. Creating the floppies is simply a case of using **dd** on a Linux or Unix system, or using the supplied “rawrite.exe” program on a DOS machine. Booting from the floppy is either **bfd()** (PROM revision less than 2), or **boot floppy** (PROM revision 2 or greater).

Unfortunately, as mentioned earlier, the v0 floppy image from both the 4.0 and 4.1 CD-ROMs has major problems, and the boot process will almost always fail with messages which tend to make it appear as though the media itself were faulty:

```
ok boot floppy
Booting from: fd(0,0,0)
SILO
Read error on block 1
Read error on block 8
Fatal error: Unable to open filesystem
boot:
```

The third method for booting, and the one I'd recommend, is a network boot. This is your only option if your system has neither a floppy nor local CD-ROM drive, and it is an elegant work around for the problems besetting the CD-ROM and floppy installs. Assuming that you have access to a network and at least one other Linux or Unix system, this is by far the most reliable method. Note that what we're talking about here is only booting across the network, not setting up a diskless client. Creating a diskless client requires a server with a fair sized chunk of free disk space, and assumes that the server has the

horsepower and network bandwidth to support the client machine, whereas a boot server needs only enough disk space to hold the install image file and serve it to the client once at boot time. The client machine is not otherwise dependent on the server, and once the install program is running on the client it need never refer to the server again.

Setting Up A Boot Server.

The “images” directory on the CD-ROM and the ftp server contain both the floppy files (boot-v0.img and boot-v2.img) and a separate tftpboot.img file. As with the floppy images, the latest tftboot.img file can downloaded from one of the Red Hat mirror sites.

It is this latter file that is required for a network boot. The method used to configure the boot server is almost identical for Linux, SunOS and Solaris systems. We'll use an example configuration, where our boot server is a machine called **tyne.gaijin.co.jp** and the SPARCstation client is **coquet.gaijin.co.jp**. The server IP address is 172.17.172.50.

- Assign an IP address for your new machine on the same sub-net as the boot server. In our example, the next available IP address for the client “coquet” is 172.17.172.52. It's hardware Ethernet address (MAC) is 8:0:20:3:9:96.
- Create a directory for the tftp boot files on the server. SunOS and Solaris machines use /tftpboot by default, and we can use this directory on a Linux system, too.
- Copy the tftpboot.img file to the newly created directory.
- Create a symbolic link from the tftpboot.img file to a unique file name which the SPARCstation boot PROM requests across the net. The format of the symbolic link is <CLIENT_HEX_IP_ADDRESS>.<ARCHITECTURE>. Take the IP address which you assigned in the first step and convert it, section by section, to hex and then add the architecture of your system. In our example, we need to convert 172.17.172.52 into hex words and, since we aren't exactly sure what architecture our new system is, we'll create links for both 4c and 4m machines.

```
172     =       AC
17      =       11
172     =       AC
52      =       34
ln -s ./tftpboot.img AC11AC34.SUN4C
ln -s ./tftpboot.img AC11AC34.SUN4M
```

Note that unlike SunOS and Solaris, the same boot image can be used for both 4c and 4m architectures.

Figure 2. Connections to SPARCstation ELC

- Enable the tftp daemon in /etc/inetd.conf. The syntax for this entry is slightly different between Linux and SunOS/Solaris systems. On the Sun systems there's a **-s** option which enables the daemon in “secure” mode. The Linux tftp daemon does not use this option and, if it is present in the config file, it treats it as a directory name and all tftpd accesses fail. The other difference between Linux and SunOS/Solaris is that most recent versions of Linux come with the inetd.conf file configured with the tcpd logging daemon configured as default.

Linux:

```
tftpd dgram udp wait nobody /usr/sbin/tcpd\
        in.tftpd /tftpboot
```

SunOS:

```
tftpd dgram udp wait root /usr/etc/in.tftpd\
        in.tftpd -s /tftpboot
```

Solaris:

```
tftpd dgram udp wait root /usr/sbin/in.tftpd\
        in.tftpd -s /tftpboot
```

- Reinitialize **inetd** using a **kill -HUP <inetd PID>** or by rebooting the server.
- Ensure that the the client's Ethernet address is in the **arp** cache, and the **rarp** cache for Linux systems, on the server.

```
arp -s 172.17.172.52 08:00:20:03:09:96
rarp -s 172.17.172.52 08:00:20:03:09:96
```

Note the leading zero padding added to the Ethernet address in both cases.

The **rarp** command might produce this error:

```
cat: /proc/net/rarp: No such file or directory
```

It indicates that **rarp** isn't compiled into the kernel, and that the **rarp** module hasn't been loaded. Use **insmod** to load it and rerun the **rarp** command.

```
server# insmod /lib/modules/2.*.*/ipv4/rarp.o
```

If the module isn't present, you'll have to rebuild the kernel with **rarp** enabled.

The client system usually takes about three minutes to boot into the installation program. The screen changes from the default black-on-white to white-on-black with a much smaller font once the kernel loads. You'll start to see the normal Linux boot messages as devices are probed and identified. At the end of the

boot sequence the system drops straight into the install program, and from that point onwards the prompts are pretty much self explanatory. There are still a few things to watch out for, though.

Virtual Consoles

Red Hat install makes virtual consoles available to the user during the whole of the installation process. **<ALT>F1** gets you to the main installation screen, **<ALT>F2** is a shell, **<ALT>F3** displays informational messages from the installation program, **<ALT>F4** displays console messages and **<ALT>F5** displays messages from the individual package installation programs as they run.

Fdisk

Figure 3. Bare Connections to SPARCstation ELC

The SPARC-Linux installation requires that you create a “whole disk” partition (file system type “5”), spanning from cylinder 0 to the last available cylinder. Root, swap and /usr partitions are created in the normal way and overlap this “whole disk” partition.

As you can see, SPARC-Linux also numbers cylinders from 0, rather than the i386 Linux default of 1.

4.0 Specifics

If your initial install is from a 4.0 CD-ROM, there are a few things which you need to know (see Red Hat's errata list for up to date information). You really should update your kernel as soon as possible to circumvent a particularly nasty networking bug which plagued the 2.0.18 kernel shipped with this release. The problem causes random hangs and crashes, especially when the machine is attached to a network where IPX packets are also present. When updating the kernel on your system, be sure to update the kernel loadable modules, too. New kernel and module packages are available in Red Hat's RPM package format on their FTP server, or alternatively, you can get a binary “snapshot” of the latest kernel from vger.rutgers.edu (see Table 2).

Another problem which can be perplexing if you don't know about it beforehand is the dump program. This slipped into the distribution without having been checked for “endian-ness” and consequently behaves very strangely indeed, complaining about seeks to negatively numbered sectors and blocks. Again, an updated dump package is available from the Red Hat site. Version 0.3-5 or greater should work.

Wrap-up

As I write this, a group of folks are putting together the first Debian release of SPARC Linux. However, Red Hat is the only complete packaged release available on CD-ROM. While Red Hat's initial, 4.0 release suffered from a few teething problems, it has brought SPARC Linux into the mainstream Linux arena, rather than being confined to a backwater as something of an oddity. The worst of the problems have been fixed in 4.1, and this newer release also comes at a much more attractive price. The availability of both versions via FTP and on other vendors' compilation CD-ROMs is adding to the popularity of this architecture, and the installed base is spreading rapidly as evidenced by the increased traffic on the SPARC Linux mailing list. While I.T. professionals may still be reluctant to migrate the whole of their user base to SPARC Linux, there certainly seem to be a growing number of organizations out there who have one or two systems at least under evaluation. Many more old SPARC workhorses are getting a new lease on life with Linux and popping up on the Internet as FTP and Web servers.

John Little worked for Sun for nine years, is from the U.K., lives in Japan and works in Tokyo for an American company. He wears a range of increasingly bizarre hats in an (mostly futile) effort to hide his incipient baldness. He can be reached by e-mail at gaijin@pobox.com.

Archive Index Issue Table of Contents

Advanced search

LJ Interviews Thomas Roell

Marjorie Richardson

Issue #42, October 1997

An interview with the founder and president of XiGraphics.

Thomas Roell is one of those multi-titled, multi-talented kind of guys. He is the Co-Founder, President and Chief Technology Officer of Xi Graphics. Mr. Roell developed the original source code for X display servers on Intel PCs for the MIT X Consortium. I interviewed him by e-mail on June 18, 1997.

Marjorie: Let's start off with some personal information: country of birth, education, fun things, etc.

Thomas: I was born in Bavaria, Germany and lived most of my life in a small town there. There was nothing fancy in my formal education—I took the standard route to a degree in computer science. I have to admit that, back then,

I thought the activities outside of school were more important than the knowledge I got in the classroom. College and the emerging Internet certainly opened up an incredible playground. Remember, we're talking about 1989 where the uVAX II was thought to be state-of-the-art.

I do try to maintain a balance between work and fun. Moving to Colorado really transformed me into a sports-nerd. Colorado is actually an outdoor paradise, offering scuba diving, hiking, biking, paragliding, roller blading and skiing. It's said around the company that, after Christmas, I missed only one possible ski weekend on the slopes!

Marjorie: When and why did you move to the U.S.? How does Colorado compare to Bavaria?

Thomas: I moved to the U.S. in May of last year—quite a change for me. However, Colorado compares very well with Bavaria. There are lots of mountains around here with an incredible number of peaks waiting to be climbed.

Marjorie: I understand that your original X server code is the basis for XFree86. Can you tell us something about the transition from non-commercial to commercial product? Why you made the decision to go this route?

Thomas: This is a very interesting question. I made the decision to go commercial in 1992. The computer world back then was totally different from today. There was essentially no high-end graphics hardware available, except in a very few niche markets. Writing good software on PC-type hardware to compete with the workstations was always a challenge to me. The only way for me to finance development (which involved buying the board, for example) was to go commercial.

Also, remember that there was no Linux or FreeBSD movement, so the phenomenon that other people would contribute as we currently see with Linux was simply unheard of at that time. The transition itself was fairly easy—suddenly the limitations on how to get hardware were gone. The original X Server quickly moved forward during this time, both in technology and in driver support.

Marjorie: Did your X server code always support Linux? Why did you choose to port to Linux?

Thomas: Initially, Linux was not supported, since all of that predates Linux by quite some time. We had an experimental Linux port running in late 1994. Keep in mind that an X Server is, to a large extent, really independent of the

underlying operating system. To be honest, back then we did not expect the “Linux Phenomenon” at all.

Marjorie: Do you intend to stay with Linux as a platform of choice?

Thomas: Linux today addresses all the issues which the commercial Unix systems in the early 90s did not address. Systems such as SCO and UnixWare attempted to become players in the desktop arena, but were not too successful. Linux, however, definitively succeeded in the desktop arena and now is a viable alternative to the Microsoft-based systems. Our move to support CDE under Linux clearly shows that we do view Linux as one of the key cross-platform operating systems of today and tomorrow.

Marjorie: Tell us something to the origins of Xi Graphics. How you met Jeremy Chatfield and how you reached the decision to found a company together?

Thomas: I have known Jeremy for quite a while. He actually gave me my first job in the U.S., a short three-month stint during a summer break from college. Our paths crossed quite frequently after that, so he was the obvious choice when it came time to found this venture. He was living in Denver at the time, which was the reason the company was headquartered here. It turned out to be one of our best choices.

Marjorie: Why did you change your company name from X Inside to Xi Graphics? How has the name change affected business?

Thomas: I've been so involved in working on our new product release for this Fall, I haven't given much thought to the name change, which was designed to distance ourselves from X-rated Internet hits. But, I do know that as Xi Graphics we're certainly keeping busy!

Marjorie: Tell us about the design decisions for AcceleratedX, in particular, why you chose a design that features client, server and window manager as separate entities. Would you change it, if you were designing it today?

Thomas: Actually this is not a decision we made. This is inherently built into the X Windows system. The separation of clients, server and window-manager makes a lot of sense. In order to make them interact, you need to have very well-defined and understood interfaces. The X Windows system has a network protocol with an exact specification of the operations and semantics. If you compare this to Windows 95 where the interface is only loosely defined, if at all, you'll see a big advantage since you don't have to guess. It is true that this separation initially caused problems. The separation meant that there were many different window-managers available, and there were two major GUI

toolkits—OpenLook and Motif—out there. This competition, on the other hand, helped move the whole system forward in a short time. Those issues are now resolved and the Motif GUI style is the de facto standard.

Marjorie: Just how fast is AcceleratedX? How does it compare to XFree86?

Thomas: This actually comes down to the question of "what is speed?" We at Xi Graphics are always faced with a two-fold challenge. First, to make our product as stable as possible and comply with the X-Protocol specification, which is quite strict about what should and should not appear on the screen. Speed comes in second. If there is a trade-off between speed and correctness, we do sacrifice speed. This does not mean our drivers are slower. It just means they could be even faster if we would ignore some hardware problems of the graphics devices.

Now to the tricky point—how do you compare speed? Obviously, we want to have a fair benchmark that represents a real world scenario, rather than a meaningless number. With new and faster hardware, older benchmarks that gave meaningful results in 1989 are not appropriate anymore. The X Performance Characterization group (XPC) observed this dilemma a few years ago and came up with a new benchmark that exercises a lot of different operations in a real world scenario. This is one of the reasons why we are not especially focusing on measurements such as stones or engine, but rather those real world numbers which give real results.

The other key question is—how to compare performance if one X Server performs the correct way and the other one doesn't? For example, what if one X Server draws lines using software because the hardware does not comply with the X-Protocol specs, and the other X Server uses the hardware, but produces incorrect results? Putting all those issues aside, AcceleratedX usually is more than two times faster than competing products.

Marjorie: What improvements have been added in the latest release? Exactly what is "overlay technology"?

Thomas: We are constantly improving our products in many areas that traditionally do not appear on a feature list. In the latest release, we've focused on getting memory usage down, which is prompted by certain applications such as Netscape being down significantly. Issues like inter-activity have been addressed by making the X Server virtually multi-threaded. Color correctness, which is especially well supported within the X Windows system by the Xcms API, has been significantly enhanced. We now offer hardware gamma corrected visuals, along with the software that is required for programmers to take

advantage of this feature, so that one application displaying the same image on two different monitors will look identical.

The most important new feature, however, is the introduction of overlays or “multiple visuals” as I prefer to call it. Everyone using applications similar to Netscape will at some time experience those nasty pop-up messages stating that the application could not get enough colors and is not willing to run. On the other hand, if a configuration with more than 256 colors is selected, other applications won't run at all. With our new release we do address this problem by allowing both application classes to run at the same time. Basically, an application can choose whether it wants to run within a 256 color PseudoColor scenario or a 16 million TrueColor mode. Because most color-intensive applications will choose the 16 million color visual, all the “Technicolor” effects are gone as well. Everyone in our office started using this special mode (which only Xi Graphics has on Intel UNIX) after seeing it the first time.

Marjorie: Give us your thoughts on the evolution of hardware from “slow and expensive” to “fast and cheap”.

Thomas: It's the traditional story in the PC market. A couple of years ago there was no market for high-end graphics solutions, except maybe in the CAD sector. So, quite naturally, there were only a few highly specialized and expensive solutions. GUIs around 1992 were mostly text-based (if you want to call that GUI), so there was really no need for more and faster graphics hardware.

The turning point was really the original IBM 8514/A graphics accelerator. This was the first fixed-function graphics accelerator for the IBM PC. Fixed-function means here that there is not an extra general purpose CPU on a huge printed circuit board with a lot of additional logic. Almost all the hardware today has its roots in this design. The real revolution came in late 1992 when “S3”, a then-unknown company, created a single chip solution that combined the VGA standard with an 8514/A clone graphics accelerator. Similar developments then made the MS-Windows desktop system usable. This in turn started the whole rush of making graphics subsystems faster, and fostered the attitude, “if you can't beat your competition with performance, make it cheaper.”

When I look back to the first graphics system for which I wrote an X Server (which, of course, was top of the line then) and what is now available for about $300, there is an increased performance factor of 250.

Marjorie: What about your other product, CDE, Common Desktop Environment? Exactly, what kind of environment are you providing the user? Which platforms does it support?

Thomas: As outlined above, the X Windows system actually consists of a number of different components. CDE provides a robust standard for both programmers and end users that utilize the different components. The normal user sees a desktop manager and a number of essential productivity utilities—such as calendar manager, note pad, file manager and help system, to name a few. So it's way more than the fancy window manager to which it is compared.

Even the brief description I gave does not address the key potential, which is that CDE is completely network-oriented. The calendar manager, for example, can contact other machines and coordinate appointments with other people. Technologies such as ToolTalk and remote execution are integral parts of CDE, and CDE is not available just for Linux and FreeBSD. Basically, every major workstation vendor picked up CDE as the standard Unix desktop interface. It does not matter whether you use a Sun SPARCStation, an IBM RS/6000 or your Linux system at home, CDE will work for you. Much emphasis has been placed on the aspect of internationalization. CDE is the first Desktop Environment that allows application developers to develop truly internationalized applications. A single program is able to support Japanese as well as German or English. Of course, the desktop itself comes with support for a wide variety of out-of-the-box supported languages.

Marjorie: What does the future hold for Xi Graphics? for Linux? for yourself?

Thomas: We continue to focus on being the prime supplier of networked graphics solutions. This will not be limited just to the core X Server and the Desktop. We are positioning ourselves to provide a Desktop solution that is a valuable alternative in the Microsoft dominated PC market. Linux is certainly the operating system of choice to do that on a cross-platform level. As for myself, there are a lot of things I haven't tried yet. Currently, my team at Xi Graphics is trying to convince me that base jumping* is not the thing to try out next weekend.

Marjorie: Any thoughts you'd like to leave us with?

Thomas: Live long and prosper!

Mr. Roell seems to like danger. Base jumping is parachuting off buildings, mountains and other things with only a few hundred feet for the parachute to open fully.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

PostScript, The Forgotten Art of Programming

Hans de Vreught

Issue #42, October 1997

A tutorial for beginners is presented on writing PostScript files to display data.

The Alparon research group at Delft University of Technology aims to improve automated speech processing systems for information retrieval and information storing dialogues. The current focus is on dialogue management for a research project of Openbaar Vervoer Reisinformatie. The company provides information about Dutch public transport systems, ranging from local bus services to long -distance trains. They are capable of giving up-to-date travel advice from any address to any other address in the Netherlands. Last year they received over 12 million calls for information.

Since we use a corpus-based approach, we analyze tons of data. Due to the size of our data we do just about everything the Unix way: We use only stdin and stdout, and we run our scripts just as **sed** does (Those who can't program, write C/C++ programs; those who can, try to stick with scripts as long as possible. See also the White Paper in the References). Basically, we torture our data with Perl (and its little friends like **awk**, **sed**, **tr**, **grep**, **find**, et al.) until it is in a simple form, e.g., on each line you have an *x* and a *y* value.

Although we could import this in some fancy presentation program, we found that the generated PostScript files by these programs are often huge. That might be okay if you have just a couple of figures, but if you have a lot of them, you start to wonder if there is a better way. Of course there is; you can write the PostScript yourself, as I often do. In a Perl script I transform the *x-y* table into PostScript. Since LATeX requires a bounding box, I always make the PostScript level-1 compliant.

In this article I will give you a crash course in how to write level-1-compliant PostScript—enough instruction so that you can make your own simple figures. I will begin with the basic operators and then we can start drawing lines, filling shapes and drawing text. After that I will present a description of compliant

PostScript and an example. I will show you how to draw a histogram, because a histogram has all facets: lines, shapes and text.

PostScript Basics

Normally, when you wish to learn PostScript, you read the Blue Book (see References). If you just wish to know sufficient PostScript for most of your needs, keep on reading. PostScript is a Turing complete stack language. The Turing complete part (well, I am theoretical computer scientist) means that it is as powerful as any other programming language. The stack part means that all computations are carried out on a stack.

For instance, run Ghostscript (not Ghostview) by typing **gs**. The command **pstack**, the basic debugging tool, will show you the current stack. Enter **1 2 3 4 pstack** at the prompt and a new stack is displayed.

When you type the stack operator **pop**, 4 is popped off the stack. Next, type **exch**, and 2 and 3 will swap places. Another handy stack operator is **dup**, which duplicates the top element. The last important stack operator is **roll** which takes two arguments, say *n* and *j*. The command *n j* **roll** (with *n* and *j* replaced by numbers, of course) rotates the top *n* elements of the stack *j* times. So if the stack shows **1 3 2 2**, the command **4 1 roll** outputs **2 1 3 2**.

PostScript also has all the normal arithmetical operators, but since it is a stack language, you do your arithmetic in reverse Polish notation; i.e., the operators always follow the arguments. The standard arithmatical operators are **add**, **sub**, **mul**, **div**, **idiv** (integer division), and **mod**. PostScript also has geometric, logarithmic and exponential functions.

PostScript works best if you do everything on the stack, but in some cases this isn't particularly convenient. PostScript also has variables, but they are a bit slower than the stack. When you start writing your own PostScript programs, you will often try to do everything with variables—this is considered a *Bad Thing*. With some practice you will use fewer and fewer variables. To give a variable a value you type:

```
/PointsPerInch 72 def
```

which assigns **72** to the variable named **PointsPerInch**. If you use **PointsPerInch,** PostScript will replace it with 72.

In PostScript you can also define subroutines. Basically this is the same as assigning a variable, only in this case the value is a code chunk enclosed in curly braces. For example:

```
/Inch { PointsPerInch mul } def
```

PostScript also has flow control commands which are beyond the scope of this primer.

Action

Time to do something more interesting. To draw a line, enter:

```
newpath 100
400 moveto 300 200 lineto
        500 300 lineto stroke
```

This instruction starts a new path by moving to point (100, 400), drawing a line to point (300, 200), then to (500, 300) and, finally, painting the current path. Without the **stroke** you wouldn't see a thing. Instead of absolute positions you can also use relative movements with **rmoveto** and **rlineto**. The thickness of the lines can be controlled by the **setlinewidth** option, e.g., to draw a hair line:

```
0.01 setlinewidth
```

Filling shapes is also easy. Replace **stroke** by **closepath fill**; the command **closepath** connects the last point with the first point to form the shape and **fill** fills the shape with the current color or grayscale. In this case we get a black triangle. If you do **0.9 setgray**, the fill color will be a light gray (0 is black and 1 is white). You can also select colors with **sethsbcolor** or **setrgbcolor**, but these options are a bit more complicated. (For further information, see the Red Book in the References.)

Placing text takes some initial preparation: First select the proper font:

```
/Times-Roman findfont 10 scalefont setfont
```

selects a 10-point Times-Roman font. Other well-known fonts are Helvetica and Courier. Placing text is easy; you move to the position where you want the text, add parentheses to enclose the text and add the command **show**. In the case where the text contains parentheses or a backslash you must “escape” them by inserting a backslash before the character to be escaped. Sounds familiar, right? So the line:

```
400 400 moveto (Hello World) show
```

prints a greeting you have seen many times before.

There is one final command that is crucial if you want to see anything rolling out of the printer: **showpage**. This command transfers the picture, made by the PostScript interpreter of your printer, to paper and clears the memory

afterwards so that you can start a new page. The command **run** can be used to load your file:**(file.ps) run** executes the file named file.ps.

Compliant PostScript

Compliant PostScript is nothing more than PostScript with some special comments and layout instructions for your PostScript program, so that other programs can read your PostScript program and perform certain operations on it. There are numerous examples of available operations: reversing the pages, scaling the pages, rotating the pages, placing two or four pages on one page, etc. Even the program Ghostview uses compliant PostScript, the title and page numbers that you see are retrieved from the PostScript comments.

Since this is all done through comments, it does not matter to your printer if the PostScript is compliantor not—the printer skips all comments. However, in a Unix environment it is common to use the output of one program as the input for another. Thus, it is quite natural not to consider a PostScript file as an end station. In a Unix environment compliant PostScript files are a must; otherwise, the print filters will not be able to process the files.

In a Microsoft environment with Microsoft's well-known word processor, PostScript is an end station. The generated output suggests that it is level-2 compliant PostScript, but, alas, it is not. There are so few rules to make your document level-1 compliant (see Red Book) or even level- 2 compliant (see DSC in the References), that you might wonder how it ever wouldn't be.

I will stick to level-1 compliant (it is a bit easier as level-2 has more overhead). In the Red Book, level-1 compliant PostScript is described in just eight pages; so if you want to know all the ins and outs, it is the *right* source.

There are three types of comments:

- Header comments go before any PostScript code.
- Body comments primarily serve to mark the boundaries of pages.
- Trailer comments follow all PostScript code and are normally used to describe certain features that were not yet known from the header comments, so they were deferred (e.g., number of pages, fonts to be used and bounding box).

Normally, a comment in PostScript starts with a **%** (percent symbol), but these structuring comments start with **%%** (or **%!** if it is the very first line). A **%%** comment is directly followed by a keyword denoting its type of structuring comment, and if arguments are needed, they follow a **:**(colon) and are

separated by spaces. As an example, let's look at the following template (the number in front of each line is just for reference):

```
1  %!PS-Adobe-1.0
2  %%DocumentFonts:
3  %%Title:
4  %%Creator:
5  %%CreationDate:
6  %%For:
7  %%Pages:
8  %%BoundingBox:
9  %%EndComments
```

The first line suggests that these files are going to be level-1 compliant. One common misunderstanding is that people think the **1** in **1.0** denotes the compliance level, but that is not the case. Only level-1 and level-2 compliancy exist, so even if you see **%!PS-Adobe-3.2** , it is not level- 3 compliant (it should be level-2 compliant).

The second line contains the fonts to be used in this file. Some programs find it handy to know in the beginning which fonts they should load. However, when you create a program that generates PostScript, you often do not know this at this time. This header comment can be deferred to the trailer comments. In that case you will have to replace the ***font1 font2...*** part of the line in the header with **(atend)**.

The third line is easy; ***text*** indicates the title of the document. Often this is the file name, but it does not have to be. Spaces in ***text*** are no problem. The fourth line is equally easy; here ***text*** should be replaced by the author or the application that created this file.

In the fifth line ***text*** should be a date and time humans can interpret. Line 6 is optional, and ***text*** should be replaced by the intended recipient. If absent, the intended recipient is **Creator**.

In line 7 **number** should reflect the number of pages in the document. Since this number is often not known beforehand, it is frequently deferred to the trailer comments. Again, substitute **(atend)** in this case.

Line 8 contains four arguments: the *x* and *y* coordinates for the lower left corner and the upper right corner. In the case of multiple pages you should use the bounding box, so that all pages lie in the bounding box. To fill in the right values, you will find Ghostview very handy. The bounding box can also be deferred to the end; again you would specify **(atend)**.

Finally, line 9 ends the header section. Besides line 1 and 9 the order of the other lines can be chosen as you see fit.

After the header you normally see some PostScript definitions of variables and subroutines. These variables are intended to remain constant throughout the rest of the PostScript program.

Next, it is time for the body comments. The first body comment is **%%EndProlog**, which ends the “invariant” section of the program. Most print filters leave everything up to this line intact.

Each page is preceded by (again line numbers are only for reference purposes):

```
1  %%Page:
2  %%PageFonts:
elm
```

The first line contains ***label*** and ***ordinal***. ***label*** should be replaced by a string, containing no white space, that indicates which page it is. Sounds a bit weird, but this means that Roman numerals are okay, and if you have two pages on one, then **1,2** or **3,4** is a valid page number as well. In our case, it is just a plain number. The ***ordinal*** part indicates the value of the page number. Having two types of page numbers is just for your convenience.

The second line is optional and describes which fonts are to be used in the output. If absent, the fonts specified for **DocumentFonts** in the header are used.

Although it is not necessary, if you want to create just a single page, the next PostScript command is **save**, which makes a copy of the environment you have built so far. At the end of the page you will find the line:

```
restore showpage
```

This command retrieves the original situation and prints the page. It means that anything you do between save and restore is fair game, i.e., you can't screw up other pages if a filter reorders the pages.

After the final page you have **%%Trailer**. Most filters leave this line and everything that follows it intact. In some PostScript programs some cleaning up occurs here, but in most PostScript programs the trailer comments follow directly (again, line numbers are for reference purposes):

```
1  %%DocumentFonts:
2  %%Pages:
3  %%BoundingBox:
```

Of these three lines, only those that were deferred in the header comments should be included here in the trailer.

If you want to make your PostScript level-2 compliant, you need to read the DSC (see references).

Creating a Histogram

The first step is to torture your raw data until you get a simple table. In practice you use Perl and friends for this step. For the sake of demonstration I will use a tiny table:

```
1993 9.0 8.6
1994 5.7 7.8
1995 6.4 7.1
1996 7.5 6.1
1997 8.4 5.9
```

This table has an *x*, a *y* and a *z*. What I wish to draw is a light gray histogram for *x* and *y* and a dark gray one for *x* and *z*. Normally, you know the minimum and maximum values in your table, or you just use an awk one-liner to determine those values.

Next step is get started using a template of the Perl script, histogram.pl, that will generate the PostScript file. This template is shown in Listing 1.

One remark about the last bounding box line. This is the size of A4 (European standard page size); for letter size you need **0 0 612 792**. In a later stage we will change this line, so that the bounding box fits more tightly.

Run the script and save the output in histogram.ps. Start up Ghostview to view this file. Not much to see, right? Time to edit histogram.ps. It is easier to do a little experimentation with this file rather than making changes directly to the Perl file (especially in a later stage when you are actually processing your data). We are going to experiment with the axes; our changes are shown in Listing 2.

When you are pleased with the result, copy it into histogram.pl just after the **save** command and add the line **1 setlinewidth** to restore the original line width. Now it is time to do the hard work: defining two subroutines **Histo-y** and **Histo-z**. Again, this normally requires some experimentation, so create the PostScript file and edit it. We will assume that each subroutine gets *x,y* and *x,z* respectively on the stack. We will give both histograms a border line. It often helps to put a couple of your data points on the stack as an experiment.

You can copy your subroutines just in front of the **EndPrologue** line of your Perl script as shown in Listing 3.

Just a few words: I warned you to avoid using variables, and I did not practice what I preached. Well, only in the case of huge tables do you do everything on the stack. Doing so is much harder and often not worth the effort—my time is

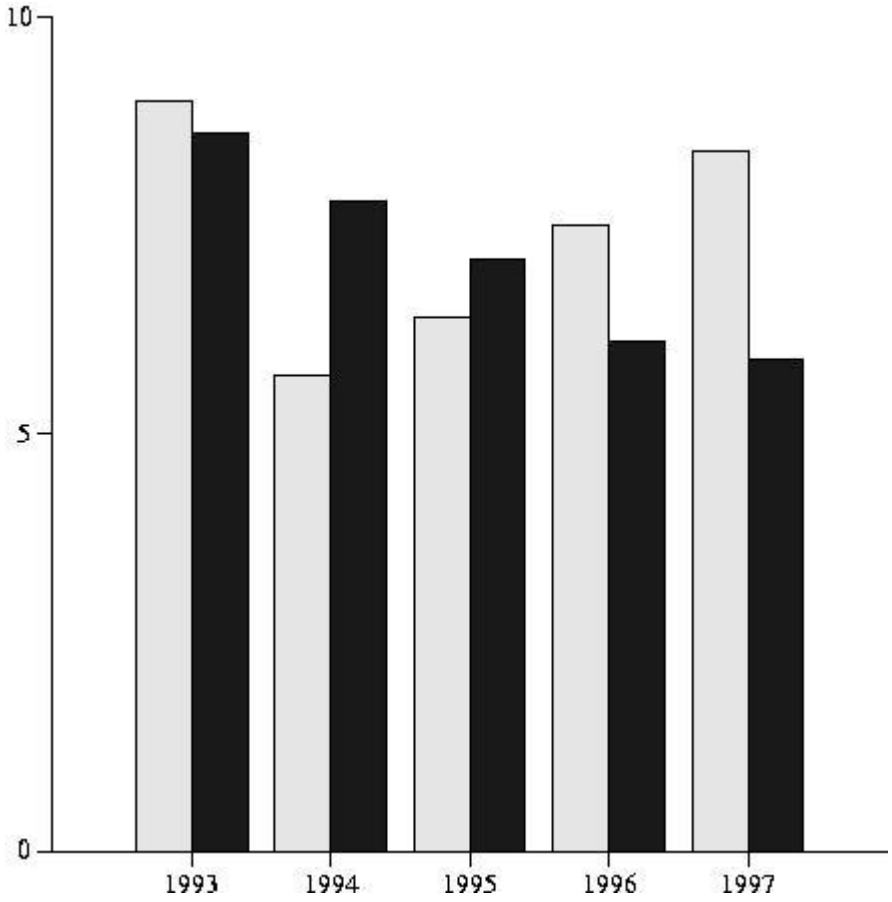
more expensive than what I gain in speed. Furthermore, I do some of the computations in PostScript; usually, it does pay off to do this in your Perl script. Finally, you normally do not want to recompute the path; you save it. I just wanted to keep the example simple.

Now it is time to complete your Perl script and process your data by adding the lines:

```
  while (<>) {
             chomp;
             ($x, $y, $z) = split;
             print "$x $y Histo-y $x $z Histo-z\n";
     }
```

Now you can run your script with the data as stdin to create a new histogram.ps. The final step is to determine a better bounding box. This is where Ghostview comes into play. Go to the leftmost and rightmost pixel of your picture and write down the *x* coordinates. Now do the same for the top and bottom of your picture, writing the **y** values. With these coordinates you can determine the bounding box (it does not have to be pixel fit) **83 85 400 405**, and you can change it in your PostScript file. (Or in your Perl script; however, if you have a huge data file to process, recreating the PostScript file can take a while.)

Now you have a fully level-1 compliant PostScript file less than 2KB is size that you can actually understand. I have seen PostScript files generated by applications under MS-DOS that need 2MB for the same picture. The complete Perl script and output PostScript are included in the gzipped tar file on the ftp site as Listings 4 and 5. The output histogram is shown in Figure 1.



Epilogue

So from now on we do everything in PostScript, right? Wrong. If it goes faster using another application and the generated PostScript file is not too large, use that application. For many pictures I still use **xfig** or something similar. Use PostScript directly if your data set is big and importing your data into the application already requires a lot of work. If you are relatively new to PostScript, concentrate on *x-y* graphics and histograms. If you have gained some experience, read the Blue and Red Books. Most importantly, have fun.

References

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

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Linux and the Alpha

David Mosberger

Issue #42, October 1997

This is the first of a 2 part series, an introduction to the Alpha family of computers in preparation for giving us the techniques for optimizing code on this high-performance platform in Part 2.

Ever since its announcement in the Fall of 1991, the Alpha architecture (see Reference 3) has been the foundation of the world's fastest systems. In fact, except for one or two brief blips, Alpha systems have been the highest-performing systems based on single-CPU SPECmark performance. With this outstanding performance record comes marketing hype and, sometimes, unrealistic expectations. It is not all that uncommon to find e-mail messages or USENET articles saying things like: “I heard the Alpha is so fast, but now I find that my dusty deck is just 10% faster on the Alpha than on the other system.” So what's the truth? The honest answer is that it depends on what you're doing. Alpha systems are without a doubt fast machines, but it is unreasonable to expect that taking a dusty deck and running it on an Alpha will result in the best possible performance. This is particularly true for programs that were written with the mind-set of the eighties, when CPU cycles were at a premium and memory bandwidth was abundant. Reality looks quite different today: CPU clock-rates above 150MHz are the rule and even laptops can run at 200MHz or more. The result is that, today, the memory system—and not the CPU—is often the first-order bottleneck.

In part 2 of this article, we will demonstrate a few simple techniques that help avoid the memory system bottleneck. Except for one case, the focus is on integer-intensive applications. The topic of optimizing floating-point intensive applications is certainly just as important but, unfortunately, well beyond the scope of this series. The techniques presented can result in tremendous performance improvements. While the techniques will be helpful for all modern systems, they normally extract the biggest benefits on Alpha-based machines. There are a couple of reasons for this bias.

One, the Alpha architecture has been designed with longevity in mind. Specifically, the Alpha architecture should be good for the next 15-25 years, which corresponds roughly to a 1000-fold increase in overall performance. For this reason, some design-tradeoffs were made in favor of long-term viability rather than short-term benefits. For example, the Alpha was right from the start a 64-bit architecture, even though, at the time of its announcement, 32-bit address spaces were considered comfortably large.

Two, the current Alpha implementations are designed to achieve high performance by pushing clock frequency to the limit. This means the CPU-to-memory-system performance gap is the largest for Alpha-based systems. For example, suppose a memory access takes 100ns. On a 500MHz Alpha CPU, this corresponds to 50 clock cycles. In contrast, on a 250MHz CPU, this is only 25 cycles. So the relative performance penalty of accessing memory is much higher on a CPU with higher clock speeds. This may sound like a bad thing, but since the absolute performance is the same, what this really means is that a fast-clock CPU system that is running a memory-bound application will be about as fast as a slower-clock system, but when running a memory-wise application, it will be much faster.

In this part of the series, I present a brief overview of existing and upcoming Alpha implementations. While it is not usually necessary to optimize for a specific CPU, it is helpful to know what the characteristics of current CPUs and systems are. I also discuss a couple of simple performance analysis tools that are available under Linux. When porting legacy code to modern systems, such tools are invaluable, since they avoid wasting time trying to optimize rarely executed code.

Overview of Alpha Family

So far, the Alpha CPU family tree spans three generations; it all began with the 21064 chip. At the time of its introduction, it was the highest performing CPU, and it still makes for a nice workstation, though it's no longer competitive with the latest generation CPUs. This chip branched off into a version that was called the “Low-Cost Alpha” (LCA), also known as 21066 or 21068. The chip core was identical to the 21064 but it had an integrated memory and PCI-bus controller. This high integration made it possible to build Alpha-based systems at relatively low cost and for the embedded systems market. Unfortunately, the design had a major weakness—the memory system was seriously under-powered. This created the paradoxical situation in which a system based on this chip performed on some applications on average, no better than a 100MHz Pentium, but outperformed a P6 running at 200MHz. As a result, the reaction to this chip varied greatly, and probably resulted in quite a few disappointed customers for Digital. On the other hand, there is no doubt that the low-cost at

which 21066-based systems eventually were sold caused a quantum leap in the number of Linux/Alpha users.

Around June 1994, the 21164 chip was announced. It had dramatically improved performance over the 21064 and was the first, and so far only, Alpha CPU to feature a three-level cache hierarchy. The first and second-level caches were both on-chip and only the third-level cache was on the motherboard. This chip, in slightly improved versions, is still going strong. At the Fall 1996 Comdex in Las Vegas, such a chip, coupled with a liquid cooling system, was demonstrated running at 767MHz. Another version, called 21164PC, is scheduled to become available around Spring 1997. It omits the relatively expensive second-level, on-chip cache but adds multi-media extensions and other performance-enhancing features. As the name indicates, this chip is designed to be price-competitive with PC processors, specifically the forthcoming Intel Klamath (an improved P6). While price-competitive, the 21164PC is supposed to deliver over 50% better performance than the Klamath. For this second-generation, low-cost Alpha implementation, it certainly looks like Digital and its co-designer Mitsubishi are not going to repeat the mistakes of the past. The 21164PC promises to be cheap and fast.

If you happen to have a deep pocket or want to take a glance at what PC processors might look like in two or three years, the 21264 might be of interest. It is scheduled to become available in high-end machine during the second half of 1997. With this chip, CPU performance is expected to take another giant leap. Current estimates call for a performance level that is three to four times faster than the fastest CPUs available today.

Between each major chip generation, there are typically “half-generation” CPUs which have improvements that derive primarily from a shrink of the chip manufacturing process. For example, the 21064 chip was followed by the 21064A, and similarly, the 21164 was followed by the 21164A. In the former case, the core of the chip remained virtually identical to the 21064, but the primary caches doubled in size from 8KB to 16KB. In the latter case, instructions for byte and word accesses were added and the maximum clock frequency increased from 333 to 500MHz.

A summary of the performance attributes of the current Alpha chip family is presented in Table 1.

2394s1.ps (min=50, max=52)

Table 1. Summary of Alpha Chip Family

Linux Performance Analysis Tools

The state of Linux performance analysis tools is in rather dire straits (this is true for freeware in general, not just Linux). Commercial products currently have an edge in this area. For example, Digital Unix comes with an excellent tool (or rather tool generator) called ATOM (see Reference 2). ATOM is basically a tool that can rewrite any executable. While rewriting, it can add arbitrary instrumentation code to each function or basic block. Digital Unix comes with a bunch of tools built with ATOM: 3rd degree (a memory-leaks and bounds checker like the well-known purify) and a number of tools that give very detailed information on the performance behavior of a program (such as cache miss frequency, issue rates and so on). At present, the freeware community can only dream of such versatile tools.

While bleak, the situation is by no means hopeless. The few tools that are available make for powerful companions when properly used. Even good old GNU **gprof** has a few features of which you may not be aware (more on this later). Let's start with the most basic performance measurement—time.

Accurately Measuring Time

The Unix way of measuring time is by calling **gettimeofday()**. This returns the current real time at a resolution of typically one timer tick (about 1ms on the Alpha). The advantage of this function is that it's completely portable across all Linux platforms. The disadvantage is its relatively poor resolution (1ms corresponds to 500,000 CPU cycles on a 500MHz CPU) and, more severely, it involves a system call. A system call is relatively slow and has the tendency to mess up your memory system. For example, the cache gets loaded with kernel code so that when your program resumes execution, it sees many cache misses that it wouldn't see without the call to gettimeofday(). This is all right for measuring times on the order of seconds or minutes, but for finer-grained measurements, something better is needed.

Fortunately, most modern CPUs provide a register that is incremented either at the clock frequency of the CPU or an integer fraction thereof. The Alpha architecture provides the **rpcc** (read processor cycle count) instruction. It gives access to a 64-bit register that contains a 32-bit counter in the lower half of the register. This counter is incremented once every N clock cycles. All current chips use N = 1, so the register gets incremented at the full clock frequency. (There may be future Alpha processors where N > 1). The top half of the value returned by rpcc is operating system -dependent. Linux and Digital UNIX return a correction value that makes it easy to implement a cycle counter that runs only when the calling process is executing (i.e., this allows you to measure the process's virtual cycle count). With **gcc**, it's very easy to write **inline** functions that provide access to the cycle counters. For example:

```
static inline u_int realcc (void) {
    u_long cc;
    /* read the 64 bit process cycle
        counter into variable cc: */
    asm volatile("rpcc %0" : "=r"(cc)
                : : "memory");
    return cc; /* return the lower 32 bits */
    }
       static inline unsigned int virtcc (void) {
          u_long cc;
          asm volatile("rpcc %0" : "=r"(cc)
                        : : "memory");
          /* add process offset and count */
          return (cc + (cc<<32)) >> 32;
       }
```

With this code in place, function **realcc()** returns the 32-bit real-time cycle count, whereas function **virtcc()** returns the 32-bit virtual cycle count (which is like the real-time count except that counting stops when the process isn't running).

Calling these functions involves very small overhead. The slowdown is on the order of 1-2 cycles per call and adds only one or two instructions (which is less than the overhead for a function call). A good way of using these functions is to create an execution-time histogram. For example, the function below measures individual execution times of calls to **sqrt** (2.0) and prints the results to standard output (as usual, care must be taken to ensure that the compiler doesn't optimize away the actual computation). Printing the individual execution times makes it easy to create a histogram with a little post-processing.

```
void measure_sqrt (void) {
          u_int start, stop, time[10];
          int i;
          double x = 2.0;
          for (i = 0; i < 10; ++i) {
             start = realcc();
             sqrt(x);
             stop = realcc();
             time[i] = stop - start;
          }
         for (i = 0; i < 10; ++i)
             printf(" %u", time[i]);
             printf(""n");
       }
```

Note that the results are printed in a separate loop; this is important since **printf** is a rather big and complicated function that may even result in a system call or two. If printf were part of the main loop, the results would be much less reliable. A sample run of the above code might produce output like this:

```
120  101  101  101  101  101  101  101  101  101
```

Since this output was obtained on a 333MHz Alpha, 120 cycles corresponds to 36ns and 101 cycles corresponds to 30ns. The output shows nicely how the first call is quite a bit slower since the memory system (instruction cache in particular) is cold at that point. Since the square root function is small enough

to easily fit in the first-level instruction cache, all but the first calls execute at exactly the same time.

You may wonder why the above code uses realcc() instead of virtcc(). The reason for this is simple—we want to know the results that were affected by a context switch. By using realcc(), a call that suffers a context switch will be much slower than any of the other calls. This makes it easy to identify and discard such unwanted outlying statistics.

The cycle counter provides a very low-overhead method of measuring individual clock cycles. On the downside, it cannot measure very long intervals. On an Alpha chip running at 500MHz, a 32-bit cycle counter overflows after just eight and a half seconds. This is not normally a problem when making fine-grained measurements, but it is important to keep the limit in mind.

Performance Counters

The Alpha chips, like most other modern CPUs, provide a variety of performance counters. These allow measuring various event counts or rates, such as the number of cache misses, instruction issue rate, branch mispredicts or instruction frequency. Unfortunately, I am not aware of any Linux API that would provide access to these counters. This is particularly unfortunate since both the Pentium and the Pentium Pro chips provide similar counters. Digital UNIX gives access to these counters via the **uprofile** and **kprofile** programs, and an ioctl-based interface documented in the pfm(7) man page. Hopefully, something similar (but more general) will eventually become available for Linux. With the proper tools, these counters can provide a wealth of information.

GNU gprof

Most readers are probably familiar with the original gprof (see Reference 3). It's a handy tool to determine the primary performance bottlenecks at the function level. However, with the help of gcc, GNU gprof can also look inside a function. We illustrate this with a truly trivial function that computes the factorial. Assume we've typed up the factorial function and a simple test program in file **fact.c**. We can then compile that program like this (assuming GNU libc version 2.0 or later is installed):

```
gcc -g -O -a fact.c -lc
```

Invoking the resulting a.out binary once produces a gmon.out file that contains the execution counts for each basic block in the program. We can look at these counts by invoking gprof, specifying the **-l** and **--annotate** options. This

command generates a source-code listing that shows how many times a basic block in each line of source code has been executed.

Listing 1. Basic-block Execution Counts

Our factorial example results in the listing shown in Listing 1. The basic block starting at the printf line in function **main()** was executed once, so it has been annotated with a 1. For the factorial function, the function prologue and epilogue were executed 20 times each, so the first and last line of function **fact** are annotated with 20. Of these 20 calls, 19 resulted in a recursive call to **fact**, and the remaining call simply returned 1. Correspondingly, the **then** branch of the **if** statement has been annotated with 19, whereas the **else** statement has an annotation of 1. It's that simple.

There certainly are no surprises in the behavior of function **fact()**, but in realistic, more complicated functions or in code that was written by somebody else, this knowledge can be very helpful to avoid wasting time optimizing rarely executed code.

Optimization Techniques: Making Your Applications Fly

Next month, we'll look into the techniques that can greatly improve the performance of a given piece of code. Most of them are not novel. Some of them have been around for so long that it would be difficult, if not impossible, to give proper credit. Others are “obvious” (once you know them). The key point is that it is the characteristics of modern, particularly Alpha-based, systems which make these techniques so important and worthwhile.

Acknowledgments

The author would like to thank Richard Henderson of Texas A&M University and Erik Troan of Red Hat Software for reviewing this paper on short notice. Their feedback greatly improved its quality. Errors and omissions are the sole responsibility of the author.

David is a graduate student in the Ph.D. program of the Computer Science department at the University of Arizona. He plans on graduating in August 1997 and to finally get a “Real Job”. After a short intermezzo with Linux involving Reed-Solomon codes and the floppy-tape driver, he forgot about Linux until the need arose for a low-cost, high-performance system based on Digital's Alpha processor. As a result, he got involved in the Linux/Alpha port and has been sticking around in the free software community ever since. When not playing with computers, he enjoys the outdoors with his lovely wife. He can be reached via e-mail at David.Mosberger@acm.org.

SpellCaster DataCommute/BRI ISDN Adapter

Jay Painter

Issue #42, October 1997

This ISDN adapter was designed specifically for Linux, and the ISDN4Linux driver is shipped with the Linux 2.1.x development kernel.

- Manufacturer: SpellCaster Telecommunications Inc.
- E-mail: isdn@spellcast.com
- URL: http://www.spellcast.com
- Platforms: Linux 2.x, Windows NT 4.0, Windows 3.51 and 95
- Price: $359 US
- Reviewer: Jay Painter

For the last 4 months, I have used two SpellCaster DataCommute/BRI ISDN cards to connect SSC's satellite office to the main office. Previously, we had used high speed modems, and after that, a set of 3Com ISDN modems. The DataCommute/BRI has given us the fastest and most reliable connection. This ISDN adapter was designed specifically for Linux, and the ISDN4Linux driver is shipped with the Linux 2.1.x development kernel. See sidebar for hardware features.

Installation

The DataCommute/BRI is a 16bit ISA card. It is easy to install because there are no jumpers to set. The board's IRQ and shared memory are all set within the software driver.

Kernel Driver

SpellCaster has written an ISDN4Linux compliant driver for their ISDN cards. It is available for download from SpellCaster's ftp site: ftp.spellcast.com /pub/drivers/isdn4linux/scisa-2.00.tar.gz. This package compiles the kernel module, sc.o, and a configuration program, **scctrl**. The scctrl utility is used to set the ISDN switch type and SPID numbers, and to check the connection status, along with numerous other options. I had no problems compiling the driver and the scctrl utility. The kernel module takes several command line arguments to set the IRQ, memory base and io base. I recommend the use of the sc.o driver option **do_reset=1** when using **insmod** to install the driver module into the kernel. If the reset option is not used, the kernel module will fail to install after a soft reboot unless you remove the module before the reboot.

Configuration

The SpellCaster/BRI must first be configured to communicate with the phone company's ISDN switch. This action includes setting the SPID numbers of your two ISDN channels, setting the phone number (the ISDN board needs to know its phone numbers), the ISDN switch type and the speed of each channel. These hardware settings are set using the scctrl utility which is provided by SpellCaster and comes with the kernel driver. After hardware configuration is complete, the connection configuration is handled by the generic ISDN4Linux utilities. These utilities require “ISDN Subsystem” support compiled into the kernel. ISDN4Linux supports single channel and dual (128K) PPP connections, as well as HDLC connections. IPX can also be transmitted by using ISDN4Linux's emulated ethernet encapsulation.

Problems

It is important to use the 2.0.27 kernel with the SpellCaster boards. There have been problems reported with the ISDN subsystem in later kernels, and it hasn't been fixed as of 2.0.30. It is essential that the SpellCaster/BRI's firmware is at least version 1.51. If you have a card with an earlier firmware revision, multilink PPP connections will have less throughput than a single-channel connection due to a bug that causes packet dropping.

Performance

I found the SpellCaster's performance to be its greatest asset. Ping times between the two SpellCasters average 47ms, and long haul FTP transfers average 15KB/sec (122Kbps) with a multilink PPP connection. The CPU utilization is so low that at maximum transfer speed, the process of receiving from the ISDN and routing to Ethernet only used 1% of the processor time on a 486-33. The stability of the connection is also impressive. Our two ISDN routers

have maintained a connection now for over a month with no intervention, despite periodic outages caused by unreliable ISDN service from our phone company. Reconnecting after an outage is handled by the ISDN4Linux subsystem automatically.

Conclusions

The hardware and the software of the SpellCaster/BRI both work well and are easy to use. The ISDN4Linux subsystem is more difficult to configure, and I recommend getting the FAQ at http://www.lrz-muenchen.de/~ui161ab/www/isdn/ before spending a lot of time attempting to decipher the cryptic acronyms.

Jay Painter is the systems administrator for SSC. He can be reached via e-mail at info@linuxjournal.com.

Archive Index Issue Table of Contents

Advanced search

Internet Programming with Python

Dwight Johnson

Issue #42, October 1997

After reading only the first three chapters, the reader will begin to understand why Python is rapidly becoming the language of choice for many programmers.

- Authors: Aaron Watters, Guido van Rossum, James C. Ahlstrom
- Publisher: M&T Books
- URL: http://www.mandt.com/
- Price: $34.95
- ISBN: 1-55851-484-8
- Reviewer: Dwight Johnson

Internet Programming with Python is a book to teach Python programming to intermediate and advanced programmers.

Authors Watters, van Rossum and Ahlstrom are well qualified to write about their subject: van Rossum is the primary author of the Python language; Watters works for AT&T Laboratories on exploratory Internet applications; Ahlstrom is the original author of the WPY/Python GUI package. All three authors can be found actively contributing to the Python mailing list and the comp.lang.python newsgroup.

Chapter One briefly describes the strengths and weaknesses of Python. As Python is a dynamically typed, object oriented, general purpose language built from a small number of constructs, it is both powerful and easy to learn and use. Because it is interpreted, it is suitable for rapid prototyping and program development. Because it easily incorporates modules written in other languages, it is an excellent glue language for the overall structure of large programming projects. Python is not suitable for programming algorithms which require very rapid or time critical execution such as data compression, device drivers, complex floating point calculations or complex database operations.

Chapter Two gives a birds-eye view of Python and Chapter Three is a fun and humorous hands-on tutorial which gives the reader some experience working with actual Python constructs and code.

Chapters Four, Five and Six present the Python language in a more formal and systematic way. The authors recommend these chapters for later reference rather than for detailed reading. Although these chapters are quite complete, they are not intended to replace the reference manual that comes with the Python distribution.

To illustrate the virtues of programming in Python, Chapters Seven, Eight and Nine introduce three of the most abstruse applications of Internet programming—the dynamic generation of HTML documents, CGI programming, and programming Internet protocols—and show how Python easily and elegantly deals with the programming issues involved.

The authors are teachers at heart and take pains to briefly and lucidly explain each of these applications before delving into their Python solutions—solutions which illustrate and justify some of Python's most powerful features: multiple inheritance, dynamic attribute access, dynamic keyword arguments and use of the extensive Python tools, libraries, modules and demos that come with the Python distribution.

Python's object-oriented features make it a convenient language for implementing a graphical user interface. Chapter Ten introduces Tkinter and WPY, two class libraries developed for that purpose, and goes on to use WPY to develop a very interesting HTML viewer application that could very easily be enhanced to become a simple web browser.

Chapter Eleven is an introduction to writing C language extensions which can be used to add new basic functionality to Python. According to the authors, “Extending Python is easy, so easy in fact that playing with Python extensions might be an excellent way to learn the C programming language...” They illustrate extending Python with an example called BStream that has the practical application of conveniently manipulating large images.

Python can be embedded as a component of another main program. As an example, Chapter Twelve shows how to embed Python under any Netscape HTTP Server product that supports the NSAPI server component API. This might be useful to provide dynamic CGI server functionality to a web server without the overhead of running separate CGI processes for each request.

Appendices give a brief guide to the Python standard libraries and a very useful tutorial on regular expressions.

Internet Programming with Python comes with a handy CD-ROM which includes the complete 1.3 and 1.4b2 versions of Python as well as source code for all the Python examples given in the book. Executable versions are included for DOS, Windows 3.1, 95 and NT, Macintosh and all of the popular Unix platforms, including Linux.

Overall, *Internet Programming with Python* is an outstanding read. The discursive portions of the book alone, in sections such as "How CGI Works," are well worth the effort.

Authors Watters, van Rossum and Ahlstrom write with wit and wisdom. They demonstrate a genuine knack for making complex subjects easy. They gently introduce the reader to the nuts and bolts of the Python language. At the same time, they consistently stay with their secondary theme by drawing all of their examples from Internet programming. And they clearly explain the application of Internet programming as they go.

The single negative is that there are a disturbingly large number of errors in the text and examples. Corrections to these have been published on the Python web site at http://www.python.org/. I strongly urge any reader who is not an experienced programmer to get and apply these corrections before reading the book.

With that single caveat, anyone interested in programming should read *Internet Programming with Python.*

As *Internet Programming with Python* amply proves, Python is a general purpose programming language which, because of its advanced features, easy extensibility and ability to incorporate modules from other programming languages, is an ideal first programming language that will greatly simplify and accelerate the development of many applications of arbitrary complexity.

After reading only the first three chapters, the reader will begin to understand why Python is rapidly becoming the language of choice for many programmers.

As Python is included in most Linux distributions, and some, such as Red Hat, make extensive use of Python in the systems administration tools they provide, *Internet Programming with Python* should be on the bookshelf of every Linux user.

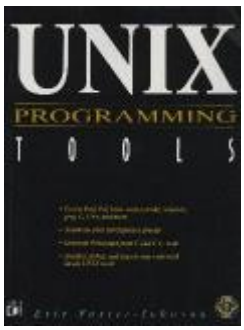
Dwight Johnson (djohnson@olympus.net) is a computer consultant living among the farms of the Sequim-Dungeness valley in the shadows of the Olympic Mountains in Washington State. He has been programming since 1967. He wrote this review in Applixware 4.3 on Red Hat Linux 4.2.

Unix Programming Tools

Andrew L. Johnson

Issue #42, October 1997

Most of the tools discussed in this book are available on the included CD-ROM, but then, most of them are included on most Linux CD-ROM distributions or are easily obtained from Linux archives.



- Author: Eric F. Johnson
- Publisher: M&T Books
- Price: US $34.95
- ISBN: 1-55851-482-1
- Reviewer: Andrew L. Johnson

Although the front and back covers of *Unix Programming Tools* and the author's introduction seem to indicate in-depth coverage, this book is really an introduction. In fact, the discrepancy between what is really covered and the implied coverage is my major gripe—but I'll take that up later. First, let's take a quick run-through of what a reader can expect.

The book itself is divided into three major sections: “Building Programs” (Chapters 1-6), “Maintaining Programs” (Chapters 7-11) and “Documenting Your Work” (Chapters 12-13).

Chapter One gives a very brief introduction to Unix, including some basic shell commands and utilities. Chapter Two whisks you through the process of

compiling and linking C and C++ programs and creating libraries. For anything beyond the basics, you'll need to consult the man pages. An example is given for creating and running a Java program, and Perl and Tcl are briefly discussed.

Chapter Three outlines the basics of using **make** to automate the build process. There is enough information here for the newcomer to begin creating and using their own Makefiles. The commands **imake** and **xmkmf** are given brief treatment, but not enough for the neophyte to begin comfortably using them.

In the Chapter Four, “Working with Text Files”, you can expect to be shown how to invoke vi or Emacs on text files, and you are provided with a few tables of the common editing commands for each editor. A handful of graphical editors are mentioned and a few screen-shots of these editors are provided. The chapter finishes with passing mention of **sed**, **awk** and **Perl**.

Chapter Five is primarily an introduction to the **grep** and **find** commands, though again, the coverage is limited. The examples of using grep all focus on finding literal text. Regular expressions are mentioned and a few character classes are shown, but with the caveat that such expressions are beyond most grep usage.

The sixth and final chapter of the first section covers installation. Here you are introduced to **tar**, **shar**, **split**, **uuencode**, **compress** and **gzip**. The **install** program is mentioned as an alternative to using tar in some situations, as well.

Section Two begins with Chapter Seven looking at the debuggers: **dbx**, **gdb** and **xdb**. You are shown how to compile with debugging turned on and how to get a stack trace, set a breakpoint and print variable values. A few graphical front ends are mentioned with screen-shots, and the C language debugger **lint** and a few memory checking utilities are mentioned. The chapter finishes with a quick run through of the Java debugger **jdb**.

Chapter Eight offers basic information on **diff** and related programs, and instructions on using the **patch** program.

Chapter Nine, on version control, is probably the best chapter in the book. This chapter begins with the barest essentials of using RCS and moves on to list the important commands and uses of RCS. Although the author gives the impression that RCS sub-directories must be used to work with RCS, the coverage is more than enough for a newcomer to begin applying version control to their projects.

The remaining two chapters in Section Two very briefly discuss cross-platform development and using **prof** and **gprof** to check program performance. These chapters, like earlier ones, provide neither breadth nor depth in their coverage.

The final section of the book is on documentation, with Chapter 12 focusing on man pages and Chapter 13 on documentation in HTML format. In Chapter 12 you are shown the basic formatting commands for creating a man page and given an example that can be used as a template.

Chapter 13 appears more focused on source code documentation and shows how one can use the tools **cocoon** and **cxref** to produce HTML formatted documentation from C++ header files or C source and header files respectively —as long as a specialized format for comments is used in those files. A similar tool for Java programs, **javadoc**, is also introduced in this chapter.

The chapter ends with a discussion of POD documentation for Perl scripts, but as POD is actually a method for easily generating man-pages, it would have made more sense to put this discussion in the previous chapter. Notably lacking in Chapter 13's discussion of documentation was any mention at all of Literate Programming techniques and tools. While Literate Programming is not mainstream, I see this as an unfortunate omission of a powerful set of tools and techniques.

As mentioned above, there is a definite discrepancy between what the covers of the book and the author's introduction claim, and what is actually covered in the book. The best example of this is the statement on the front cover: “Covers Perl, Tcl, Java, Emacs, make, sed, awk, grep, C, C++ and more.” In actuality, awk appears just twice in the book—once on page 110: “awk is another text file tool, although it's mostly used for creating files or reports on data kept in files.”; and once in the index, referencing page 110. **sed** gets three full sentences, also on page 110. Perl's coverage is limited to showing how to invoke perl on a script, or how to use the **#!** notation to create an executable script. Notably absent is any mention of Perl's built-in interactive debugging environment.

On the back cover you are told that you will find out how to “get the most out of your text editor”. However, in reality, only a basic introduction to **vi** and **Emacs** is provided in Chapter Four. In subsequent chapters, there are occasional passages on integrating these editors (mainly Emacs) with some of the other tools, but don't expect to get the most out of either of these editors with just the information contained in this book.

In the introduction, the author suggests that the book will be useful to newcomers and “hard-core UNIX developers”, and that this book will cover all the “nitty-gritty details”. However, in the summary of Chapter Two, he gives us

the following description: “Well, that's the whirlwind tour of creating C, C++, Java, Perl and Tcl programs on Unix.” “Whirlwind Tour” is an apt description for this book's coverage of Unix programming tools.

Most of the tools discussed in this book are available on the included CD-ROM, but then, most of them are included on most Linux CD-ROM distributions or are easily obtained from Linux archives. A similar level of introduction to many of the tools in this book can also be found in some of the introductory Linux books (see Other Resources), which have the added benefit of providing greater detail on the Unix/Linux environment in general.

For those wanting better coverage of the major programming tools, as well as an exploration of some programming issues in the Unix/Linux environment (such as terminal programming, sockets, semaphores, pipes, data management and more), I would suggest *Beginning Linux Programming* (see Other Resources), which, as its title suggests, is suitable for those new to Unix programming, but offers far more information than the book reviewed here.

Andrew is working on his Ph.D. in Physical Anthropology. He currently resides in Winnipeg, Manitoba with his wife and two sons, where he runs a small consulting business and enjoys a good dark ale whenever he can. He can be reached at ajohnson@gpu.srv.ualberta.ca.

Archive Index Issue Table of Contents

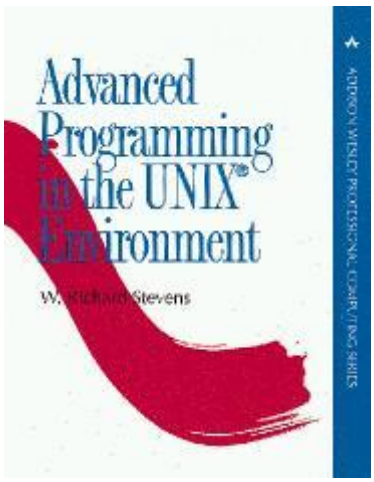
Advanced search

Advanced Programming in the Unix Environment

David Bausum

Issue #42, October 1997

The program is interactive, and it allows a user to display and print (in both textural and graphical modes) current and historical data about stocks, options and market indices.



- Author: W. Richard Stevens
- Publisher: Addison Wesley Longman Inc.
- URL: http://www.awl.com/cp/
- Price: $63.43 US
- ISBN: 0-201-56317-7
- Reviewer: David Bausum

Advanced Programming in the Unix Environment is not a new book; it was first published in 1992. However, it is the Unix programming book that convinced me that I could port a project of mine from DOS to Linux.

The Project

In 1992 I wrote a large program designed to work with the real-time stock market data feed provided by Data Broadcasting Corporation (see http://

www.dbc.com/). The program is interactive, and it allows a user to display and print (in both textural and graphical modes) current and historical data about stocks, options and market indices. In the background, the program receives serial data (via satellite to a “black box” provided by DBC that connects to a PC's serial port), breaks the data into transactions, stores the transactions in a database and feeds the transactions to the display module based on user instructions. I wrote this program on a DOS system using Watcom's 32-bit compiler and utilizing most of the 16MB DOS can address.

Within a few years I began to find DOS's 16MB limit restrictive and began looking for a new working environment. About the same time I received a sample copy of *Linux Journal*. That particular issue (#14, June 1995) had the words “Intelligent Serial Boards” on its cover—those words got my attention. By then I had upgraded my program to work with a Digiboard, and I knew one requirement for a migration to a new OS would include being able to use it or another intelligent board. I subscribed to *LJ* and began to look for books relating to Linux/Unix.

When I began programming in the early eighties, I learned Basic and FORTRAN by reading what I call “no-name” books. By that I mean there was nothing memorable about either the books or their authors' style. By contrast I learned Cobol from McCracken and C from Kernighan and Ritchie. Anyone familiar with either book will recognize the difference between those books and “no-name” ones. I realized that if I hoped to port my DOS application to Linux, I needed to find a book which would do for Unix what the book by Kernighan and Ritchie does for C.

The Book

In the second half of 1995 and the first part of 1996 I kept running into references to *Advanced Programming in the UNIX Environment* by W. Richard Stevens. In the Spring of 1996 I ordered the book, and it has turned out to be everything I had hoped it would.

It is a big book—over 2 inches thick and more than 750 pages in length. It is hard cover, and it lies flat when open. It is an attractive book—both the cover and the general page layout. In the preface Stevens says he used **troff** and **groff** to format and prepare the camera-ready copy for the book. As one who has prepared copy for more than one book, I know that an attractive book does not just happen. It takes expertise and time, and Stevens's effort makes the reader's journey through the book more enjoyable.

The book discusses over 220 functions used by various Unix libraries. When a function is introduced, it is placed in a box with system *include* files required by

the function, information about function returns and a function prototype. The book has over 10,000 lines of source code (all in C) and is filled with numerous small program examples. The code is available by FTP and was tested by Stevens on four flavors of Unix. The book has numerous tables which group together flags of a particular type or other information. It has numerous figures which show the relationships of the items under discussion. Where appropriate, it includes the output from an example and uses it to clarify or emphasize the current discussion item. It does these things within the context of the included Unix variants (listed below). Where necessary a topic is described for each variant. With a lesser writer the encyclopedic detail would become suffocating, but Stevens surrounds all of the above items with enough text to make a very readable book and an extremely valuable reference.

In the preface Stevens breaks the book into 6 parts and briefly describes each part. The following is a slightly different breakdown with longer descriptions.

Part 1 contains Chapters 1-6 (160 pages). The first two chapters provide a gentle introduction to Unix and a discussion of the included variants of Unix. They are SVR4 which dates to 1990, 4.3+BSD which refers to the state of BSD in early 1992 and POSIX.1 which dates to 1990. The remaining four chapters in this part (over 100 pages) discuss files, directories, access permissions, inodes, file I/O and special files such as the password file. This material duplicates Chapter 8, “The Unix System Interface”, in the Kernighan and Ritchie book, but it goes into much more detail and discusses many more topics.

Part 2 contains Chapters 7-9 (100 pages). These chapters deal with processes. Topics discussed include:

- what happens when a process starts and when it stops
- how to access program arguments and environment variables
- memory allocation
- process resource limits
- process creation via forks and execs
- process IDs and other properties
- the relationship between a parent and a child process

Part 3 contains Chapters 10-12 (150 pages). At this stage we are about one third of the way through the book, and the going has not been too rough. That changes with this part. Chapter 10 deals with signals, Chapter 11 with terminal I/O and Chapter 12 with advanced terminal I/O. Once Stevens describes each of the over 30 signals available, he shows how signals in early versions of Unix were unreliable. Then, he describes functions introduced with POSIX.1 which make signals safe to use. Fortunately, these POSIX safe functions are available

in Linux. His treatment of terminal I/O begins with an examination of the termio's structure which holds over 50 special flags (or switches), characters that are given special treatment during input and baud rates. He shows how to get and save these values. He describes canonical (i.e. line oriented) and noncanonical I/O. Again, I found his discussion highly applicable to Linux. Chapter 12 deals with various additions to the I/O system discussed in part 1 and in Chapter 11. Some of the topics included are nonblocking I/O, record locking, streams, multiplexing (via select or poll), asynchronous I/O and memory mapped I/O. The select function is available in Linux, but I don't think all of these topics are—yet.

Part 4 contains Chapters 13-15 (100 pages). Chapter 13 is a short chapter dealing with daemon processes. Chapters 14 and 15 deal with interprocess communication. That discussion begins with pipes and winds up 80 pages later with two examples of sets of functions used in client-server programs. Stevens treated part of Chapter 15 in much more detail in his 1990 book *UNIX Network Programming* (Prentice Hall).

Part 5 contains Chapters 16-18 (115 pages). At this stage we are two thirds of the way through the book, and the “advanced” in the title is beginning to show (to this DOS programmer). Each of the chapters in this part is devoted to developing a single program or library—a database program, a PostScript printer program and a modem dialer. I spent several days studying the printer program and learning how the logging facility used by the kernel and several daemons works.

Part 6 contains 1 chapter, 3 appendices, a bibliography and an index (120 pages). Chapter 19 deals with pseudo terminals. Appendix A is very valuable, providing a 20-page summary of all the functions introduced in the book. The material is arranged alphabetically and includes a function prototype, return values, required system include functions and a reference to the page which introduces the function. Appendix B provides source code used by many examples in the book. Appendix C provides solutions to some of the exercises which end each chapter. The index is quite complete (25 pages).

When I finish a book, I have several criteria for deciding if I can recommend it. Where does it now live: on my desk, in a pile of books near my desk or on a distant shelf? Was the book interesting in a general sort of way or did it change the way I think or act? Stevens's book lives on my desk, often open to a particular spot. Also, it has given me many ideas I am using in the port of my stock program to Linux. Three examples are serial I/O, coordinating multiple workers and error messages.

Chapter 11 made it relatively painless for me to get both standard serial ports and my Digiboard working under Linux. Chapter 10 gave me the idea of adding Unix signals and timers to the central work loop of my DOS program. This allows me to use the same program structure as the DOS program. In addition, it allows the program to sleep when it has no work and to awake when it does. Thus, it will be Linux “friendly”. Chapters 13 and 17 together with Appendix B showed me how to log errors in a special file. Using a single line, my program can call a function that writes to standard error, a program specific log file or both. The call uses a **printf()** format with a variable number of arguments, and it can either end the program or return to the calling function.

Conclusions

I have felt for many years that programming and construction are related. Migrating from DOS programming to Linux is like moving from wood-frame house construction to the construction of sky scrapers using concrete, steel and glass. Such a transition requires understanding new materials and how they interact. That is precisely the information provided by Stevens's book. He describes basic library calls, uses them in small code segments and pieces them together in several larger projects. I heartily recommend this book and hope your mileage will be as good as mine.

David Bausum received a Ph.D. in mathematics from Yale in 1974. Since the early 80's most of his energy has gone into software development and related activities. He coedited The Journal of Military History Cumulative Index: Vols 1-58, 1937-1994. He can be reached via e-mail at davidb@cfw.com.

Archive Index Issue Table of Contents

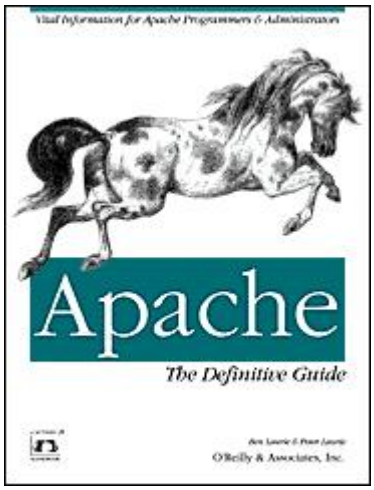
Advanced search

Apache, The Definitive Guide

Luca Cotta Ramusino

Issue #42, October 1997

I jumped at the chance to read and review Apache, The Definitive Guide. I was not disappointed.



- Authors: Ben Laurie and Peter Laurie
- Publisher: O'Reilly & Associates
- ISDN: 1-56592-250-6
- Price: $34.95 US
- Reviewer: Luca Cotta Ramusino

My first installation of Apache dates back to version 1.0.0. I wish I could say it was an exercise in TCP/IP wizardry, but thanks to the program's careful engineering, it was really nothing of notice. At work, I noticed a little-used Sun SPARCstation 4 not connected to the outside world. Since I had wanted to experiment with **http** and the Web for a long time, I downloaded a Solaris binary of Apache from the canonical ftp site, uncommented a couple of lines in the configuration files, started the **http** daemon and went to look for coffee. I added a script for starting and stopping Apache on system boot and shutdown

and haven't touched it since. As far as I know, “Little Sparky” is still happily serving HTML to all interested parties.

After over a year of faithful service, I felt it was time to add some whizbang to Little Sparky. Being a big fan both of “In a Nutshell” books and the Apache http server, I jumped at the chance to read and review *Apache, The Definitive Guide*. I was not disappointed. The book actually lives up to its bold claim, as it contains everything you will ever want to know about the world's most popular web server, although sometimes the information provided is quite terse.

Weighing in at a little over 250 pages, the book can be roughly divided into three sections, each more advanced than the previous. The authors warn that unless stated otherwise, all information is applicable to Apache version 1.1.1, and they admit that Apache is a moving target. Keeping the book current with the latest code development has not been an easy task.

The first part of *Apache, The Definitive Guide* covers, among other things, basic server operations, CGI scripting and user authentication.

The second part addresses more specialized needs such as language arbitration (i.e., delivering specialized content in response to the browser's preferred language setting), server-side includes and the use of Apache as a proxy server.

Finally, the third part caters to the more adventurous palates. Intrepid webmasters wishing to extend and modify Apache's standard behavior are treated to three chapters on the Apache API and on writing custom Apache modules. Not surprisingly, one of the authors is a member of the core Apache programming team. Alternatively, if you like to search the web for ready-made features not included in the base distribution, you can read about the most popular contributed modules, such as FastCGI, a server module designed to improve the execution of CGI scripts.

The book travels at a fast pace throughout—sometimes a little too fast, given the richness and complexity of the information available on Apache. Take virtual hosting, for example. Virtual hosts are a popular means for Internet service providers to offer rented web space to different domains using the same machine.

Virtual hosts are introduced as early as Chapter 3, and the topic is immediately exhausted by handling **httpd** and Unix configuration in a scant three pages. Another case in point is cookies, a clever device invented by Netscape to counter the Web's inherent *statelessness*. Cookies are given only cursory

coverage while describing the server configuration directive **CookieTracking**. Personally, I would have wished for a more extensive discussion.

The authors give plenty of examples whenever they introduce a new concept or topic. All of the code, HTML documents and configuration files are neatly organized on the companion CD-ROM, so that you can exactly reproduce the sample sites simply by copying the appropriate directory tree to your hard disk.

A word of warning—while the authors briefly summarize the workings of Unix and TCP/IP, you *really* need to be competent in both before you pick up this book. Also, it helps if you have a correctly configured system before you start playing with the examples provided.

If you have a slow link to the Internet, you will appreciate the added convenience of two Apache distributions on the accompanying CD-ROM. You get both a stable 1.1.1 and a more advanced, but potentially misbehaving, 1.2b4. The distributions are all source code, as the book is not geared specifically to Linux. However, compilation and installation is straightforward and complete directions are given at the beginning of the book.

I liked *Apache, The Definitive Guide* for a number of reasons. In short, the book has everything you need to get started with the Apache web server, and to help you progress from there to more complex operations. I believe it complements nicely the skimpy “official” documentation available on Apache's web site. Thanks to its broad coverage of topics, the book should appeal to both beginners and experienced webmasters.

Luca Cotta Ramusino works at Foster Wheeler Italiana, where he tries to pass as a “suit” in order to promote the use of Linux. He can be contacted via e-mail at lcotta@systemy.it.

Archive Index Issue Table of Contents

Advanced search

Linux as an Internet Kiosk

Kevin McCormick

Issue #42, October 1997

Here's a company that uses Linux to set up Internet terminals in cybercafes.

If you are at all interested in consumer applications of the Internet, you have no doubt heard about cybercafés and so-called “Internet kiosks”. Both attempt to bring ubiquitous Internet accessibility to the general public. In cybercafés, the idea is to let you browse the Internet (or even do real work) while basking in the comfort of a familiar café. Internet kiosks are supposed to be quick and easy-to-use stations for checking e-mail or looking up a bit of information on the Web.

Café Liberty, a cybercafé in Cambridge, Massachusetts, was the first of its kind in the greater Boston area, founded in the fall of 1994. From the start, we have had several computers available for use by customers for a small hourly fee. Our own experiences, coupled with that of other café owners, eventually led us to the conclusion that there were a lot of us interested in giving their customers Internet access, but who had absolutely no idea how to go about it.

In March 1996, the NetPod project was born, to be funded and managed by Café Liberty. We set out to build a public access Internet terminal, a “NetPod”, that would allow anyone with a dollar to surf the Web, TELNET, check e-mail or do any common Internet function. The system had to be extremely easy to use, foolproof, fast, reliable and painless to install in the target site. Perhaps not surprisingly, we chose Linux for every machine in our network. The article, “Linux—The Internet Appliance?” by Phil Hughes, in the April 1997 *Linux Journal* struck a chord with us, since much of what he said is exactly what we are doing. While our system is not designed for the home, it does attempt to provide an interface that anyone can understand, it does it on cheap hardware, it is an Internet appliance and, of course, it runs Linux.

The NetPod Interface

In order to understand our network architecture, it's necessary to understand what kind of services we wish to offer. When a user approaches the NetPod, he sees a heavily modified XDM (X Display Manager) which presents him with several options: Guest Login, Account Login, About and Create Account (see Figure 1). The machine costs six dollars an hour to use, paid in increments of one dollar for ten minutes. Users insert bills into a bill validator on the side of the NetPod (see Figure 2).

Figure 1. Login Screen

Figure 2. NetPod Terminal with Validator

A guest login provides access only to Netscape Navigator and to TELNET, while an account (which costs three dollars a month) also provides e-mail (both sending and receiving) and access to Usenet newsgroups from a full news feed.

After login, the user is presented with a large Netscape window and a tool bar on the right hand side of the screen that lets them switch between the various parts of the system (see Figure 3). Their remaining time is also displayed on the tool bar. When an account holder logs out, their remaining money is kept for their next login session. When a guest logs out, any remaining money is lost.

Figure 3. Netscape Window with Toolbar

As mentioned earlier, we used XDM for the login screen. This is because XDM is relatively easy to modify and takes care of all the details of logging a user in, like setting X access controls and creating an environment. The entire login program was simply linked to XDM through the **Greet()** function. Since we didn't want to deal with much of the bloat of Motif or the complexity of the X toolkit, we designed our own library of GUI tools for the NetPod. This let us tailor every last detail of our system's appearance. It also let us easily construct unique interface elements, like shaped graphic buttons, some of which can be seen in Figure 1.

The tool bar was considerably more difficult to write. We chose to use FVWM as a window manager because it is easily customizable and provides a **module** function that allows a separate program to probe FVWM's internal window information and structures. We configured FVWM to allow no window movement, no special menus and no key control; in short, it was locked up tight so the user couldn't do anything involving window management.

The tool bar continually receives information from FVWM about window updates so we can extract the window IDs of Netscape windows, TELNET

sessions, etc. and move around the windows when the user clicks on the tool bar buttons.

This scheme would work fine, if Netscape were a relatively simple program, but Netscape opens many other windows during normal operation, such as mail and news windows, dialogs and others. Keeping track of all these windows is a bit of a chore, and it has only become more difficult with our recent incorporation of Netscape Communicator, which opens far more windows than Navigator.

Simple though it may be, many users have praised our straightforward interface for its ease of use and its speed. People used to sluggish Windows and Mac systems simply cannot believe machines like ours (Pentium 100 with 16MB RAM) can be as quick and responsive as they are—which is due in large part to our use of Linux and the X Window System.

The Network

Our network is a mishmash of random hardware and hand-built software. Coming into Café Liberty are two 384Kbps fractional T1 lines. One runs to our ISP and provides Internet service using the Cisco HDLC protocol. The other is a frame-relay line that serves the NetPods. The two T1s are attached to our router (a 486DX/40 running Linux), which also has two Ethernet cards. One Ethernet is for the café (including network jacks for public use), and the other is for NetPod machines.

You don't often hear of a single Linux box being used as a router for multiple Ethernets and T1 lines with differing protocols, let alone a wimpy 486DX/40. But, thanks to SDL Communications' RISCom/N2 cards, it is possible. One RISCom/N2 with a T1 CSU/DSU attached to its v.35 port runs the frame-relay connection, and the other RISCom/N2csu (with built-in CSU/DSU) runs the Cisco HDLC Internet connection.

Getting the cards up and running is mostly straightforward, though there are many small details, like electrical characteristics of the T1 lines, that we sometimes had to just guess at and hope it worked. After the cards were up, routing was easy—just compile IP forwarding into the kernel, set up some subnets and all falls into place.

The NetPod Ethernet has only a couple of machines on it, the most important of which is the NetPod server. The server is a Pentium 100 (running Linux) with 32MB of RAM that deals with NetPod mail (using ZMailer), the Usenet feed (inn), NFS serving of users' home directories, and the NetPod database. The database, which keeps track of user information, was custom-written and communicates with the NetPods over encrypted channels for security.

The server has IP firewalling turned on in full force. While anybody can connect to the SMTP, POP and IMAP daemons, everything else (most specifically NFS) is tightly controlled.

Each NetPod has a 56Kbps frame-relay connection. As far as we know, we are the only Internet terminal company in the world to have chosen frame-relay; the vast majority use standard phone lines and a couple use ISDN. Each frame-relay line is assigned a different PVC (Permanent Virtual Circuit) number. Their data is consolidated and appears on the frame-relay T1 at café Liberty.

Though frame-relay may seem like massive overkill for an Internet terminal, it has enormous benefits.

Frame-relay is fast. The latency on frame-relay lines is a lot less than a PPP connection over a modem. This translates directly into faster TCP connection times for Netscape. We are guaranteed 56Kbps of bandwidth, while the average 33.6Kbps or 56Kbps modem is unlikely to ever get its theoretical maximum bandwidth.

Frame-relay is cheap. Our lines cost somewhere around $150 a month. This may seem like a quite a premium to pay for the comparatively mediocre speed benefits mentioned above, but remember that Massachusetts is in NYNEX (phone company for northeast U.S.) territory. [NYNEX, which covered the east coast of the U.S. from Maine to New York, and Bell Atlantic, which covered from new Jersey south to Virginia, have just merged. The combined company is Bell Atlantic—Ed.] We have found that NYNEX would charge us *thousands* of dollars per month for a 64Kbps ISDN connection up 24 hours a day. The situation with standard analog dial-ups isn't much better; there is simply no way to avoid a ludicrous per-minute charge on business phone lines of any type.

Thus, frame-relay was the obvious choice. Yet the startup costs for frame-relay are crippling. One SDL RISCom/N2dds (with built-in 56Kbps CSU/DSU) and frame-relay software costs $750. The installation costs for the line are several hundred more. Despite these numbers, we feel that there are very real benefits to the frame-relay technology. Being able to remotely access the NetPods at any time of day or night is immensely useful.

In an attempt to reduce the cost of frame-relay, we have embarked on a side project to add frame-relay support to Linux. We had often wondered why Linux had no frame-relay protocol driver. It seems that the main reasons are complexity and cost. You can tell that parts of frame-relay were designed by committee; every person involved managed to get his or her pet feature fit in somewhere, yet they don't all fit quite right. All the relevant standards

documents have to be purchased from the ITU (International Telecommunications Union) and then deciphered.

We figure we can drop the cost of frame-relay about $500 if we use a BAT Electronics CSU/DSU (the most inexpensive we could find) and then tack on a minimalist synchronous to asynchronous converter of our own design. This converter takes the CSU/DSU's synchronous bitstream, repackages it in RS-232 bytes at 115,200 baud and sends it to the NetPod, which then interprets it with the frame-relay driver. As of early June, we have established successful TCP connections through this system, and we are now concentrating on improving the stability and performance of the drivers. Once they are ready for release, we will donate the drivers to the standard kernel distribution so that any device that generates an HDLC-framed bitstream (like Z8530-based synchronous cards) can be used for frame-relay.

We are a 100% Linux shop. Except for some graphics work that we do on Macintoshes to keep professional printers happy, every last machine is a PC running Linux. Linux has given us so much that, in the true spirit of Linux, we feel we have to give something back to the community, which we hope to do with our frame-relay package.

The Competition

Though we were one of the first, we are certainly not the only company trying to create a public Internet access solution. We are initially targeting a small segment of the market, namely cafés and similar establishments, but we realize that our system can be applied to a much broader range of sites. We feel that we have, in a general sense, the best Internet terminal system on the market.

We feel this way for a number of reasons. First, we have an interface that is as idiot-proof as we can make it (Netscape notwithstanding). We have a fast and reliable link to the Internet that is up all the time. We provide accounts and other services that others do not. We don't make users pay through the nose to use our terminal.

By comparison, other Internet terminals (most of which run Microsoft operating systems) are more geared towards giving the user “neat toys” to play with, without providing any incentive to actually *use* the system. In order to get these toys, the machines have to run Windows, which means that they give up all the benefits of Linux that we have exploited. They are unstable and crash all the time. They do not provide an easy-to-use interface. Billing systems are often kludged on; many systems simply cut power to the keyboard and mouse when a user's time is up. They provide minimal security; on virtually every terminal we have played with, we have managed to get full administrative access to the system in only a few minutes.

The fundamental point to remember here is that, without Linux, we could not have done **any** of this with as little capital, as little time and as few headaches as we did. Some of it is outright impossible with any other operating system, even other Unices. Because Linux is Unix-like, we can control every aspect of the system. And because Linux is free, we can do things like add our own frame-relay drivers to the kernel with impunity.

The Future

Our first NetPod became available for public use at café Liberty in August, 1996. After many months of feedback from our users (and after waiting three months for NYNEX to install our frame-relay lines), we placed two more NetPods at the Someday café and Seattle Joe's café in the Cambridge/Boston area in February, 1997. We are preparing for a massive set of new sites this Fall.

Building the infrastructure for the NetPod system, both the network and the core software, has been challenging, but we have shown that it works, and that it works well. As we faced each challenge, we saw that a box running Linux could provide a solution. Now we can concentrate on adding new features to expand the appeal of the system.

Unfortunate though it may seem, it turns out that many of our users want to access their America Online accounts through our NetPods. We purchased a copy of Wabi (Windows application binary interface) through Caldera, and we have found (much to our amazement) that it actually runs AOL's Windows software, and in fact runs it quite well. We recently incorporated it, along with many other new features and a streamlined, more complete set of user tools. There are still many new features under construction.

In short, if it involves networking, we want to add it to the NetPod. The reason the NetPods can do what they do, and the reason we're justified in even considering some of the crazy ideas we have, is that we use Linux on every system in the NetPod network. Just imagine using Windows 95 to implement multiuser access controls over a distributed file system using frame-relay, then switching from Netscape to an SVGAlib virtual terminal running Quake, then to AOL running inside a Windows emulator. It should send shivers up your spine. With Linux, it's a piece of cake.

Contacts

Kevin McCormick is a senior at the Massachusetts Institute of Technology, majoring in Electrical Engineering and Computer Science. He is the head programmer for the NetPod project. When not slaving away at MIT or NetPod, he tries to network everything in sight to the Internet, including (no kidding) the toilets at his fraternity. He can be reached at fbyte@netpod.com.

LINUX JOURNAL

Integrating SQL with CGI, Part 1

Reuven M. Lerner

Issue #42, October 1997

This month, Reuven shows us how to send and retrieve postcards on the Web using a relational database and CGI.

Last month, we began our exploration of integrating relational databases into our CGI programs. CGI programs often have to save and retrieve information. They typically do this using text files on the server's file system. By using a relational database, however, we can make our programs flexible, powerful and robust, while at the same time reducing the amount of code that we have to write.

Like the Web, relational databases use the client-server model, dividing the world into database clients (which make requests) and servers (which respond to those requests). Requests are typically written in SQL (Structured Query Language), which can be embedded within programs.

This month, we look at a simple project that uses a relational database to allow users to send electronic postcards to each other. Next month, we will spend more time on this project, improving on our original database design and making the program even more useful.

Creating a Postcard System

Over the last few years, “postcard” web sites have become increasingly common. These sites, which often appear just before a major event or holiday, allow people to send electronic postcards to their friends and family.

One way to accomplish this task is to e-mail the postcard, perhaps as a MIME attachment. This is relatively easy to do and makes the system relatively simple. However, such a system requires us to send the entire postcard, an operation which can use up a great deal of bandwidth if our site gets a large number of visitors.

In addition, many users still have to pay for their Internet connection and might not wish to be sent an unsolicited 100 kilobyte mail message, even if it does contain a beautiful picture and warm wishes. This can be a particular problem in the modern age of spamming, when people send each other large quantities of mail without regard for the fact that the recipient might end up having to pay for telephone and networking charges.

Add to that the fact that not everyone uses a MIME-compliant mail reader, and it becomes clear that at least for the time being, sending large, sophisticated e-mail messages is not a good way to make friends.

Thus, we will use a different approach. The postcards will be housed in a database on our server and will be accessible via a CGI program. When a postcard is created, a short e-mail message will be sent to the recipient, indicating the URL from which she can retrieve the postcard, using the web browser to display any graphics.

We will write two CGI programs: one to create the postcard and send the notification via e-mail, another to allow the recipient to retrieve the postcard.

Creating the Database

Before we can write our programs, we need to create a table in our relational database. For the purposes of demonstration, I'm using MySQL 6.3, a SQL database that runs nicely on my Red Hat Linux system. You can get more information about MySQL at http://www.tcx.se/.

In order to create the database tables, we need to decide what information we wish to store about each of the postcards. And in order to do that, we need to know how we wish the postcards to look.

Let's assume that the postcards are web pages constructed on the fly by a visitor to our site. A specific postcard is created when a CGI program receives a URL containing a unique identifier following a question mark, known in Web lingo as the “query string”. Thus, one postcard would be available via the URL http://www.oursitename.com/cgi-bin/show-postcard.pl?12345, while another postcard would be available via the URL http://www.oursitename.com/cgi-bin/show-postcard.pl?67890.

The CGI program show-postcard.pl takes the value of the query string (which, in the above examples, is either 12345 or 67890) and uses it as an index in our database table. That table contains a short message from the sender, as well as a picture of the sender's choice.

In order to make this work, we need a table with seven columns:

1. The postcard ID
2. The sender's name
3. The sender's e-mail address
4. The recipient's name
5. The recipient's e-mail address
6. The graphic of the postcard
7. The text of the postcard

To avoid dealing with MySQL's security system, we put all of our table information in the database named “test”. We can enter the MySQL command-line interface by entering the following on the command line:

```
mysql test
```

On typing that command on my system, I was greeted with the following message:

```
Welcome to the mysql monitor. Commands ends with ; or \g.
Type 'help' for help.
mysql>
```

The **mysql>** prompt is MySQL's way of signalling me that it is waiting for an SQL command.

Just as in C and other programming languages with typed variables, columns in an SQL table must have a data type associated with them. The main data types we will use are **mediumint** (a medium-size integer), **varchar** (a string of variable length) and **blob** (an untyped storage element that accepts large amounts of data).

First, we'll create a table with seven columns, one for each of the pieces of data we wish to track, using the following SQL query:

```
create table postcards (
        id_number mediumint not null primary key,
        sender_name varchar(60) not null,
        sender_email varchar(50) not null,
        recipient_name varchar(60) not null,
        recipient_email varchar(50) not null,
        graphic_name varchar(100) not null,
        postcard_text blob);
```

When I enter this at the MySQL prompt, I get the following output:

```
mysql>     create table postcards (
   ->   id_number mediumint not null primary key,
   ->   sender_name varchar(60) not null,
   ->   sender_email varchar(50) not null,
   ->   recipient_name varchar(60) not null,
```

```
    ->   recipient_email varchar(50) not null,
    ->   graphic_name varchar(100) not null,
    ->   postcard_text blob);
Query OK, 0 rows affected (0.02 sec)
```

Our database now contains a table named “postcards” with the appropriate seven columns. Notice how each column, except for postcard_text, is defined as being “not null”. This indicates that the field must contain a value. By defining the table in this way, the database server can enforce some conventions about how the data is stored, thus avoiding potential problems.

The first column, id_number, will be used to identify a particular postcard. In order to ensure that only one postcard has this particular ID number, we create the id_number column with the keywords “primary key”, which is another way of saying that its value must be unique across all rows. By making id_number a primary key, we can retrieve postcards by asking for all rows with a particular value of id_number—the result will be either a single row (with a matching value of id_number) or no rows at all (indicating that no postcard exists with that ID number).

The second through sixth columns are of type **varchar**, which simply means that their lengths vary as necessary, up to the maximum number of characters indicated in the parentheses. I chose these numbers somewhat arbitrarily; if you suspect that e-mail addresses will always be fewer than 50 characters, then you might wish to shorten the appropriate fields. By the same token, if you suspect that users might have names longer than 60 characters, you should extend the lengths of these fields.

The sixth column, graphic_name, contains the pathname of the graphic to be inserted in the postcard.

The final column, defined as type “blob”, contains the text that the sender wishes to send to the recipient. Because this message is optional, we have indicated that this column can contain a null value. MySQL allows us to enter a value in postcard_text, but if we fail to provide such a value, the column will simply contain a null value.

For a summary of our table, we can use the MySQL **describe** command:

```
mysql> describe postcards;
```

The output of this command is shown in Table 1.

Table 1. Output of describe Command

Storing and Retrieving Postcards

Now that we have created our “postcards” table, let's insert some dummy data directly at the MySQL prompt. Then we will write the program show-postcard.pl to display the dummy postcard. Finally, we will write a program to allow users to enter postcards via an HTML form.

We can insert data into a table by using the SQL **insert** command. Let's say we wish to insert a postcard with the following information:

```
ID: 12345
Sender name: Reuven Lerner
Sender e-mail: reuven@netvision.net.il
Recipient name: Bill Clinton
Recipient e-mail: president@whitehouse.gov
Graphic: smile.gif
Text: Hey there, Mr. President!
```

To insert this information, use the following SQL command:

```
insert into postcards
  (id_number, sender_name, sender_email,
   recipient_name, recipient_email, graphic_name,
   postcard_text)
values
  (12345, "Reuven Lerner",
   "reuven@netvision.net.il",
   "Bill Clinton",
   "president@whitehouse.gov",
   "smile.gif",
   "Hey there, Mr. President!");
```

Typing this command at the MySQL prompt produces this response:

```
Query OK, 1 rows affected (0.34 sec)
```

In other words, one new row was successfully added to the table. To retrieve it, use the SQL command:

```
select * from postcards where id_number = 12345;
```

which produces the result shown in Table 2.

Table 2. New Database Entry

While it might look ugly, this output actually makes sense. The problem is that most CRTs are only 80 columns wide, while the table is nearly twice that width. (And this magazine is even narrower, making it even worse.) Luckily, when our program retrieves a row, it does not have to worry about the formatting.

If you are interested only in certain columns, you can specify a subset of the entire row, as follows:

```
select sender_name,graphic_name,postcard_text
 from postcards where id_number = 12345;
```

Submitting this query produces the result shown in Table 3. The output row contains only the row corresponding to our postcard, and the columns we have requested. As an added benefit, it is easier to read than the entire row that we retrieved earlier.

Table 3. Output of select Query

Now that we have seen the sort of SQL query that is necessary to retrieve information from the database, it should be a snap to write our CGI program. You can see an initial stab in Listing 1.

This version of show-postcard.pl expects to be invoked with a single argument (the postcard's ID number) in the query string, as we mentioned earlier. If we go to the URL /cgi-bin/show-postcard.pl?12345, we should see a postcard addressed to Bill Clinton, with the message that we inserted by hand at the MySQL prompt.

The program works in a relatively straightforward way. First, it creates an instance of CGI, a Perl object (available from CPAN at http://www.perl.com/CPAN/) that makes CGI programming easier. After sending a MIME header indicating that the program will send its output in HTML-formatted text, we retrieve the value of the query string by using the **param** method, as follows:

```
my $id = $query->param("keywords");
```

If the query string is empty, we print out an error message and give the user another chance to enter a postcard ID number, by using the little-known <isindex> tag. Note that we check the numeric value of $id, by using the == operator, rather than simply checking to see if $id is equal (with “eq”) to the null string. This prevents problems if someone invokes show-postcard.pl with an argument that includes characters other than numbers.

Assuming that a number does arrive in the query string, we connect to the test database, and then build up our SQL command in the variable $command. We could, of course, simply insert the command string inside of the call to $dbh->query. However, building the command in a separate string makes it easier to understand. It also has the added benefit of allowing us to debug the program. We can print the literal query by uncommenting the following debugging line:

```
# Uncomment for debugging
# print "<P>SQL command: \"$command\
        "</P>\n";
```

When the above line is uncommented, we can see exactly what is being passed to MySQL and find our program's problems more easily.

Our program then sends the request to MySQL using the “query” method. The value returned is a handle into the database, which can contain one or more rows matching our query. Since we are requesting all rows matching a particular ID, we can be sure that no more than one row is returned. This is true because we set id_number to be a primary key, which makes it unique.

What happens if the user invokes show-postcard.pl with an ID number that does not correspond to any postcard? If we were to ignore this possible error, users entering nonexistent ID numbers would see the outline of a postcard, but no actual content. This isn't particularly friendly for our users, who would like to know when they make a mistake and to be given a chance to correct it. Thus, before we retrieve information from the returned row, we make sure that there *was* a returned row. If not, then we can safely assume that this is because the ID number submitted by the user did not match the id_number column for any row in the database.

Our code accomplishes this checking by using the “numrows” method on the statement handle ($sth)--the object which allows us to read the results of our query. If numrows equals 0, we have not received any rows from the server, and we complain to the user that he has entered an ID number which does not match any on our server.

Obviously, to be displayed, the graphic with the specified name must exist in /tmp (or whichever directory you name in the final part of the program) on the server. If the graphic's name is misspelled or if it is placed in the wrong directory on the server, the program will not be able to display it, so take care. (We will take a closer look at this problem next month, when we look at the use of multiple tables for the graphic.)

Finally, show-postcard.pl turns the row into a Web page that can be sent to the user. Indeed, since the “postcard” is really a Web page, you could argue that, while I have talked about this project as if it were useful only for sending postcards, you could easily adapt this strategy for creating personalized home pages on your system, with each user getting a different page.

Creating the Postcard

Now that we can retrieve postcards without too much trouble, we have to take care of the final part of this project: allowing users to create postcards using HTML forms.

The basic idea is as follows: The sender enters all of the necessary information into an HTML form. The CGI program receiving the submitted form saves the data to the “postcards” table, sends e-mail to the recipient indicating how to retrieve the postcard, and thanks the sender for using our service.

We have already seen how to insert data into the table using an SQL query. All we have to do now is create a CGI program that turns the contents of a form into such a query, and an HTML form that submits its data to our program. You can see an example of such a program, send-postcard.pl, in Listing 2.

Listing 2. Program send-postcard.pl

In many ways, send-postcard.pl does the same thing as show-postcard.pl. It takes variable values from the HTML form and inserts those values into a canned SQL query. That query is then sent to the database server, which processes it—in this case, by inserting a new row into the database.

As you can see from the listing, we first grab the contents of each of the HTML form elements. In this particular version of the program, we do not check the lengths of each of the fields. It would undoubtedly be a good idea to do so in a production version, given that the database has been instructed to accept names and addresses with a certain maximum length.

Next, we create an ID number for the postcard:

```
my $id_number = time & 0xFFFFF & $$;
```

Why didn't we take a simple value, such as time (the number of seconds since January 1, 1970) or $$ (the current process ID)? And why do we perform a bitwise “and” on these values? Because the ID number must be unique; otherwise the database will not accept the new row. We also want to avoid sequential numbers, so that users will not be able to easily guess the numbers. This is far from random and can be guessed by someone interested in doing so; however, it is better than nothing at all and makes life a bit more interesting.

Finally, we create the entry for this postcard in the table, building up the SQL command little by little:

```
my $command = "";
$command = "insert into postcards ";
$command .= "  (id_number, sender_name,
        sender_email, recipient_name, ";
$command .= "   recipient_email, graphic_name,
        postcard_text) ";
$command .= "values ";
$command .= "  ($id_number, \"$sender_name\",
\"$sender_email\", ";
$command .= "   \"$recipient_name\", \"$recipient_email\", ";
$command .= "   \"$graphic_name\", \"$postcard_text\") ";
```

Notice how we have to surround all but one of the values with quotation marks. This is because they are character values and blobs (as opposed to integers), and thus must be quoted when passed in an SQL query.

Once the SQL query has returned, we know that the postcard has been inserted into the database. Unless, of course, $sth is undefined, in which case we die inelegantly with an error message.

```
   # Make sure that $sth returned reasonably
   die "Error with command \"$command\""
            unless (defined $sth);
```

Finally, we send e-mail to the recipient indicating that there is a postcard waiting for her, along with the URL for retrieving the postcard. So long as the ID number stored in the database matches the value of $id_number in our program, we should not have any problems. We finish up by thanking the sender for using our system.

Sending the Postcard

Now we come to the part which will enable our users to send postcards to each other: The HTML form from which the information is submitted to the send-postcard.pl program.

This form, as you might expect, is relatively straightforward. It contains five text fields, one for each of the fields we expect to get from the user, as well as a text area into which the user can enter arbitrary text. You can see the page of HTML for yourself in Listing 3.

Listing 3. HTML Form for Submitting Postcard

This system, while a bit crude, does demonstrate how to create a postcard system on your web site with a bit of work. In addition, by taking advantage of the power of SQL and the features of a relational database, we created a relatively robust system without a lot of work and without having to debug a lot of code.

You could easily add another few HTML form elements to postcard.html, making it possible for the sender of a postcard to set the background color, text style and font of a particular postcard. The possibilities are indeed limitless, although you should avoid making such an HTML form look like the cockpit of a jumbo jet.

There are, of course, a number of loose ends with this project. One such problem has to do with the graphics, which we mentioned briefly above. In addition, what happens if the ID number is lost? Currently, there isn't any way for someone to come to our site and retrieve any postcards that they might have sent or received. We will take care of that next month, as we continue to look at and use SQL in our CGI programs.

Reuven M. Lerner is an Internet and Web consultant living in Haifa, Israel, who has been using the Web since early 1993. In his spare time, he cooks, reads, and volunteers with educational projects in his community. You can reach him at reuven@netvision.net.il.

Archive Index Issue Table of Contents

Advanced search

Letters to the Editor

Various

Issue #42, October 1997

Readers sound off.

Corel License from Caldera

In Phil Hughes' response to Thomas L. Gossard in the July LTE column, he said that Corel's license to Caldera for WordPerfect allows its use only on Caldera's version of Linux.

Corel's web page (http://www.corel.com/) has links which lead to a page on http://www.sdcorp.com/ for WordPerfect 7, showing it as certified for Red Hat, Slackware and OpenLinux, without mentioning any restrictions on using other versions of Linux. The current Corel port is WP 6, which is getting fairly old now. The Corel/SDCORP web pages say that WordPerfect 7 for Linux will be available in June (until a few days ago, it said May).

—Bob Nielsen nielsen@primenet.com

I was referring to the use of WordPerfect 6, the only one available at the time. All new Caldera licensing will allow use on other Linux platforms.

—Phil Hughes info@linuxjournal.com

Bliss

“The first virus able to infect a Linux system has been found by McAfee Associates. The virus, named Bliss, has spread to Linux systems, as many Linux users play Internet games while logged in as root.” [“From the Editor”, May 1997]

Have you done ANY independent research about Bliss, or do advertising dollars simply obligate you to propagate untruths which the marketing-driven McAffe Associates has spread about this virus?

Bliss cannot be spread by playing Internet games whiled logged in as root.

Check your facts.

—Kirk Haines khaines@oshconsulting.com

For the record, McAfee Associates has never advertised in our magazine. Thanks for letting us know that Bliss cannot be spread in this fashion.

Errors in July Issue

I just received *LJ* Issue 39 and have discovered some errors in the published article “Porting Scientific and Engineering Programs to Linux”, which I wrote with Charles Kelsey.

In the first line of the fifth paragraph, there is a reference to NCBO code. I don't know what NCBO means. In the manuscript that I submitted this line read “One thing that makes porting this code to a new platform somewhat challenging is that it is a safety related, pedigreed code.” Perhaps you intended for the final product to read, “One thing that makes porting the MCNP4a code to a new platform somewhat challenging is that it is a safety related, pedigreed code.”

Also, in Listing 1. FORTRAN Patch File, whoever retyped the patch file typed less than symbols (<) where there should have been commas (,). Humorous and quite understandable, since they're on the same key. Fortunately, the files offered for download at URL ftp://ftp.linuxjournal.com/pub/lj/listings/issue39/2177.tgz appear to be correct.

Thank you for your attention and for the opportunity to share with the Linux community.

—Gary Masters gmasters@devcg.denver.co.us

Please accept my apologies for these inadvertent changes in your text.

Correction to Correction

Regarding the Correction in the July issue, page 93, I would have thought that most Unix C programmers would know that the **main()** function in C is required to return an integer result. Thus, the correct first code line for Richard Sevenich's main() is:

```
int main()
```

not **void main()**. See Steve Summit's excellent comp.lang.c FAQ for a more in-depth explanation. There seems to be more than enough confusion about this

point propagated by the DOS/Windows crowd without *LJ* contributing to it as well.

—Matthew Saltzman mjs@clemson.edu

Inherent Limitations in FORTRAN Articles

In “Porting Scientific and Engineering Programs to Linux” [July 1997], the authors [Charles Kelsey and Gary Masters] write:

“The **f2c** has inherent limitations, leaving **g77** as the only viable alternative for compiling large, complex applications written in FORTRAN 77 under Linux.”

I was amazed to read such a blanket dismissal of f2c. At Berger Financial Research, we've been using f2c to compile FORTRAN since kernel .99pl8. We write volumes of FORTRAN and use standard libraries such as SLATEC, LAPACK, MINPACK and various routines from TOMS. The last time we had a problem with f2c was a couple of years ago, and the maintainer fixed it within a week of reporting it.

What exactly are the “inherent limitations” of f2c that the authors refer to?

—Dr. Harvey J. Stein abel@netvision.net.il

Java Articles?

Why don't you ever have articles on Java in the WWWsmith section? It seems all you ever cover is Perl and CGI. Is WWWsmith supposed to be about web programming or web mastering? If it's supposed to be about web programming, I think Java topics are very important (probably more than Perl, CGI and SQL). I mean, which gets more press coverage?

—Jeff Warren jwarren@sun.science.wayne.edu

Perl and CGI seem to predominate because Reuven Lerner, the author of the “At the Forge” columns, likes to write about them. We have four Java articles promised to us by various authors that have not yet come in. We also have an article about Java and the kernel that will be published soon, and Java was the focus of the October, 1996 issue of LJ, so it is not a subject we have ignored.

Possible Improvement to Magazine

I was trying to find a local store that sells Linux distribution CD-ROMS. I ended up buying a book with Red Hat 4.1, Slackware and Caldera's Lite version from Barnes & Noble ($59.99 US—good deal, comparatively speaking). Many popular

computer magazines package shareware or demo CD-ROMs with their magazines, so why couldn't *Linux Journal* print a special edition for newbies that would include a CD with a working distribution of Linux? If you released a quarterly edition for $10-$15 US, I would buy it just because I hate downloading updates. If you use my idea, send me a free subscription or something (assuming I'm not the 10,000th person to suggest this).

—Joshua Neal tinara@null.net

Well, you're certainly not the first to suggest it. It's on our list of “things we'd like to do sometime”.

XForms Article

I really appreciated the article “Programming with the XForms Library” [Thor Sigvaldason, July 1997]. I am looking forward to the next two segments of that series.

I am a Software Engineer with a day job developing exclusively for Windows 95 and NT. I am in the process of developing a large application in Java that will run on the NC, but that is the first project that I have worked on outside the realm of Microsoft platforms.

I have a personal interest in pursuing development in Linux. As far as I am concerned, you could dedicate 90% of the Journal to development and the other 10% to configuring the system. I realize that this probably doesn't line up with most of your readers' interests (and I am probably exaggerating a bit anyway), but I think there are a lot of Linux users, novice as well as experienced, who would appreciate seeing a little more space dedicated to development.

—Jeff Brown jbrown@fastrans.net

XForms Article

This is a response to an article by Thor Sigvaldason in Issue 39 of *LJ*, titled “Programming with the XForms Library”:

How can you encourage people to write programs using the XForms library? Since the sources aren't free, only a selected range of platforms are supported.

Furthermore, the most recent version of this library is (v0.81) available for Linux only in ELF format. What will those people do, who are using the a.out binary format for good reasons? Well, nothing, because the latest a.out library is

v0.80j, which is incompatible with v0.81. And the authors of the XForms library aren't going to support the a.out format in the future.

In my opinion, this is contrary to the spirit of those who developed Linux initially. It was meant to be a free Unix system based on free tools and a free kernel. (With maybe one exception, Motif, which is an established standard for nearly every available platform.) XForms doesn't fulfill this criterion.

—Thomas Ott Labalutsch@aol.com

Archive Index Issue Table of Contents

Advanced search

Internet Changes/Linux Changes

Phil Hughes

Issue #42, October 1997

This month the Internet took a big step toward maturity compliments of the U.S. Supreme Court.

Both Linux and the Internet are growing—in size and in maturity. This month the Internet took a big step toward maturity compliments of the U.S. Supreme Court. I'm talking about the Communications Decency Act.

At issue was an overhaul of telecommunications regulation that included a section restricting the distribution of indecent or “patently offensive” material to minors over the Internet. While this restriction may sound reasonable to people less involved in the Internet than your average Linux user, I think we all know that attempting to legislate control over something with no central control, such as the Internet, doesn't work. If there were central control, the Internet itself couldn't work.

In what seems to be a well informed decision, the court said that the Internet deserved full First Amendment protection, pointing out that it was unique as a public forum for the exchange of ideas and information. They rejected the argument that the Internet is similar to television and radio industries. The difference, obvious to Internet users, is that while information is pushed at you on television, you seek information on the Internet.

Justice Stevens said “The [Communications Decency Act] is a content-based regulation of speech,” and went on to say, “The vagueness of such a regulation raises special First Amendment concerns because of its obvious chilling effect on free speech”.

At the same time, Linux is maturing in a very good way. First, CD distributions for the PowerPC are finally starting to appear. In June, the Linux for PowerPC project announced another release that supports hardware from Be Inc, Apple

Computer, IBM, Motorola and most any other manufacturer of PowerPC computers.

The PowerPC is distinct from the MkLinux port for PowerMacs. MkLinux is based on the Mach microkernel. This project is based on a port of the standard Linux kernel. More information is available at http://www.linuxppc.org.

Another way Linux is showing its maturity is by addressing the usefulness of any computer system to people with disabilities. One organization that is involved in this effort is the Center for Disabled Student Services at the University of Utah. They offer a mailing list and pointers to pages that offer access information. You can find their web page at http://ssv1.union.utah.edu/linux-access/.

I wasn't surprised to find software addressing use of Linux systems by the blind including information on efforts to make documentation on Linux available text to speech and text to braille software. There is also a tips page on accessible web page design and even a list of hardware and software for the blind.

Here at *LJ* we have had inquiries over the years about printing the magazine in braille. This has never been practical because of the volumes involved. To address this we have made parts of the magazine available on-line and also have plans for a back issues CD. It is great to see that there are Linux-based options based on Linux for the blind to access this information.

On the Cover

Archive Index Issue Table of Contents

Advanced search

What Price High-Performance I/O?

Phil Hughes

Issue #42, October 1997

It was recently (meaning a few hours ago) called to my attention that there is an organization called the I2O Special Interest Group that is addressing this problem.

If you have been around the PC industry for a while, you most likely remember that the ISA architecture (AT bus as it was initially called) was recognized as not the best answer for high performance I/O, because of speed limitations and rather poor interrupt structure. Enter IBM with its MicroChannel architecture. All you had to do was pay IBM money, and you could use the design. Some manufacturers bought in, but it soon flopped because what the industry really wanted was an open standard. In fact, AT bus became ISA (Industry Standard Architecture) because of IBM's proprietary approach.

However, ISA still wasn't the answer. The first issue, bus speed, was addressed with the PCI bus which, being wider and faster, has satisfied the bandwidth requirements of I/O hungry systems. But, as the speed of everything has increased, so has the number of interrupts that must be dealt with.

The best solution is to have intelligent peripherals that don't have to interrupt the CPU as often in order to carry out their tasks. The most common example today is the buffered serial card UARTs (universal asynchronous receiver/transmitter). Another is intelligent serial cards that enable the transfer hundreds or thousands of characters in a single DMA (direct memory access) transfer and the take care of the character-by-character transfer themselves.

Each intelligent I/O card requires a driver for system communications. More accurately, a driver is needed for each operating system that wishes to talk to the card; thus, manufacturers must invest in support of each system. This investment means that most vendors tend to support only the most popular operating systems.

Enter A Solution?

It was recently (meaning a few hours ago) called to my attention that there is an organization called the I2O Special Interest Group that is addressing this problem. Here are a few quotes from their web page:

- The objective is to provide an open, standards-based approach ... and provide a framework for the rapid development of a new generation of portable, intelligent I/O solutions.
- The I2O model provides an ideal environment for creating drivers that are portable across multiple operating systems and host platforms.
- The I2O model is intended to provide a unifying approach to device driver design...

They also pose the question, "Do you see the Unix vendors supporting I2O in their future releases?" and answer it by stating, "SCO is a SIG member and has indicated it will support I2O in future releases of its OS. The SIG welcomes all other Unix vendors to join as well."

Is Everything Cool?

All of this rhetoric sounds like we are all friends, and we will all interoperate happily ever after. However, there does seem to be a catch.

In one of those **nice** answers about compatibility, we see our first clue that there is a potential problem: "The SIG is set up so that only members and their licensees can design with the specification,...." Even more to the point: "The I2O Specification...is an *agreement* about the intellectual content and the terms and conditions for how the Specification can be used. Therefore, to make the Specification available to non-members a non-disclosure agreement must be executed."

I have attempted to contact them for clarification but, so far, they have not returned either my phone call or e-mail.

I'm sure all Linux folks are familiar with the non-disclosure problem. Non-disclosure is why Diamond video boards weren't supported until Diamond changed their mind, and why Linux for the Mac didn't exist for years. To put it another way, you can't comply with both the GPL and a non-disclosure agreement.

What Can We Do?

Fight—not let someone who claims they are making an open standard get away with an "open to anyone except free software" standard. The first organization

to take action is Software in the Public Interest, the same folks who bring us Debian Linux. I have just received a draft of a proposal for an Open Hardware Certification Program. In this program, vendors will make a set of promises about the availability of documentation for programming the device-driver interface of the specific hardware device.

The idea here is that while the program will not guarantee a device driver is available for a specific device and operating system, it does guarantee that anyone who wants to write one can get the information necessary to do so.

I am sure there will be more on this topic on the Usenet newsgroups, on the web and in the press. If you are a vendor, contact SPI (http://www.debian.org/ will point you in the right direction) for more information on their certification program. If you are a potentially unhappy consumer, check out http://www.io2sig.com/ and let the SIG members know what you think about the exclusion of free software from their **open** standard and about SPI's effort for real open hardware. Finally, watch the *LJ* web pages for news on what is happening in this important battle.

Phil Hughes is the publisher of *Linux Journal*. He can be reached via e-mail at info@linuxjournal.com.

Archive Index Issue Table of Contents

Advanced search

DDD—Data Display Debugger

Shay Rojansky

Issue #42, October 1997

DDD provides a graphical user interface for the Unix debuggers, gdb and dbx. This article gives a good introduction to its many features.

One of the most common uses for Linux today is as a platform for program development. The rich suite of GNU software that comprises this platform is one of the most comprehensive development environments in existence. However, one universal quality of these tools has been both the reason for their success and their weakness when competing with other products—the character or command-line based interfaces.

On the other hand, a character-based interface requires that the user be familiar with its intricacies. New users are usually baffled because they don't know the magic commands and keystrokes. For this reason, many commercial programming suites provide an integrated windows-based environment. This almost always severely limits the programmer' use of tools—For instance, it may be impossible to use the tool in a script. Vendors consider these limitations a small sacrifice to make when the gain or loss of a new user is at stake.

The Unix environment has long lacked freely available, graphical, easy-to-use programming tools. The **ddd** command addresses this lack of graphical tools by providing a graphical user interface for the Unix debuggers **gdb** and **dbx**.

The debugger is an essential part of any programming environment, providing two basic capabilities:

1. Running a program interactively in order to observe the code execute and test for bugs
2. Examining the core dump of a crashed process

The standard Linux debugger is gdb, the GNU debugger. gdb provides an interactive text-base method for accomplishing those two tasks, including step-by-step execution, breakpoints, variable watch and other options that are expected in a full-fledged debugger. However, gdb is not easy to run—lots of experience is needed to fully operate gdb, and even then it tends to be somewhat awkward. To the rescue comes ddd—an X Windows front end to both the gdb debugger and the dbx debugger.

Installation

The ddd homepage is at http://www.cs.tu-bs.de/softech/ddd/, and the latest version (as of writing) is ddd 2.1.1. **ddd** requires Motif to execute; however, the latest versions apparently compile and run fine with the free Motif clone, Lesstif. Also, a statically-linked version of Motif is provided, although it is a very large binary that will hog memory. After compiling ddd (or downloading the binary), place the binary in a convenient location (/usr/local/bin, for example).

Outline

ddd is a front end—it doesn't do any debugging. Instead, it sends all user commands to an actively running gdb (or dbx) process. The ddd environment consists of four important windows:

1. The “Debugger Window” contains the actual communication between gdb and ddd. This window always displays the standard I/O of the debugged program.
2. The “Source Window” contains the program source and the basic source debugging actions.
3. The “Command Tool Window” contains buttons for most of the actions you'll need; stepping up and down stack, setting breakpoints, etc.
4. The “Data Window” contains all data-related information, such as variable and function watching.

To use ddd, write a program and compile it, using the **gcc** command with the **-g** switch set so that debugging information is included.

```
gcc hello.c -g -o hello
```

Then run ddd. Upon startup, only the debugger console opens. You must open the executable (or core dump) via the **File** menu; then, open the Command Tool Window and the Source Window. If you wish to manipulate data as well, open the Data Window via the **Window** menu.

At this point you can start running the program and diagnosing any problems. First, set a breakpoint, a line of code at which the debugger should stop running the program and wait for instructions. Click on a code line and then set the breakpoint by clicking the **Break at()** button; a “red stop sign” appears next to the line to mark the breakpoint. You can clear it at any time by clicking on the line and pressing the **Clear at()** button.

Start running the program by clicking the **Run** button in the Command Tool Window. The program executes, just as if you had started it at the prompt, but stops immediately on reaching the first breakpoint. Now you can continue program execution line-by-line in one of two ways—**Step** and **Next**. **Step** executes the line of code, and if that line is a function for which the code exists, **Step** enters the function code and steps through it. **Next** executes the function and proceeds to the next line. Note that a “green arrow” marks the line that is to be executed. After moving through the program code using **Step** and **Next** for a while, you can press the **Continue** button to continue running the program until the next breakpoint is reached or until program end, if no more breakpoints are set.

Executing portions of code and playing around with breakpoints can be nice, but there's substantially more to debugging. A vital part of debugging is observing how the program manipulates data. Since ddd is a graphical application, it can contribute most where data visualization is used.

Begin debugging a program that manipulates variables by opening the Data Window. During the debugging process, select a variable by clicking the left mouse button on it and then press the **Display()** button; the variable and its value will appear in the Data Window. Executing any line of code that modifies that particular variable changes the representation in the Data Window. You can, of course, create more than one variable “watch” and even position them in the window as you like. You can create a function watch by clicking the right mouse button in the Data Window and then choosing **New Display**. With this option you can enter any valid C expression and have it evaluated.

Variable and function visualization is not unfamiliar to gdb users. However, one of the most amazing things in ddd is visualizing pointers; for example, create an integer pointer and have it point to an integer. Create a watch on the pointer. In the content section of the watch, you will see the memory address of the integer pointed at. Click on the content and then click the **Dereference()** button.

At this point a new cell is created, containing the integer variable, with an arrow extending from the pointer to the integer. Taking this concept a little further means that you can graphically visualize complex data structures, structures, arrays and almost anything you wish. For example, Figure 1 displays the source code for a linked list, while Figure 2 displays the ddd visualization of it.

Figure 1. Source Window

Figure 2. Data Window

Another worthy feature of ddd is its help system—a very useful context-sensitive mechanism, help can be tapped by pressing f1 and then clicking anything in ddd. Additionally, a more complete and methodical manual can be opened from the Help menu, offering menu-based text pages with examples.

Conclusion

ddd has more features than I have covered here; however, this article should give you a good start. It is refreshing to see Unix enter the modern era with graphical front ends which do not limit the user. When I first found ddd, I was quite impressed at its potential for novice Unix programmers. However, I also find myself firing it up quite often, even though I am a fully "qualified" Unix hacker with many gdb scars. The simplicity and elegance of ddd will make even the most die-hard shell fanatic roll over and accept it in many situations.

Shay Rojansky is an 18-year-old high school student about to be drafted into the Israeli Defence Forces (IDF), where he hopes to push Linux as an OS. He sometimes works in his high school as a system administrator (mainly Linux). He can be reached via e-mail at roji@cs.haji.ac.il.

Archive Index Issue Table of Contents

Advanced search

cat

Patrick Hill

Issue #42, October 1997

Here's a spiffy little command that can be used to combine files, look at the contents of a file and do limited text editing.

The Linux **cat** command at first seems so simple as to be unnecessary. In actuality, it is an excellent example of the Unix philosophy: create programs which do one thing and do it well. The thing **cat** does well is display the contents of a file or files. Many other utilities can handle this task, but none have all the options **cat** does. First, let's look at the simplest case:

```
cat /etc/motd
```

This command will display the contents of the motd (messages of the day) file to your screen. Unlike **more** (or **less**), **cat** will not stop when the screen is full. This is a feature, not a bug. You don't want pauses when redirecting (using the > operator) cat's output to a device, e.g., a printer or modem:

```
cat /etc/motd > /dev/modem
```

cat comes from the word conCATenate, which describes one of its best uses: to concatenate or “glue together” two or more files. If you have several individual files about animals you would like to collect together into one file, **cat** will do the work for you. For example:

```
cat tiger lion cougar > bigcats
```

would redirect the concatenation output, containing the three feline files in the specified order, into a new file named **bigcats**. If you find another file, **panther**, that needs to be added to the **bigcats** file, use **cat** with the append (**>>**) operator in the following way:

```
cat panther >> bigcats
```

Using **>>** ensures any prior content of **bigcats** is preserved. The content of **panther** is appended to **bigcats**. If you were to use the **>** operator here, you

would replace the contents of **bigcats** with the contents of **panther**. Always use **>>** when you wish to add to the end of an existing file.

Be careful not to use the same file name when redirecting the output of the **cat** command, or you could lose one of the files. For example, don't do the following:

```
cat myfile yourfile > yourfile
```

In this case **yourfile** gets overwritten by **myfile**.

Another surprisingly handy use for **cat** is to redirect standard input like this:

```
cat > newfile
some notes I want to save in newfile.
CTRL-D
```

This creates a new file (named **newfile**). You type as much text as you wish, then type **ctrl-D** to save the file. You can backspace over mistakes, but you cannot go to a previous line after you press the enter key. I often teach this particular option of **cat** to novice Unix users, who occasionally need to create simple files, but don't want to learn vi or other simple editors. There may be Unix systems without vi or your favorite editor, but **cat** is always there.

The operator **>>** can also be used to append notes to the end of **newfile**:

```
cat >> newfile
Adding another note to newfile.
CTRL-D
```

cat Switches

Like most Unix commands, the behavior of **cat** can be modified by command line switches. If you use the **diff** command to compare files, it will show you the numbers of lines that differ between the files. However, most files don't have line numbers. Use **cat** with the **-n** switch to number each line of a file:

```
cat -n kittens > num_kittens
```

The file **num_kittens** is **kittens** with a number in front of each line, including blank lines. Use the **-b** switch to number only lines that are *not* blank.

One last **cat** trick: using the **-v** switch will show you “hidden” characters, such as control characters that may not show up in your editor. Try this experiment:

```
cat > catestv
CTRL-v testing CTRL-O Testing esc-b
CTRL-D
```

If we use **cat** to view the file, we see only the normal text:

```
cat catestv
Testing Testing
```

To see what's actually in the file, use the **-v** option:

```
cat -v catestv
^V Testing ^O Testing ^[b
```

Here, a **^** in front of a character signifies a control character. (**CTRL-[** is the same as ESC).

Patrick Hill (apathos@bham.net) is a computer engineer at Alabama Power Company in Birmingham, Alabama. He is known around the office as the guy who uses **cat** for an editor.

Archive Index Issue Table of Contents

Advanced search

Grundig TV-Communications

Ted Kenney

Issue #42, October 1997

Linux servers and applications are being used in Denmark to provide an interactive teletext system for TV viewers and to offer phones and movies to hospital patients.

In a modest building outside Copenhagen, Grundig TV-Communications engineers Niels Svennekjaer and Peter Hansen tend to the information needs of Danish hospital patients and TV viewers, monitoring and making fixes to Linux-based applications running on PCs. In the basement, a Linux server hosts an interactive teletext system. Television viewers phone this system and navigate a Raima database of classified ads, horoscopes, job listings and other information, entering their choices through the telephone keypad and receiving text responses on their TV screens. Already, Denmark's TV2 nation-wide broadcaster and several local stations have adopted this technology; the government unemployment bureau recently made it the official channel for job listings.

In a hardware-cluttered office upstairs, Svennekjaer phones in via modem and fine-tunes a Linux server at a local hospital. This application makes amenities such as phones and pay-per-view movies available in Denmark's highly efficient (yet somewhat spartan) hospital system and has already been sold to five hospitals in Scandinavia. Grundig TV-Communications' vision includes taking this system into Poland and the Baltic nations.

Is the quintessential hacker's operating system making inroads at one of the Europe's largest electronics manufacturers? At Grundig TV-Communications, a Danish outpost of the large German firm Grundig AG, Linux has become the standard platform for information-on-demand applications. Grundig TV-Communications likes the freely distributable operating system's low cost, robustness and the availability of tools. They are also impressed by its support—not from a raft of engineers on call at a vendor's headquarters, but from the multitude of other Linux users who can be reached via the Internet.

Grundig has chosen Linux for its new hospital and broadcasting systems and will likely move to Linux with its core product, a hotel teletext system serving some 200,000 rooms in Europe and Asia. It has also standardized on Raima's embedded database as the data management component in its applications. Both the hospital and broadcasting applications use Raima Database Manager++, a low-level database tailored to the needs of C and C++ programmers. In the next step of their evolution, these products will move to a client/server architecture with Raima's Velocis Database Server. Svennekjaer believes this redesign will better support the heavier load of concurrent users and more complex queries the applications are beginning to experience.

Teletext is largely unknown in the United States, but it is widely used in Europe, where viewers obtain sports scores, news updates and other text by tuning into special channels. This service is supported by a band of the broadcast spectrum that governments set aside for teletext and other special purposes, such as remote equipment maintenance.

Grundig TV-Communications goes well beyond these usually passive teletext services by adding a dimension of user interaction. When viewers call into the server, a personal “page” appears on their screen. From that page they begin navigating the Raima database. A user might first request all real estate listings, then “drill down” to two bedroom, waterfront holiday homes on the Jutland peninsula. At TV2 the resulting data streams are sent from a Pentium server PC to one of 16 inserter PCs placed in TV station broadcast towers at different locations in the country, where they are mixed with the broadcast signal.

The revenue from this application comes from content providers, who pay for their data to be accessible in Grundig's Linux server. Television advertisers frequently provide teletext codes for obtaining more information, and the server's peak loads occur when commercials are particularly successful in drawing viewers. The server and database respond to requests for some 200,000 pages of information per day.

For the hospital application, Grundig TV-Communications provides tape-deck-sized panels which sit next to the patient's bed. When they check in, patients receive “smart cards” to insert in these panels to activate their personal television, telephone and radio accounts. Listening to the radio is free, but patients pay by the minute for phone and TV and per-movie for videos. Most are happy to pay. Danish medicine is of high quality, but the government-run system provides few amenities—nurses wheeling trolleys of shared phones are still a common sight in many hospitals, according to Svennekjaer.

A Linux-based application controls access to these services and monitors patient usage and billing. Grundig shares the revenues from this application

with two partner companies which handle marketing, installation and billing. So far, hospitals earn no money from the application, but they benefit from having happier patients.

The multi-user, multi-tasking demands of these applications made Unix a natural choice of operating systems, Svennekjaer says. Interactive Unix, their first platform, was produced by an independent vendor, sold to Kodak, then sold to Sun. The Grundig team felt this platform's future was uncertain and decided to switch. For cost reasons, they wanted to use PCs, so they evaluated SCO and Solaris. About the same time, Svennekjaer says, some of his friends were “playing around” with Linux version 0.99 at the Aalborg University Center in Denmark. They recommended that he try this unknown (to Svennekjaer) Unix. “We ported our application to Linux in one week, and it worked. And it was free!” Svennekjaer recalls.

But did the freely distributable operating system make sense for commercial applications targeted at hundreds of deployment sites? When the Grundig TV-Communications team weighed the risks and rewards, Linux looked better than the alternatives. First, there was the price tag of approximately $2,000 per site for a commercial release of Unix versus $20 apiece for CD-ROMs with the complete Linux system plus ample tools, add-ons and enhancements.

And what about the frequently-voiced concern that Linux is an “unsupported” operating system? The idea of support, it turns out, is relative for developers working far away from the OS vendors—or perhaps for any developers making especially complex system demands. Svennekjaer notes that in the past he relied on local distributors of commercial software products for help. When his dealer couldn't help him with a stack overflow, the $2,000 he spent on a supposedly “supported” product started looking like money wasted.

In contrast, with Linux “we get the source code, so we can address problems directly when they come up,” he says. In addition, when particularly difficult issues arise, Svennekjaer posts his questions to one of the many Linux newsgroups on the Internet. “Usually the next day we have five answers” from Linux programmers around the world, many of whom are intimately familiar with the source code.

For example, Svennekjaer uses a RAID with four hotplug/hotswap disks as backup for the teletext application, in case its hard drive fails. Distributed Processing Technologies (DPT), the manufacturer of the SCSI disk-controller for the RAID, did not provide a Linux driver at that time, but one was included with the kernel source of Svennekjaer's Linux distribution. However, after moving to an updated version of the disk-controller, Svennekjaer experienced problems with the old driver. So he tracked down the driver's author, Michael Neuffer,

from an e-mail address listed in the driver source, obtained an updated version and was back on track within days. (Since that time, DPT has begun distributing a Linux driver developed by a third party.)

According to Svennekjaer, the problem with Linux tools is not lack of availability, it's that there are so many that it's distracting—CAD/CAM programs, ray-tracing software, biomolecular analysis programs, diagram editors, multi-user games, web servers and other interesting free or demo packages announced in comp.os.linux.announce.

While Svennekjaer downloads many of his Linux-based tools for free, his firm has opted for commercial database technology, first with Raima Database Manager++ and later with Velocis Database Server, an SQL client/server DBMS that evolved from the RDM++ technology. Velocis and RDM++ are unique in that they support database architectures using the relational database model, network database model and combinations of both models. This flexibility can be useful. Relational “keyed” data access is typically better for finding randomly selected records, while the network model is faster when retrieving related records or drilling down through hierarchical data, as is often the case for users of the teletext application.

The current applications based on RDM++ enable multiple processes to access the database. The move to client/server is intended to support even heavier usage, as both the hospital and teletext systems scale up to serve thousands of users, and as these users generate more complex queries.

Client/server architecture will centralize database processing on the server, thus reducing network traffic. It will also improve concurrency, the ability of multiple processes to share data. In databases, concurrency is regulated by locking whereby a certain portion of the database is made off limits to other processes while a single process accesses it. Like most file-server architecture database management systems, Raima Database Manager++ provides file-level or table-level locking—a data file or database table is the typical unit that is locked. Velocis, on the other hand, provides row-level locking, in which a much smaller unit (data files or tables are made up of many rows) is made unavailable to other processes. It also includes features for protecting the integrity of transactions, such as “roll-forward recovery”, in which related changes to the database are completed only as a unit, even if the system goes down during a transaction.

In the future, the teletext application will offer more demanding services, like providing a calculator so a user can figure out the monthly payments on a car with a specific price, interest rate and down payment. Grundig TV-Communications also wants to let users customize their queries. In the past, the Grundig team has “hard-wired” its calls to the database, coding them in C function calls that are embedded in the application. However, the introduction

of Velocis will add SQL capabilities to the system. SQL is a high-level database query language that can be used to give end-users the ability to define their own queries.

What's It?

For example, a user might want to view listings for all cars with air conditioning in a particular price range and within a particular geographic area. As more possible variables exist for a search, much more complex queries can be defined. Offering SQL will let users search using their own criteria, eliminating the Grundig TV Communications' engineers need to define all queries in advance. It's one more step toward making information truly available on demand.

> Grundig TV-Communications Engineer Niels Svennekjaer can be contacted at nrs@gtv.dk. Lars Teil, who manages the Linux teletext system at TV2, can be reached at late@tv2.dk.

Ted Kenney is Marketing Manager for Raima Corporation in Seattle. He visited Grundig-TV Communications in March, 1997.

Archive Index Issue Table of Contents

Advanced search

New Products

Amy Kukuk

Issue #42, October 1997

Khoros Pro 2.2, Kai C++ Version 3.2, Open Sound System v3.8 and more.

Khoros Pro 2.2

Khoros Pro 2.2 from Khoral Research, Inc. is now available. Functional advances introduced with Khoros Pro 2.2 are a new widget set and new geometry services with APIs. These programs provide data format and operating system independence and the ability to process large data sets. Programs under Khoros Pro provide automatic data format conversion, data type conversion, file format conversion and support of large data sets and multidimensional data. The price for a single user is $225 US.

Contact: Khoral Research, Inc., 6001 Indian School Rd NE, Suite 200, Albuquerque, NM 87110-4139, Phone: 505-837-6500, Fax: 505-881-3842, E-mail: info@khoral.com, URL: http://www.khoral.com/.

Kai C++ Version 3.2

Kuck & Associates, Inc. (KAI) announced the immediate availability of Version 3.2 of the KAI C++ compiler on Linux for Intel x86. This release of KAI C++ includes C++ class libraries, support for member templates, improved debugging and new usability features for large codes. Some new features and improvements include support for member templates, support for partial specialization and optimization of template expressions. Contact the company for price lists.

Contact: Kuck and Associates, 1906 Fox Drive, Champaign, IL 61820, Phone: 217-356-2288, Fax: 217-356-5199, URL: http://www.kai.com/.

Open Sound System v3.8

Open Sound System v3.8 from 4Front Technologies provides sound card drivers for most popular sound cards under Linux. The drivers support automatic sound card detection, Plug-n-Play and Java Audio. 4Front also introduced SoftOSS technology for Linux which provides a software based wave-table engine that doesn't require any wave-table hardware. OSS/Linux sound drivers comply with the Open Sound System API spec. The price for a driver is $20 US.

Contact: 4Front Technologies, 11698 Montana Avenue, Suite 12, Los Angeles, CA 90049, Phone: 310-820-7365, Fax: 310-826-2465, E-Mail: info@4front-tech.com, URL: http://www.4front-tech.com/.

ARKEIA

Knox Software released ARKEIA, a network backup program. It is built around an architecture with three separate modules which include server, client and graphics. ARKEIA provides server backup, network security and data integrity. ARKEIA interfaces with all new devices and all types of servers. Contact the company for pricing information.

Contact: Knox Software, 1325 Howard Avenue Suite 328, Burlingame, CA 94010, E-Mail: info@knox-software.com, URL: http://www.know-software.com/.

Mathematica 3.0

Wolfram Research, Inc. announced the availability of Mathematica 3.0. Mathematica 3.0 offers features such as edit-able typeset mathematical expressions, customizable palette interface and new algebraic computation and simplification as well as other new functions.

Contact: Wolfram Research, Inc., 100 Trade Center Drive, Champaign, IL 61820-7273, Phone: 217-398-0700, Fax: 217-398-0747, E-Mail: info@wolfram.com, URL: http://www.wolfram.com/.

PerfectBACKUP+

Unisource Systems, Inc. released Perfect Backup+. Perfect Backup+ is a backup and restore program. A few new features include backup and restore from devices connected to remote hosts on a TCP/IP network, a real-time monitor which shows the percentage complete and real-time elapsed while the backup is in progress and a new menu option to replicate or convert media types at the device's maximum throughput rate while the system is in multi-user mode. The price for PerfectBACKUP+ starts at $99 US.

Contact: Unisource Systems, Inc., 1409 North Cove Blvd., Longwood, FL 32750, Phone: 407-834-1973, Fax: 407-834-1973, E-Mail: sales@unisrc.com, URL: http://home.XL.CA/perfectBackup/.

Empress RDBMS 6.10

Empress Software released Empress RDBMS 6.0 with enhanced BLOB handling performance for multimedia data. The product is available as a developer's toolkit which also includes an HTML toolkit, ODBC Server and Client and Dynamic SQL. Pricing starts at $1,000 US for two users.

Contact: Empress Software, 6401 Golden Triangle Drive, Greenbelt, MD, 20770, Phone: 301-220-1919, Fax: 301-220-1997, E-Mail: info@empress.com, URL: http://www.empress.com/.

Archive Index Issue Table of Contents

Advanced search

Pgfs: The PostGres File System

Brian Bartholomew

Issue #42, October 1997

The details of how Pgfs came to be written and how it can save you disk space.

The PostGres file system is a Linux NFS server that presents software versions as distinct file trees in an NFS file system. Each version is completely distinct from all other versions, and can be modified independently without regard to versions before or after it. Each version retains all of the properties it would have on a normal file system, such as file ownership, permissions, binary file contents, cross-directory hard links and non-files such as devices and symlinks. The effect is the same as if each version of the software had its own separate directory, except far less disk space is used.

Design Motivation for Pgfs

As an example, let's say that a year ago, you picked your favorite Linux distribution, and installed it on your new computer. The distribution had about 15,000 files, and took up about 200MB on disk. During the year you did a lot of hacking on your software, and it is now quite different from the original distribution. These modifications were done incrementally, and some of them replaced the original binaries with completely new binaries, for instance upgrading sendmail(8) or ftpd(8). Now you wish to compare your machine to the original distribution, examine the changes you've made, and apply them to a new distribution of Linux.

How do you record the changes you've made? You could save a complete copy of your distribution every time after you modified it—doing that would consume 200MB * 100 mods = 20 GB of disk. Even using a pair of 9 GB drives that's awfully expensive. However, you notice that most modifications change only a few files—perhaps a total of a half MB per modification. Storing only the files that changed would use 200MB + (100 mods * 0.5MB/mod) = 250MB of disk space—that's much better. What application would store only the differences for you? You could use CVS, but CVS isn't really suitable.

Now let's say you are a systems administrator, and you've faced this version control problem daily for years without finding a satisfactory solution. So since you're also a developer, you decide to build an application to store similar file trees that exploits the compression opportunities you've found. Fundamentally, this application would need to eat file trees and spit them back out again, and it must use less disk than keeping a whole copy of each tree. It should accept files one at a time or in bulk. You shouldn't have to extract and resubmit a whole file tree just to make one change.

How would you implement this application? Start by deciding what data structures are needed and what routines are needed to manipulate the structures. Let's start with files. Files consist of two parts—a stat(2) structure and a big chunk of binary data. Suppose you store the binary data in a file and name this file with a number. Then, name the stat structures with a number and store the fixed-length structures in an array on disk. The structures can be broken apart using field-splitting routines and assembled using record-making routines. Next, a structure is needed to represent the different versions of your software. Each version of your operating system consists of a tree of files. You name a tree of files that represent one specific software version a “version set” or “verset” and number them.

The next thing needed is some routines to search the stat array on disk for a specific structure, add structures and delete them. Since you will be doing random access to the structures, store them in a dbm (database management) file and use the dbm access routines instead of writing your own. Dbm also gives you routines to handle an index into the stat structure numbers, in order to make your access faster. You will need to write maintenance routines to copy dbm files and to copy fields from one dbm file to another with a different structure layout.

To add a new stat structure into your array may require modifying fields in structures other than the one you're adding; for instance, when you add one file to an existing file tree. Your programming task would be a lot simpler if you could collect a bunch of these modifications and do them all at once, or not do any of them if you discover a problem. The idea of doing all or none of a complex modification is called “transactions”.

To use your application you need commands for it to accept. Some commands might be “add this whole tree of files”, “add this single file” and “replace some bytes of a file with these bytes”. The NFS people have figured out the minimal set of file operations you need. (See Sidebar 1.) Now decide how stat(2) structures will be modified for each of the file operations and write pseudo code to modify the stat(2) array. While designing the semantics of the NFS

commands, start thinking about sending your application NFS commands, making it an NFS server.

Next write your application. How about using an SQL database? A database decouples the application data structures from their representation on disk giving you the following advantages:

1. Structures can be defined with arbitrary fields and stored in tables.
2. A full set of routines to add, delete and modify structures is available, as well as indices to find structures quickly.
3. A nice command language is available to translate between structure formats as the application evolves.
4. Routines in the database are designed to operate on chunks of data that won't fit in memory all at once, so your application can grow without problems.

To add a field called “cokecans” to tally the number of cans of Coke it took to create each file, just add it. You can transfer your existing data to a new table with the cokecans field in a couple lines of SQL. Compare this to coding in C where a bunch of custom binary-format conversion programs would have to be written.

Then, find the skeleton of a user-level NFS server and port it to Linux (see Sidebar 2), and hook up the source of NFS commands to the command input of your application. Now you have an NFS server that presents file trees, but compresses away the similarities between trees. Since your application can be used like any file system, you don't have to build any specialized programs to manipulate versions—you can search files with **grep** and compare file trees with **diff**.

To control your application, create some fake magic filenames that it can treat as special, like **procfs**. The lines that are written to these files are the commands to your application. Now your application can be controlled with **echo(1)** commands from the shell rather than some obscure socket protocol.

The above description is not exactly how I went about writing Pgfs, but it does outline the design motivation. After I tried to store copies of a BSDI distribution under CVS and failed in practical terms, I set out to write an NFS server implemented on a database. My first version was coded in Perl5 using the PostGres client library, and I typed in NFS commands as space-separated text strings. I recoded in C to pick up the NFS RPCs. My first database design schema used one table for “names”--holding filenames and symlinks, and another table for “inodes”--holding the rest of the stat(2) structure and the pointer to the file contents. However, I didn't like the join operation (i.e., matching up rows from

two tables with the same key), and I didn't want to implement join either in the database or the application code.

How Pgfs Works for the User

Let's use your favorite Linux distribution as an example of a file tree that needs version control. To start out, copy the virgin operating system from the CD-ROM into Pgfs:

```
cp -va /cdrom /pgfs/1/1
```

Let's examine that destination pathname. The **pgfs** part is where Pgfs is mounted. The first **1** is the "module". Storing software that evolves independently in different modules saves disk space. The second **1** is the verset. We have a brand new empty Pgfs, so we'll write into verset 1. Once the copy is done, use **ls** to see what's in the **pgfs** directory:

```
ls -l /pgfs/1/1/bin/su /pgfs/1/1/dev/cua0
```

The output from **ls** looks like this:

```
-rwsr-xr-x  1 root   bin     9853 Aug 14 1995 /pgfs/1/1/bin/su
crw-rw----  1 root   uucp    5, 64 Jul 17 1994 /pgfs/1/1/dev/cua0
```

Notice the **suid** bit on **su(1)** and leading **c** on the **cua0** mode. Pgfs stores attributes and non-files just like any other file system. This copy of **su** will make you root if you picked the mount option to accept **suid** bits when you mounted Pgfs. Next, copy verset 1 to a new verset, so that the new verset can be modified without changing the files in the old one:

```
echo "cpverset 1" > /pgfs/ctl
```

In your new verset, you install a newer version of sendmail:

```
cp /tmp/sendmail /pgfs/1/2/usr/sbin/sendmail
chown root.bin /pgfs/1/2/usr/sbin/sendmail
chmod 6555 /pgfs/1/2/usr/sbin/sendmail
```

Now that you have two different versets, you can compare their contents. You access multiple versets with shell wild cards or other filename expansions. To find what versets there are, do **ls /pgfs/1**.

```
strings - /pgfs/1/1/usr/sbin/sendmail | \
        grep version.c
@(#)version.c  8.6.12.1 (Berkeley) 3/28/95
strings - /pgfs/1/{1,2}/usr/sbin/sendmail | \
        grep version.c
@(#)version.c  8.6.12.1 (Berkeley) 3/28/95
@(#)version.c  8.8.2.1 (Berkeley) 10/18/96
strings - /pgfs/1/*/usr/sbin/sendmail | \
        grep version.c
@(#)version.c  8.6.12.1 (Berkeley) 3/28/95
@(#)version.c  8.8.2.1 (Berkeley) 10/18/96
```

Visualizing Software Evolution

Most version control packages focus on the individual modification history of single files, and that's what their tools display. I think the idea of the set of files known as “customer release 1.0” is more important than the idea of how each file happened to arrive in that set.

Suppose a new employee comes across a Pgfs containing 200 versets. One of the first things she wants to know is what each verset represents and how they interrelate. Why is this verset here? Where did this verset come from? Which versets represent consistent software releases? Tools with the file as the basic unit would ask her to compare file histories at this point. Too bad there's no way to coherently display 40,000 individual file history trees when she's comparing two versions of /usr. Tools based around the verset scale work better, because there are a lot fewer versets than there are files per verset.

I wanted a program that reads an entire Pgfs database and plots the relationship of each verset to each other, in terms of quantity of shared files and unique files. Run against Pgfs, the program shows that verset 1 and 2 have 19,998 identical files and 2 different ones, and the different ones are /usr/foo and /usr/bar. The program plots boxes for 200 different versions of /usr, with connecting lines that vary in color and width depending on the percentage of shared files and the percentage of older and newer files. If I told the employee in words that two copies of /usr were “almost identical”, “quite a bit different”, or “from two different operating systems”, she would have a good idea of the approximate numbers I meant. In my program I want those pigeonholes to be visually obvious from the pictures that compare versets.

Access Transparency

For most system administration purposes I don't care how or why files changed. If I apply a vendor patch to a kernel, all I care about is getting the kernel tree back before and after the patch. I don't want to reverse- engineer the patch script into file adds, deletes, renames and modifies just to shove it into Pgfs. I shouldn't need to notify the version control system which files to view or modify with checkin/checkout commands. I just want an NFS file system, and whatever I have in the directory when I leave is stored in the verset for next time. Since I'm not going to be giving Pgfs hints about what I'm doing, every operation needs to be possible. Therefore, each verset must be totally independent from all the others. I don't want to be forced to evolve my files from previous files in a branch structure without loops, or to keep my filenames constant between versions because of the lack of directory versioning, to name two well-known limitations of CVS.

Pgfs Architecture:

Here's a description of the real Pgfs program that you can download. Pgfs is a normal user-level program that reads and writes ordinary TCP streams and UDP packets. Since it is a normal program that requires no privileges, it can run on any Linux system. It doesn't use any ground breaking system call features, so no kernel modifications are necessary. The TCP stream packets are generated by the PostGres client library, so Pgfs can interact with a PostGres database using SQL. The UDP packets are formatted by the conventions of the NFS protocol. All this means is that an NFS client such as a Linux kernel can choose to send NFS packets Pgfs' way, and can mount a file system as if Pgfs were any other variety of NFS server. The AMD automounter is another example of a user-level program that acts as an NFS server. AMD responds to the directory-browsing NFS operations that trigger an automounter response, whereas Pgfs responds to all NFS operations.

In essence, Pgfs is an NFS <-> SQL translator. When an NFS request comes in, the C code submits SQL to get the stat(2) structures for the directory and file mentioned in the request, doing error and permission checking as it goes along. First it compares the request with the data it gets back about the file, enforcing conditions, such as whether **rmdir** can or can't be used to delete a file.

If the request is valid and the permissions allow it, the C code finds all the stat(2) structures that must be changed, such as the current file, the current directory, the directory above and hard links that share the file's inode. Then these modifications are made in the database by SQL. The modifications include side effects like updating the access time that you might not ordinarily think of.

Each NFS operation is processed within a database transaction. If an “expected” error occurs that could be caused by bad user input on the NFS client, such as typing **rmdir** to delete a file, an NFS error is returned. If an “unexpected” error occurs, such as the database not responding or a file handle not found, the transaction is aborted in a way that will not pollute the file system with bad data.

Pgfs does all the things “by hand” that go on in a “real” file system. It uses PostGres as a storage device that it accesses by inode number, pathname and verset number. For an example, the **nfs_getattr** NFS operation works like the **lstat(2)** system call. **getattr** takes a file identifier, in this case an NFS handle instead of a pathname, and returns all the fields of a stat(2) structure. When Pgfs processes an **nfs_getattr** operation, the following things happen:

1. The NFS packet is broken apart into operation and arguments.

2. NFS operations counters are incremented.
3. The NFS handle is broken into fields.
4. Bounds-checking is done on the **nfs_getattr** parameters.
5. **stat(2)** information is gotten for handle, e.g., select * from tree where handle = 20934
6. Permissions are checked.
7. File access times are updated, e.g., update tree set atime = 843357663 where inode = 8923
8. NFS reply is constructed.
9. Reply is sent to NFS client

Storage Schema

The single table that holds all the stat(2) structures has fields defined as shown in Table 1.

Inode numbers are unique across the entire database, even for identical files in different versets. Each file in each verset has one database row. Each directory has three rows; one for it's name from the directory above, one for . (dot), and one for .. (dot dot) from the directory below.

Philosophically, compression of similar file trees is the business of the back end of a program—it should not be visible to the user. In Pgfs, each collection of file bytes is contained in a Unix file, shared copy-on- write across all the versets from which the filename was inherited. Whenever a shared file is modified, a private copy is made for that verset. This matches Pgfs' system administration orientation, where files will be large and binary and replaced in total, and the old and new binaries won't be similar enough to make differences small. This differs from source code, where the same files get incrementally modified over and over and differences are small. With the keep-whole-files policy, doing a **grep** on files in multiple versets won't be slower than staying within a single verset. There is not a big delay while a compression algorithm unpacks intermediate versions into a temporary area.

Audience Participation Time

So far I've identified versets only by integers, but integers are boring. Since Pgfs is built on top of a database, all manner of things are possible. What naming schemes have you come up with for versets in other projects to support a configuration management effort? Are all the names/identifiers in a flat space? Are they hierarchical, and do they inherit properties from the location they are placed in? Let me know, I'm particularly interested in experiences from large projects with tens and hundreds of thousands of files per verset.

So far the only method I've supplied for making a new verset is by sending Pgfs a special command to copy one verset into a new one. However, a verset is just a collection of database rows, so it can be manufactured by a SQL program, perhaps one that represents a semi-automated multi-way merge across 14 versets. Take the base file tree from here, take this patch there, take this other patch over yonder, and make all the daemons owned by fred, thank you very much. What would an interface to control this process look like? Would you have an interactive file-browser shopping-cart thing, where you pulled bits and pieces from wherever you find them? How would this process resolve collisions?

There are more interesting open-ended questions in the BUGS and TODO files of the Pgfs distribution that concern both interface and implementation. I encourage you to pick a couple that interest you and talk about them on the host-gen mailing list.

Closing Thoughts

The most important message I want to give you is that file system hacking is not just for wizards anymore. NFS supplies a portable file system interface that eliminates the usual kernel-hacking requirements. NFS semantics are not great, but they are adequate for many things. Anyone can use the NFS-decoding portion of Pgfs as a skeleton and write a file system with whatever semantics they dream up. Mundane possibilities are an ftp-browsing or web-browsing file system. More interesting areas involve wide-area, fault-tolerant file systems with distributed physical redundancy. The job of higher-level protocols is to turn failure into bad performance. Instead of a list of Linux ftp sites to pick through by hand, wouldn't you rather use a file system that automatically gets blocks from the best-performing site of the moment? Wouldn't you like to create a new local storage area for a subtree of your favorite archive site, and the only thing your users need to know is...access just got faster? These ideas raise lots of interesting authentication and trust issues, many of which can be solved by the PGP model of the web-of-trust. Now, go forth and code.

Sidebar 3

Brian Bartholomew is writing Pgfs as a component of the Host Factory automated host maintenance system. Host Factory integrates hosts into a Borg collective. Working Version does large site toolsmithing, and further info on Host Factory is available at http://www.wv.com. Brian can be reached at bb@wv.com.

Archive Index Issue Table of Contents

Advanced search

Kernel-Level Exception Handling

Joerg Pommnitz

Issue #42, October 1997

An in-depth look at how the kernel is now confirming the validity of memory access addresses—not for the beginner. The information in this article applies to kernel versions 2.1.08 to 2.1.36. With version 2.1.37, a few of the kernel internals changed. The exception handling did not change, but the compiled code would be slightly different from the included examples.

A Linux system consists of the kernel and an assorted collection of user applications. The user applications communicate with the kernel through system calls. System calls are entry points into the Linux kernel which allow the application to use services provided by the kernel. These services are often executed on input data provided by the user application. These input data can, on occasion, constitute a problem.

Linux is a multiuser operating system; i.e., it supports multiple concurrent users. It doesn't know anything about these users or the programs they use. An erroneous or malicious application might pass an invalid argument to a system call. If the kernel then tries to use the invalid argument, unpredictable behavior can occur. For that reason, the kernel must be “paranoid” about the arguments of system calls and must carefully check each argument to ensure that it neither endangers system stability nor harms other users. This is easy for some classes of arguments; file descriptors must be open, **ioctl** commands must be allowed for the specified device, etc.

One class of arguments contains pointers to the location of input data in the address space of the application or to a buffer that is going to receive the data of the system call. As an example, the system call **write** comes to mind. One of its arguments is a pointer to a location in the application which address space that contains the data to be written. The kernel must confirm that the address is truly a part of the address space of the process and does not overlap the address range reserved for the kernel. In the case of a system call which writes

data into the user address space (for example, **read**), the kernel additionally has to check that the destination address is actually writable.

In kernel versions up to 2.1.2, each user memory access had to be guarded by a call to the function **verify_area**. This function checks that an address range is valid for a certain operation (read or write). The **verify_area** approach has four problems:

1. It is slow, because the kernel has to look up the virtual memory area covering the address range in question. Virtual memory areas (VMA) are the data structures that the kernel uses to keep track of the memory mapped into each process. Searching for a certain VMA is a time-consuming process, even with an efficient algorithm. (Linux uses Adelson-Velskii-Landis trees to optimize VMA retrieval.)
2. It is normally not required, because most programs provide valid pointers to the kernel. Despite this, the kernel has to spend precious time for each user memory access to guard against the rare, “buggy” application.
3. It is error prone. A programmer can easily forget the **verify_area** call before a user memory access, since the address verification and the actual user memory access are separate functions.
4. It is not always reliable, because in new, multi-threaded applications the memory mapping can change between a **verify_area** operation and the actual user memory access.

In particular, point 4 required that a new solution be found for verifying addresses. With the advent of real multi-threaded programs for Linux, it was no longer just a cosmetic problem.

Requirements for a Better Solution

A new solution for the problem of address verification had to meet the following criteria:

1. Run as quickly as possible for the normal case of valid addresses.
2. Provide an easy-to-use programming interface for kernel programmers.
3. Be stable and correct for all cases.

Instead of implementing the address test in software, points 1 and 3 of the requirements best met by giving the virtual memory hardware (present in all Linux-capable hardware) the real work. Point 2 resulted in a merge of the access check and the actual memory access into a single access function.

Implementation

The implementation relies on the MMU (memory management unit) to do the right thing. Whenever the software tries to access an invalid address, the MMU delivers a page-fault exception. This is true not only for user applications, but also for the kernel. If the kernel tries to perform an invalid memory access, a special handler intercepts the resulting exception and fixes the access. Because the handler is only invoked for problematic cases, the normal memory access has almost zero added overhead. Some experimental implementations of the new user memory access scheme have been done by Linus in kernel versions 2.1.2 to 2.1.6. They proved that this scheme works in the real world, but has a rather clumsy API inside the kernel. For Linux 2.1.7 Richard Henderson implemented the current, much-improved version introduced in this article. It is now available for most of the architectures supported by Linux. I use the x86 implementation as an example; the code for the other supported architectures is very similar.

In this new version, only the macros and functions in the machine-specific header file uacess.h are allowed to access the user address space. There are functions and macros to handle zero-terminated strings, to clear memory areas, and to copy single values to and from the kernel. One of the macros is **get_user**, which is called with two arguments: *val* and *addr*. It copies a single data value from the user space address *addr* to the variable *val*. On success, it returns the value 0, on failure -EFAULT (“bad address”). Its source code in the file uacess.h is somewhat hard to understand and even harder to explain.

Instead, I will track down a usage example from the function **rs_ioctl** in drivers/char/serial.c. The line of source code in the kernel is:

```
   error = get_user(arg, (unsigned int *) arg);
```

Listing 1. C Compiler Output

Looks innocent, doesn't it? Well, Listing 1 shows what the C compiler makes of this code. Let's do a line-by-line walk-through, beginning with the register assignments. On output, register ECX contains the value read from the user address space, and register EDX contains the error code of the access operation. Register EBX holds the address of the value to be read. Line 01 initializes the error code with **-EFAULT**. In line 02, the resultant value is set to 0.

In lines 03 to 07, the kernel does the checking which ensures that the address in register EBX does not overlap with kernel memory. First, it checks if the access is from inside the kernel. This situation can occur when the kernel itself invokes a system call (for example, in the NFS client implementation). In this case, access is always granted. If the access is from outside the kernel, it checks

that the given address range does not overlap the range reserved for the kernel.

In Linux, the kernel address space is mapped into the address space of every process. The kernel memory starts at address 0xC0000000. If a user program were allowed to pass a pointer to an address in kernel memory as an argument to a system call, the access by the kernel would not cause a page-fault exception. This in turn would allow the user program to overwrite kernel data. In our example (integer access = 4 bytes), the highest possible address a user process is allowed to access is 0xC0000000 - 4 = 0xBFFFFFFC = -1073741828. Every address larger than this touches kernel memory and is therefore invalid. If memory addresses are invalid, execution continues at label .L2395 in line 21.

Now that the preliminaries are over, we prepare to access the data pointed to by EBX. Assuming that the access will be successful, we set register EDX to 0 (line 09). In line 10 the actual access takes place. Note that the address of the instruction that might fault is tagged with the local label 1. The following code in the lines 13 to 15 unconditionally set the error code to **--EFAULT**, the resultant value to 0, and jumps back to local label 2 in line 16. This looks like an infinite loop—it isn't.

Another important decision that Linus made at the beginning of the 2.1 development cycle was to completely abandon the a.out executable file format for kernel development in favor of the more modern ELF. Normally, ELF is only associated with easily shared library support for user applications. However, that's only one of its nice features. Another one is that ELF binaries (both object files and executables) can have an arbitrary number of named sections.

The **.section .fixup, "ax"** command in line 12 instructs the assembler to place the following code in the ELF section named **.fixup**. In line 16, we switch back to the previous code section; normally this is **.text**. This means that the assembler moves the code from lines 13 to 15 in the generated object file out of the normal execution path. Lines 17 to 20 accomplish a similar task by instructing the assembler to place two long values into the ELF section called **__ex_table**. These two values are initialized to the address of the instruction that might fault (label 1 backward, 1b) and the address of the fixup code (label 3 backward, 3b).

With the help of the command **objdump**, which is one of the GNU utilities, we can examine the internal structure of the linked kernel. See Listing 2. Internal Kernel Structure.

Listing 3. Code Left in Normal Execution Path

As expected, there are two sections named **.fixup** and **__ex_table. objdump** also shows us the code which remains in the normal path of execution. Looking at Listing 3, we can see that the whole user memory access is reduced to just 12 machine instructions. The code originally bracketed with the **.section** directives is now in the section **.fixup**:

```
c01a16bf <.fixup+17b3> movl   $0xfffffff2,%edx
c01a16c4 <.fixup+17b8> xorl   %ecx,%ecx
c01a16c6 <.fixup+17ba> jmp    c018fafd <rs_ioctl+359>
```

The ELF section **__ex_table** contains pairs of addresses bracketed by <>: the faulting instruction address and the matching fixup code address. In our example, we expect the pair <c018fafb,c01a16bf>. Using the command:

```
objdump --section=__ex_table --full-contents\
        vmlinux
```

and a small program to swap bytes from internal representation to a human-readable form gives us this output:

```
c018f292 c01a1699 c018f51d c01a16a5
c018f5a5 c01a16ad c018fad0 c01a16b5
c018fafb c01a16bf c018fc5b c01a16cb
c018fd02 c01a16d5 c018fd72 c01a16e1
c018ff5a c01a16e9 c018ff7b c01a16f3
```

Now we have all the pieces we need to assemble the final picture. When the kernel accesses an invalid address in user space, the MMU generates a page-fault exception. This exception is handled by the page-fault handler **do_page_fault** in the C file arch/i386/mm/fault.c. **do_page_fault** first obtains the unaccessible address from the CPU control register CR2. It then looks for a VMA in the current process mappings that contains the invalid address. If such a VMA exists, the address is within the virtual address space of the current process. The fault probably occurred, because the page was not swapped in or was write protected. In this case, the Linux memory manager takes appropriate action. However, we are more interested in the case in which the address is not valid—there is no VMA that contains the address. In this case, the kernel jumps to the **bad_area** label.

At this point, the kernel calls the function **search_exception_table** to look for an address at which the execution can safely continue (fixup). Since every faulting instruction has an associated fixup handler, this function uses the value in **regs->eip** as a search key.

Loadable kernel modules in Linux complicate matters. After insertion at runtime, they are effectively part of the running kernel and can access user memory just as any other part of the kernel. For that reason, they have to be integrated into the exception-handling process. Kernel modules must provide their own exception tables. On insertion, the exception table of a module gets

registered with the kernel. Then **search_exception_table** can look through all of the registered tables, performing a binary search for the faulting instruction.

To access the start and the end of an exception table, we use a feature of the linker which resolves a symbol name consisting of the name of a section prepended with **__start__** or **__stop__** to the start or end address of the section in question. This feature allows us to access the exception table from C code. If **search_exception_table** finds the address of the faulting instruction, it returns the address of the matching fixup code. **do_page_fault** then sets its own return address to the fixup code and returns. In this way, the exception handler gets executed. The exception handler sets the value we tried to get to 0 and the return value of the **get_user** macro to **-EFAULT**. Then it jumps back to the address immediately after the failed user memory access. It is the task of the surrounding code to report the failure back to the user application.

Summary

The new user access scheme implemented in the 2.1 development kernels of Linux provides an efficient and easy-to-use way to copy data from and to the kernel address space. The added overhead for the average, error-free case is close to zero. Because access and test for accessibility are atomic, the original problem with multi-threaded applications is resolved. This solution meets all four of our requirements.

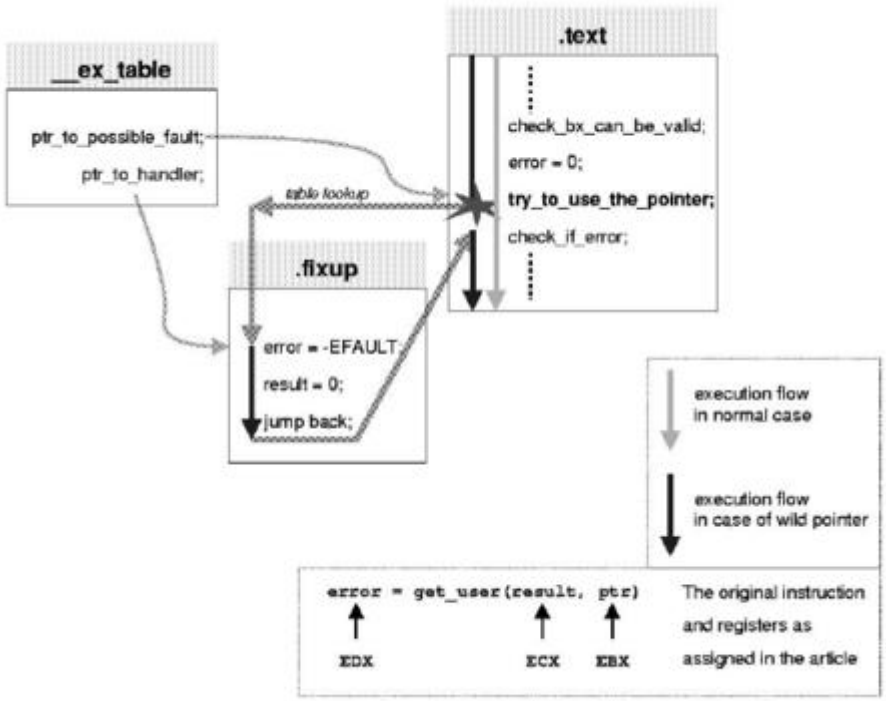


Figure 1. Execution Flow Chart

The name Jöerg Pommnitz can be found throughout the Linux kernel code. He recently changed jobs and moved without giving us his direction.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

The Dotfile Generator

Jesper K. Pedersen

Issue #42, October 1997

The Dotfile Generator is a useful tool to help in configuring programs by using X11 widgets to make option choices.

The Dotfile Generator, TDG, is a configuration tool that configures programs using X11 widgets like check boxes, entries, pull-down menus, etc. In order for TDG to configure a given program, a module must be built for it. At the moment modules exist for the following programs: bash, fvwm1, fvwm2, emacs, tcsh, rtin and elm.

TDG is freely available and can be downloaded from ftp://ftp.imada.ou.dk/pub/dotfile/dotfile.tar.gz. The home page of the Dotfile Generator is at http://www.imada.ou.dk/~blackie/dotfile/.

Introduction

The Unix system was developed many years ago, long before graphical user interfaces became commonplace, so that most of the applications today work fine without a graphical user interface. Examples of these applications are editors and shells.

A basic concept in Unix is that the programs are very configurable. For example, in Emacs if the user asks to go to the next line after the end of a file, there are two logical ways for Emacs to handle this situation:

1. Insert a blank line, and move to it.

2. Beep, to tell the user that there is no next line.

Instead of implementing only one of the solutions, the people behind Emacs chose to implement both, and let you decide which one you prefer. Since the program works without a GUI, the standard method for configuring such options is to use a *dot-file*. In this file, you can *program* the method emacs will use.

This solution requires that the user learn the programming language used in the dot file and read lots of documentation to discover the available configurations. This task can be difficult and tedious, and for that reason many users choose to use the default configuration of the program.

If you take a look at some dot files, you may find that most of the configurations can be described by the following items:

- Configurations with two possibilities (like above)
- Configuration where the program wishes to know a number or text. Examples of this could be questions like: "How many times should one press ctrl-d to quit?", and "e-mail address to use in the **Reply-to** field?".
- Configuration where the user may choose an option from a list, e.g., "Which editor would you like to use: emacs, jed, vi or vim?"

The configurations above can be easily done with a GUI using the following widgets in order: a check box, an entry and a pull down menu. This is exactly the method used in TDG.

The Basic Concept of TDG

TDG is a tool which configures other programs (e.g., Emacs, bash and fvwm) with widgets like those described above and many more. The widgets are placed in groups to make it easy to find the correct configuration without having seen it before. And most important of all, help is located at the configuration of each option instead of in a manual far away. To get help, you just press the right mouse button on the widget that contains the configuration you wish to know more about.

When you start TDG you are offered a list of standard configurations from which you pick one to use as a starting point. This is convenient if you do not have a dot-file for the given program or if you would like to try a new configuration. If, on the other hand, you already have a dot-file that you would like to put the finishing touches on, you can read this file into TDG. Note, however, that not all modules have the capability to read the dot-file (the fvwm2, rtin and elm modules have, the other modules do not, since it is complicated to create such a parser.)

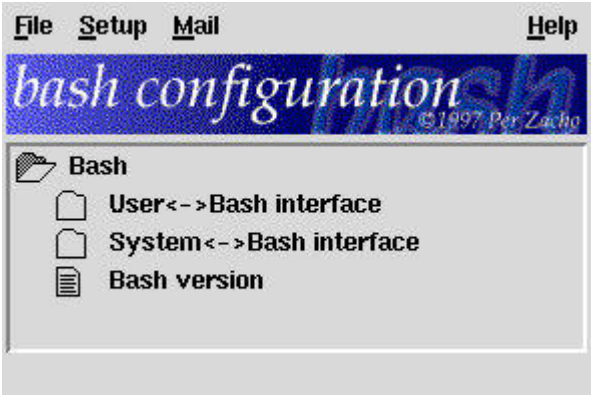


Figure 2. TDG Menu Window

When you have selected a start-up configuration, the menu window is displayed (see Figure 2). In this window, you can travel through the configuration pages in the same way you would through a directory structure. If you select a page, a new window is displayed showing the configuration for this page (see Figure 3). This window is reused for all the configuration pages, i.e., only one configuration page is visible at a time, so you do not have to destroy the window yourself.

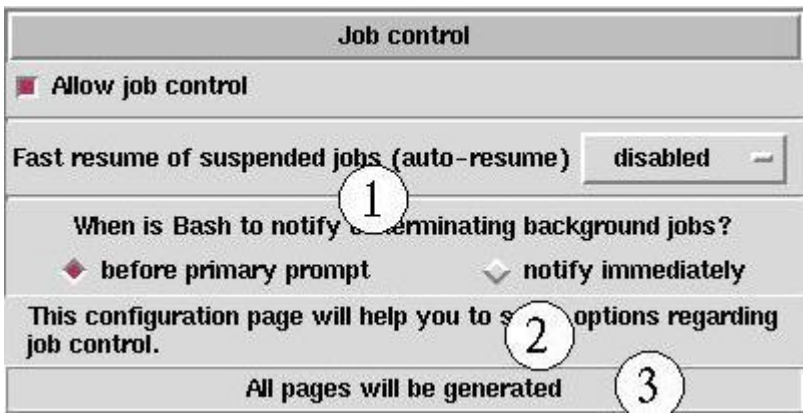


Figure 3. Configuration Page

The actual configuration is located in region 1 of Figure 3. Region 2 is the help region, and it is in this region that help for the whole page is shown when requested. Help for the individual configuration is also shown here, when the right mouse button is clicked on one of the widgets. In region 3, information is shown on what will be generated. There are three possibilities:

1. You can generate all pages. This is the most natural thing to do, when you want a configuration for a given program.
2. You can generate just the page shown. This is useful if you are playing around with TDG to see what will be generated for the different configurations.
3. Finally, you can tell TDG to generate specific pages using the radio buttons in this region.

In the **Setup-Options** menu, you can select which of the above three methods to use.

When the configurations are complete, you must tell TDG which file you wish to generate from the Options menu (**Setup->Options**). Now, it's time to create the actual dot-file by selecting **Generate** in the **File** menu.

Once you have generated the dot-file, you may decide to change some of the configuration. If so, go to the configuration page in question, change your configuration and regenerate. If, however, you are testing several different options for a single configuration (i.e., several items from a pull-down menu), you may find it cumbersome to generate the whole module over and over again. In this situation, you can chose **Regenerate this page** in the **File** menu. Note that if some part of the configurations on the page affects other pages, these other pages will not be generated unless you regenerate the whole module.

To see how to use the generated dot-file, go to the **Help** menu, and select "**How to use the output**".

The Configuration Widgets

TDG uses a lot of widgets for configuration of the different options. Some of them are well known from other applications and include: check boxes, radio buttons, pull-down menus, entries, text boxes (for multi-line text), directory and file browsers. Others, however, are specifically designed for use in TDG.

The ExtEntry Widget

The ExtEntry is a container that repeats its elements, just as a list box repeats labels. A number of the elements in the ExtEntry may be visible on the screen at one time. The elements in the ExtEntry can be any of the widgets from TDG. An element in an ExtEntry is called a tuple. In Figure 4, you can see an ExtEntry from the tcsh module.

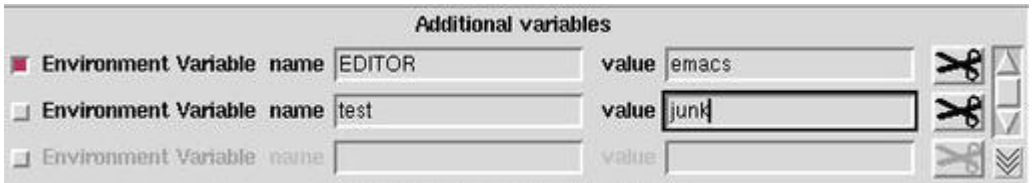


Figure 4. ExtEntry for tcsh

This ExtEntry has three visible tuples, although only two of them contain values (as you can see, the third one is grayed out). To add a new tuple to the ExtEntry, press the button in the lower right corner, just below the scroll bar. If the

ExtEntry contains more tuples than can be shown in the window, you can scroll to the end of the tuple list using the scroll bar.

When the left mouse button is clicked on one of the scissors, a menu with four elements is displayed. These elements are used to cut, copy and paste tuples within the ExtEntry.

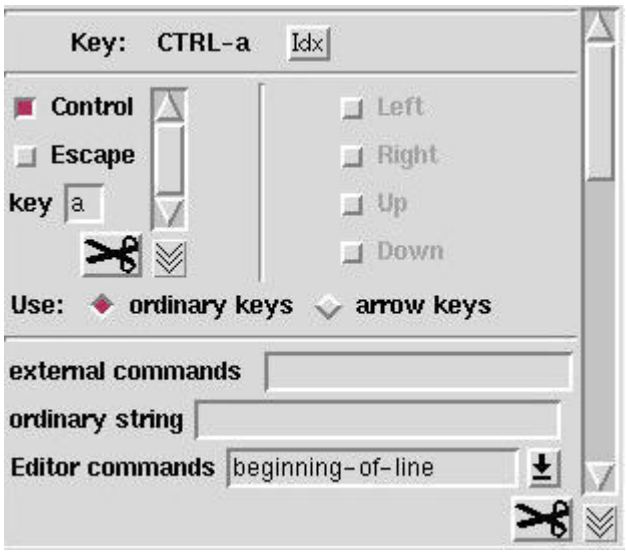


Figure 5. ExtEntry Display of Large tuple

If the tuples get very large, only one of them can be displayed on the screen at a time (see Figure 5). When the tuples contain many widgets, scrolling the ExtEntry becomes slow. In these cases, the ExtEntry may have a quick index. In Figure 5, you can see the quick index at the top of the ExtEntry (it's the button labeled **Idx**). When this quick index is invoked, a pull-down menu is displayed with the values of the element associated with the quick index. This makes it easier to scroll ExtEntries.

The FillOut Widget

Every shell has a configuration option called **Prompt**, a text string that is printed when the shell is ready to execute a new command. Special tokens can be inserted in this text, and when the prompt is printed, these tokens are replaced with information from the shell. For example, in bash the token **\w** is expanded to the current working directory.

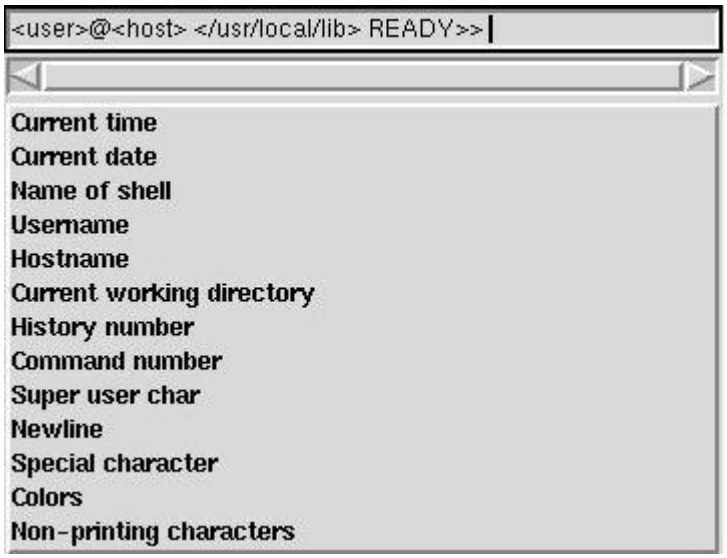


Figure 6. FillOut Widget

In TDG, a special widget called **FillOut** has been created to do configurations like the one above. In Figure 6, you can see a FillOut widget from the bash module. At the top of the widget there is an entry in which you can type ordinary text. The tokens are placed below it. If you select one of the tokens, it is inserted in the entry at the point of the cursor. Some of the tokens may even have some additional configurations. For example, the token "**Current working directory**" has two possible options: "**Full directory**" and "**only the last part**". When tokens with additional configurations are selected, a window is displayed in which these configurations can be made. If you wish to change such a configuration, press the left mouse button on the token in the entry.

The Command Widgets

TDG can be extended by the module programmer through the Command widget making it possible to configure specific options with widgets s/he has developed. At the moment three such widgets exist: the directory/file browser, the color widget and the font widget.

The widgets appear as a button within TDG, and when the button is pressed, a new window is displayed in which the actual configuration can be done.

Save, Export and Reload

When you have configured the different options in TDG, you may wish to leave it and come back later to change some of the configurations. When you leave TDG, you can save your changes using an option in the **File** menu. Next time you enter TDG, your saved file will be one of the the files you are offered as a start-up configuration.

One important point to note is that this "**save file**" option is an internal dump of the state of TDG. That is, this file depends on the version of TDG and the module. Therefore, if you wish to send a given configuration to another person, this format is not appropriate. A version-independent format does exist called the *export format*. To create such a file, you have to select **Export** instead of **Save** in the **File** menu.

To restore the configuration on a single page to its original value or to merge another person's configuration with your own, select **Reload** in the **File** menu. To tell TDG that you want to reload only some of the pages, select the **Detail** button in the load window. This button brings up a window in which you can select the configuration pages to reload. Here you can also tell it how you want the pages to be reloaded. You have two possibilities:

1. Overwrite—The pages that you are loading will totally overwrite the contents of the file.
2. Merge—Tuples in the ExtEntries are appended to those already present in the module. Other configurations are ignored in the file.

Here's another difference between the save files and the export files—you cannot merge with save files. In other words, if you have a save file that you wish to merge, first you have to load and export it, and only then can you merge with it.

The End

Additional information as well as information on work currently in progress can be found on http://www.imada.ou.dk/~blackie/dotfile/, the home page of TDG.

I have just finished a module for procmail, a mail filter for sorting your incoming e-mail.

John D. Hardin (jhardin@wolfenet.com) is working on a module for configuring the firewalling and IP Masquerading setup for stand-alone systems connected to the Internet via dial up. He may expand it to more general firewalling in the future.

If you have the time, I would like to encourage you to develop a module for your favorite program. On the home page of TDG there is a link to a document that describes how to create a module for TDG. Send me e-mail at blackie@imada.ou.dk, and I will be happy to help you get started.

Jesper Pedersen lives in Odense, Denmark, where he has studied computer science at Odense University since 1990. He is a system manager at the university and also teaches computer science. In his spare time, he does Jiu-

Jitsu, listens to music, drinks beer and has fun with his girlfriend. He loves pets, and has a 200 litre aquarium and two very cute rabbits. His home page can be found at http://www.imada.ou.dk/~blackie/, and he can be reached via e-mail at blackie@imada.ou.dk.

Archive Index Issue Table of Contents

Advanced search

Best of Technical Support

Various

Issue #42, October 1997

Our experts answer your technical questions.

Creating Hunt Groups

I need to add a new user for a static IP account. How can I set it up so that the new user can dial into our regular hunt group? The way I have it set up now, I would need to assign each user a modem just for them. I was told that I can set things up so users can call into our main hunt group number. —John Jhong Red Hat

That is exactly what we do here at Red Hat. We have a four-line hunt group for which we give users the number only to the first line. Then we put a modem on each of the four lines. If the first line is occupied, the caller is rolled to the second, then the third, then the fourth. I assume this is what you mean by “hunt group”.

All of our users have a static IP number, too. We simply create an extra account for each user. In typical ISP fashion, we usually make it their normal login ID preceded by a capital <\#145>P'. (For example Pdjb would be the dial in account for djb). Next, we create a short shell script that is set up as the “shell” for Pdjb. A typical script might look like this:

```
#!/bin/bash
/usr/sbin/pppd modem crtscts netmask \
255.255.255.248 :static.ip.address.here
```

We would put that in /usr/local/bin/Pdjb.

Next, you make that script executable and edit the shell for your Pdjb user to look like so:

```
Pdjb:fakepasswd:1000:1000:RHS ppp account:
     /home/ppp:/usr/local/bin/Pdjb
```

Then the user can dial in directly as Pdjb and PPP will start automatically. This is only a brief introduction to setting up PPP. For more detailed information, see Robert Hart's PPP-HOWTO available at http://sunsite.unc.edu/LDP. —Donnie Barnes, Red Hat Software redhat@redhat.com

Configuring Master Daemons

When I try to start wu.ftpd or in.telnetd, I get the error "Socket operation on non-socket" —Ian Webber Slackware 3.2

Those aren't programs. They are daemons. They should be run only from **inetd**, which is a "master daemon" that runs all the time, listening for things like **FTP** and **telnet** requests. You will find configuration information for inetd in /etc/inetd.conf.

Once inetd receives a request for FTP or telnet, it will then start **in.ftpd** or **in.telnetd** and connect its I/O to the socket that requested the connection. With this system, you don't need to have many daemons sitting in memory doing nothing, since inetd can start them when they are needed. —Donnie Barnes, Red Hat Software redhat@redhat.com

I Still Can't Print!

I have friends who chuckle at me because I have had Linux for about a year and still can't figure out how to get it to print to my HP printer. —Randy Barrett Red Hat 3.0.3

Since you don't specify the model of your HP (and you've been trying for so long) I'll assume it is a 540 or similar printer. What you need to do is download the printing filter **aps-491.tgz** in /pub/Linux/system/printing from sunsite (or your favorite mirror). All you need to know is to which lp port your printer is connected. The APS Filter installation software will automatically write the filter and set up /etc/printcap so that you can start using your HP printer. —Mark Bishop, Vice President Southern Illinois Linux Users Group mark@vincent.silug.org

XView Shuts Down Monitor

Sometimes when I start the **XView** application by typing **openwin**, the monitor turns off. Since the monitor power supply is the same as that of the whole system, I need to start all over again.

I have tried various combinations with the **XConfig** file. —Harjeet

I recommend downloading **XFree3.3** from ftp.xfree86.org or your favorite mirror. Read the RELNOTES file before you download anything. XFree 3.3 has a pretty decent tool (**XF86Setup**) that you run to set up XFree for operation with your particular video card. Once you get XFree up and running by entering **startx**, XView should pop up with no problem. —Mark Bishop, Vice President Southern Illinois Linux Users Group mark@vincent.silug.org

Linux C Graphic Libraries

I just switched over to Linux to do my graphics programming. I cannot find basic Linux C libraries that are equivalent to the Turbo C++ version of C graphic libraries. Where can I find those? —Christopher Carver Red Hat 4.0

There is no direct freeware clone of the Borland BGI graphics library, but there are a number of libraries that should serve your purposes, whether you want to display a few curves on the screen or render complex 3D images. Visit your favorite Sunsite mirror, and explore the /pub/linux/libs/graphics directory. You should find everything you need. If you are looking for a direct clone of the BGI graphics library, look for the file **bgi_library.tar.gz** which you should also find in the aforementioned sunsite location. It is a good shareware product (with an extended commercial version) that provides source API compatibility. —Chad Robinson, BRT Technical Services Corporation chadr@brttech.com

Archive Site for Kernel Development

Where is an there an archive site for kernel source code and documentation for developing the Linux Kernel? —Jung-Ho Park Red Hat 4.1

The Kernel Hacker's Guide is a document written just for this purpose. You can find it on your favorite Sunsite mirror in /pub/linux/docs/kernel in the subdirectory kernel-hackers-guide. There is other documentation in this directory that may be of assistance to you. Also, the Usenet newsgroup comp.os.linux.development.system has ongoing discussion of kernel development issues. —Chad Robinson, BRT Technical Services Corporation chadr@brttech.com

If you would like to submit a question for consideration for use in this column please fill out the web form at http://www.ssc.com/lj/ or send e-mail with the subject line “BTS” to info@linuxjournal.com.

Answers published in Best of Technical Support are provided by a team of Linux experts. If you are interested in becoming a part of that team please e-mail info@linuxjournal.com.

Archive Index Issue Table of Contents

Advanced search