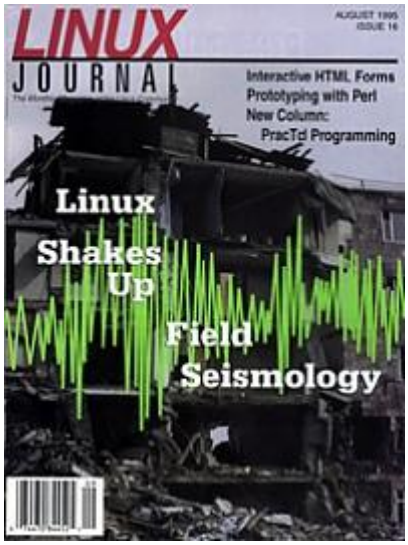
Linux Journal Issue #16/August 1995



HTML Forms: Interacting with the Net

Eric Kasten

Issue #16, August 1995

In this last of three articles on the World Wide Web, Eric discusses how to create interactive HTML forms, which allow you to collect data and interact with users as well as serve documents.

You have set up a World Wide Web server, and now have a number of HTML (hypertext markup language) documents for web-surfing visitors to enjoy. You're comfortable with HTML, and are ready to find new things for your server to do. In your network travels, you remember filling out some electronic forms to give feedback to the creator of one of your favorite home pages.

This article will help you acquire the basic knowledge needed to write HTML forms, and explains what needs to be done so that you and your server can interact with your Web visitors.

Elements of a Form

A working form really consists of three basic elements. The first is the form itself. The form is constructed using HTML text, as for your homepage, with a few different markup tags. The second element is the script or program. This program must be constructed in accordance with the common gateway interface (CGI) specification, if it is to communicate properly with your server and the user's Web client. The CGI script is the engine behind the interface; it will actually act on the data the user types into the form. The third element is the httpd (hypertext transfer protocol daemon) server, which calls the CGI program, passing it the data the user has entered.

Let's take a look at what elements a form can posses. Much like other HTML constructs, forms are built using markup tags and simple text. A form is encapsulated by **<FORM>...</FORM>**, where the **...** is replaced by text and other form markups. Keep in mind that markup tags are case insensitive, though I will

continue to capitalize them for clarity. Following is a list and descriptions of the major available form markup tags.

<FORM>...</FORM>
Indicates the start and end of an HTML form.

<INPUT>...</INPUT>
Indicates the start and end of form input.

<SELECT>...</SELECT>
Indicates the start and end of a selection list.

<TEXTAREA>...</TEXTAREA>
Indicates the start and end of a free-form text input area.

FORM Markup Tag

Form markup tags may use attributes to help control how a form will be displayed to the user. Let's take each markup tag in turn, and examine the valid attributes for each. First let's look at the **FORM** tag.

ACTION
Typically a URL indicating a script or program to be executed.

METHOD
Valid values are **POST** and **GET**.

The **ACTION** attribute specifies a URL (uniform resource locator) which will be used to carry out some action based on what is entered in the form. The URL usually specifies a program, which exists in a script directory on the server. For instance, **http://some.server/cgi-bin/donothing.sh** will result in the form data being returned to the program **donothing.sh** for processing. The program will then return an appropriate response to the client.

The **METHOD** attribute is used to specify how the data which is entered into the form is to be returned to the server. The data may be appended to the URL specified by the action attribute using the **GET** method. When the **GET** method is used, the http server will pass the information to the **ACTION** program encoded in an environment variable. When the **POST** method is used, the http server will pass the information to standard input.

```
<FORM ACTION="http://www.you.org/cgi-bin/donothing.sh" METHOD=POST>
```

begins the definition of a form which is processed by the donothing.sh script on the current host, which reads data from its standard input.

INPUT Tag

The **INPUT** tags are used to specify fields where data can be entered by the user. This tag, like all of the remaining form markup tags, must appear between a **<FORM>** tag and its associated **</FORM>** tag. Following is a list of valid attributes.

NAME
: Indicates a symbolic name for the input field. The **ACTION** program uses this to differentiate fields.

TYPE
: Specifies the type, such as **checkbox** or **radio button**, that is to be used.

VALUE
: This gives a default value for the input field.

CHECKED
: A boolean indication of status for elements such as checkboxes.

SIZE
: The physical display size of text entry fields.

MAXLENGTH
: The maximum allowable number of input characters for text entry fields.

The **NAME** of an **INPUT** field allows fields to be differentiated or grouped. The name of a field is used by the **ACTION** program to determine what a user entered in each field of the form. The **NAME** attribute is also used to establish logical groupings of some form element types, specifically radio buttons.

Valid settings for the **TYPE** attribute are **checkbox**, **text**, **password**, **radio**, **hidden**, **reset** and **submit**. A **checkbox** is an element which can take on one of two states, either checked or not checked. This provides a basic boolean true or false element for form entry. The **text** element provides a single-line text entry field in which the user can enter data. A **password** field is a **text** entry field in which the entered text is hidden from view in some fashion.

Radio buttons are groups of buttons which allow a single button to be toggled at a time. The other buttons in the group are untoggled when one button of the group is selected. A **radio** button group is established by setting the **NAME** attribute for each button in the group to the same value.

A **hidden** input is not displayed to the user at all, and the user cannot modify it. A **hidden** input encodes state information into the form. For instance, it might

be possible to have one form which should be processed in different ways, depending on context. Each instance of the form could include **hidden** input indicating the context and directing the processing appropriately.

Of particular note are the **submit** and **reset** input types. Clicking on **submit** causes the form contents to be transmitted to the server, and then to the **ACTION** program for processing. The **reset** button causes the form elements to be set to their initial values, allowing the user to easily return the form to its initial state.

A default value for a form element can be specified using the **VALUE** attribute. For text entry elements, this indicates a default string of characters that are initially present when the form is retrieved. If the field is a **radio** button, this is the value the element takes on when it is checked (when the element isn't checked, it has no value). For the **submit** and **reset** elements, the **VALUE** attribute can be used to set the button label.

The **CHECKED** attribute is valid only for the **checkbox** and **radio** elements. If the **CHECKED** attribute is present, the radio button or checkbox is toggled by default. Setting the physical length of a text entry field can be done by using the **SIZE** attribute.

The **MAXLENGTH** attribute limits the number of characters that are accepted in a particular text entry field.

SELECT Tag

SELECT is the next major markup tag. The **SELECT** tag is used to encapsulate a selection list. Several **<OPTION>** tags may be included between a **<SELECT>** and a **</SELECT>**, to add elements to the list. A selection list may take on two physical forms. If it has a **SIZE** of one, it appears as a popup menu. If the **SIZE** attribute is greater than one, it appears as a scrollable list displaying **SIZE** options one at a time. Here are the possible attributes of the **SELECT** tag:

NAME
: Indicates a symbolic name for the selection menu.

SIZE
: The physical number of lines that are visible at any time.

MULTIPLE
: If this attribute is present, multiple items of the list may be selected at one time.

These attributes are straightforward, and I'll leave them for your exploration later. Before we move on, I should mention a little more about the **<OPTION>** tag. The option tag can have the attribute **SELECTED**. When present, this attribute indicates that a particular list item is selected by default. The **<OPTION>** tag is much like the **<li>** of normal HTML lists; it does not require a terminating **</OPTION>** tag. Instead, the appearance of an **<OPTION>** tag indicates the beginning of a new list item and the termination of any preceding items. Also, a selection list item can be only simple text. List items cannot be marked up, nor can they be anchored items.

TEXTAREA Tag

A form element where a user can type in free-form text, much like entering text into an editor, is constructed using the **TEXTAREA** tag. A text entry area is has the basic form of:

```
<TEXTAREA>default text</TEXTAREA>
```

The ***default text*** is the initial text, if any, which is present in the text entry area. This form element has three easy-to-use attributes.

NAME
: Indicates a symbolic name for the selection menu.

ROWS
: The vertical size of the text entry area.

COLS
: The horizontal size of the text entry area.

Assembling the Pieces

Listing 1. A Simple HTML Form

Now that we know what things we have available, let's create a basic form. Listing 1 shows a simple HTML form, while Figure 1(139K) displays how Mosaic might present this form.

Keep in mind that the **ACTION** attribute needs to specify your host and a valid script or program. In the example, the shell script echo.sh (shown later) will be executed on your.http.host when the form is submitted. The script or program needs to reside in a directory which your server recognizes as a valid location for executable programs. Be sure to check the documentation for your server to be sure it is configured properly to allow for this sort of execution. A typical

location for these sorts of programs is in the cgi-bin directory under the server root, and that is how this example is configured.

Interacting with the Client

The form is only one of the three parts necessary to interact with a user. The second is the http server, which we will not cover here (please refer to the documentation for your server). The third is a CGI program or script. As mentioned above, these programs *must* reside in a directory recognized by the http server as a valid location for executables. A CGI program needs to be able to understand the encoded form data as it is returned from a client, and must be able to respond appropriately. The encoded form data will appear either on the command line or in the environment variable **QUERY_STRING**, depending on whether a **METHOD** of **GET** or **POST** is used. Typically, a program needs only write the necessary response on stdout, and the response will then be transmitted back to the client by the http daemon.

A number of environment variables are also typically set by the server for the CGI program's use. Following is a *partial* list of environment variables that I find useful. Please refer to hoohoo.ncsa.uiuc.edu/cgi/env.html for further discussion of other environment variables.

REQUEST_METHOD
: Set to the **METHOD** used to make the request.

QUERY_STRING
: Set to the encoded form data when the **GET METHOD** is used.

REMOTE_HOST
: Set to the remote hostname if available.

REMOTE_ADDR
: Set to the IP address of the remote host.

CONTENT_LENGTH
: The length of the data returned in a client's query.

Usually, a CGI program need only respond to a request with an appropriate http header, possibly followed by a document. The response is simply written on stdout, where the data will be returned to the client. A header consists of an http header directive followed by a relevant text string. The header is terminated by a blank line. Two of the most used header directives are the **Content-type** and **Location** directives. The **Content-type** directive indicates the type of data which is to follow. For example, **Content-type: text/html** indicates that the document which follows the header on stdout is written in HTML. The

Location directive is used to provide a means by which redirection can take place. For instance, **Location: http://goto.another.host/web/doc.html** would cause a client to retrieve the document specified in the URL.

Probably the easiest way to explore the construction of a CGI program is with an example. Listing 2 shows a shell script which will respond to a client's HTML form request.

Listing 2

The response is to echo the encoded query, some of the environment variables, and the decoded content of the query. This program is useful as a test program when creating new forms, and as a base for building other CGI scripts. Figure 2(135K) displays the results of posting the form shown in Figure 1 to this script.

Examine the **QUERY_STRING** in Figure 2 Notice that spaces are encoded as addition signs, and that an ampersand in the input is encoded as a hex value preceded by a percent sign. Also notice that each name/value pair is separated by an ampersand. The shell script decodes this string back into the data as it was entered by the user. There are other programs, such as CERN's cgiparse, which will also help you decode CGI form data.

Conclusion

You now should have the basic building blocks of form construction and processing at hand. Many things which can be done with HTML forms and CGI programs, including providing man pages via http or constructing gateways for accessing other system information. Good luck, and have fun!

Resources

Eric Kasten has been a systems programmer since 1989. Presently he is pursuing his masters in computer science at Michigan State University, where his research focuses on networking and distributed systems. Well-thought-out comments and questions may be directed to him at tigger@petroglyph.cl.msu.edu. You may also visit his home page at petroglyph.cl.msu.edu/~tigger.

Archive Index Issue Table of Contents

Advanced search

Linux Goes to Sea

Randolph Bentson

Issue #16, August 1995

An Interview with Stephen Harris.

Randolph: *What is your background and how did you learn of Linux?*

Steven: I attended St. Peters College, Oxford University, 1987-1990, studing the joint honours school of Mathematics and Computation. This is where I was first exposed to Unix (Sun3 with SunOS 3.5, and High Level Hardware Orions under BSD 4.2 with some 4.3 extensions).

In summer 1990 I joined Papachristidis Ltd. (who do the administrative and agency functions for Hellespont Group) as one of a team of three. Various organisational changes mean I'm in charge of the whole lot now...

The Hellespont Group is a Greek shipping company, based in Piraeus, running a number of ULCC (Ultra Large Crude Carrier) and VLCC (Very Large Crude Carrier) oil tankers—some of the largest ships on the ocean, trading worldwide. There is an office in London as well, and the two offices are connected by an analogue leased line providing two speech and four data circuits. In the offices we run a network of Unix machines, with the users accessing the machine via dumb terminals (e.g. DEC VT320) and terminal servers. Our primary interface is simply users running **vi** and creating **roff** documents. Admittedly, not the most user-friendly of interfaces, but highly flexible and capable of being run on many Unix systems. On top of this we have written a large variety of shell scripts and C programs, also highly portable, and a well-defined file tree which determines where reports and files are to be stored, matching the hard-copy filing.

Back in Novemeber 1991 I had just got Interactive Unix disks (a real old version) and had installed it on my 386DX-20, when I heard in one of the comp.unix groups about a free Unix clone and a reference to alt.os.linux. I quickly found out where the kernel was, and downloaded Linus' root and boot disks. I was impressed.

Randolph: *It's clear that you enjoy working with Linux. How did you persuade your firm to consider Linux?*

Stephen: I showed the early Linux version to my Greek and American colleagues, who weren't impressed. I passed through 0.11 and 0.12 kernels when Owen LeBlanc's MCC-Interim Release 0.95 came out. 5 disks with a working Unix on them! Threw away Interactive (which never worked properly) and installed Linux. Peter MacDonald came out with his SLS package and I spent a fortune on phone bills downloading the disks.

At this stage the shipboard project was being planned around Sun Sparc systems (one per ship with VT terminals). My colleagues didn't think Linux was capable. SLS changed their minds, so I have Peter to blame (thank!) for convincing them. I offered to make copies of SLS available for people in the UK who couldn't afford to download it themselves, and Adam Richter sent me an Yggdrassil CD free. Now that I had the source code I built my own setup, which meant I could get around some of the problems SLS had.

From there it was a small step to experimenting in the office, installing a small Linux system on my desk, and determining if it could be used. Except for a few network problems (packet storms), it all worked quite well.

Randolph: *Can you tell us more about how Linux is used on board?*

Stephen: Firstly, the system now deals with 90% of the typed communications between ship and shore. Using software on the office hub, messages can be sent to the office and relayed out to third parties, using the much cheaper landlines. Frequently, one message is sent to multiple parties, so now the ship simply sends one message to the office and the computer automagically sends it to all the other recipients. Of course, the office doesn't just have telex either, and uses many alternatives, but that's another story.

Next, ship's reports can be created on the system, and printed, filed, sent to the office very easily. Similarly, reports and procedures written in the office can be distributed in source form to the ships for printing and local storage, making it an easy way to update manuals.

We are now sending more traffic than we were before installing the system, but the increase has had minimal cost effect compared to our savings... When we first installed fax machines onboard (many years ago) we initially saw a reduction in cost since fax was cheaper than telex, but then costs exploded when everything was faxed (even things that don't require it). Contrarily, the email system actually takes traffic away from fax (sending data as fixed-format email rather than fax) which has helped reduce costs more! We anticipate costs

to rise slightly above the current level (to nowhere near the previous one!) as we send more data, but this should improve efficiency and the ability of the office to monitor shipside better, and so is an overall gain.

Spares and inventory control has been done on one of the ships—the complete parts list from the ship's plans has been entered, allowing an easy search for the official specifications when needing to order replacement parts and so on.

Requisitions (spares, stores, etc) are already handled by a structured document that is parsed by a script and printed in the office, instead of needing to fax the document from ship to shore.

The possibilities are endless.

Randolph: *What was involved in installing this on board?*

Stephen: At the time we had a ship in drydock just outside Lisbon, so that was the perfect ship to test on. The ship wasn't going anywhere, and we had access to landside facilities in case we forgot something!

The initial design called for a computer and five terminals, which would be placed in strategic locations in the ship (Captains office, Chief Engineers, Ships Library, Cargo Control Room, and Engine Control Room) allowing access to the main system for those that needed it. Wiring through the ship was done with unshielded twisted pair (3 pair) cables, terminating in RJ12 connectors, with two sets of wire to each location. The main unit was placed in the Radio Room. The wiring was chosen because it meant that the same wires could be used for telephones, 10baseT ethernet, or serial cables, simply by making the correct modular cables.

Installation was pretty straight forward. My American colleague, also onboard with me, had spent the previous week buying and fitting together the hardware, and making sure it was actually capable of booting a minimal system. I also had my notepad running Linux, so I could build an emergency boot floppy set if everything failed with the tape.

This was the beginning of a daily find bug / fix bug cycle that lasted through the two weeks we were onboard. While I did this, my colleague installed the working files to make the file tree look identical to that in the office. The end result was a system that looked very close to that running in the office—on the vastly more expensive Sun's.

After this, we left the ship to purchase some new equipment, some different modems, and so on, with plans to return a month later before the ship left dock.

When we returned, we replaced the terminals with smaller Linux systems, so the setup we now have is a central 486DX2-66 and three client machines, all running X and NFS'd together. The central server acts as communications and printer server for the network. To cope with a failure of the server machine a simple script was developed to change some of the main configuration files on one of the clients so that it acts as the server. Admittedly a server without any of the user files, but enough that the communications can be continued, and some simple work can be done.

We have since installed a similar system on a further five ships, this time using only 486SX-33 VLB machines (with VESA graphics) as clients; machine prices dropped to a level that it didn't make sense to use any with less power!

Randolph: *What kinds of special problems do you find on board?*

Stephen: A ship is one of the most electrically noisy environments I can think of, but only the longest cable produced noise—and then only when left unplugged from the terminal! We were quite impressed that the serial cables worked over such a large distance.

Power for the system is also problem: the ships generate their own power: 220V at 60Hz. To cope with this, we bought American equipment, and had a transformer to convert from 220 to 110V. We had a Triplite UPS to cope with the inevitible power fluctuations, and an Isobar to try and cope with any surges. A week earlier, another ship's generator went overspeed and shot over 660V through the mains—not something I wanted to risk on the PC!

Connecting the modem to the system was a lot of hard work. (When in range of shore stations we use cellular phones and when at sea we use the satellite based voice communication system Inmarsat A.) This wasn't the fault of Linux though! The main problem was the modem. Eventually we got reliable connections using Taylor UUCP. Using the 'g' protocol we could get 300+ cps average throughput. Not good compared to what v32 should be able to do, but a **lot** better than telex! When we upgraded the office Suns to use Taylor UUCP we managed to get 600+ cps average throughput using the 'i' protocol. The ship polls the office at four fixed times a day (different ships have different times to cut down on collisions).

Randolph: *How do you administer these distant systems?*

Stephen: Basically we give the radio officers some intensive training in the office on a system configured the same as the server. We show them how to perform basic admin tasks, such as backups, rebooting, sending email, and so on. They are also shown the hardware, how to reseat cards, and so on. A big problem on the ship is vibration, and so we expect more hardware problems than software. That, along with scripts and cron jobs, copes with most of the forseable problems. For the others...well, I'm only a phone call away!

In fact, from the problems we have had so far, our expectation of hardware failure being the biggest problem seems correct. One PC failed with a bad power supply (luckily this was on the first ship in the month we were upgrading the modems and so on, so we set up a new server and put that onboard at the same time). One laser printer failed to feed properly. One Boca board port blew. But the hardware is cheap and spares can be sent to a port where the ship is due, and the radio officers can perform the physical swap out. Only one software problem has occured (and unfortunately recurs because I haven't found a fix yet—printer daemons on slaves sometimes “stick”).

Randolph: *What are your plans for future development?*

Stephen: Well, we are still building the network. We consider the configuration to be a success and are planning to add a similar configuration to our remaining ships.

The office communications hub has recently been replaced with a Linux server that now routes most messages for the group, including our external email connections. A longer-term plain is to upgrade the office, replacing the dumb terminals with networked Linux machines.

One thing we will **not** do is “version chase”. The kernel and libraries we are using are out of date. **But** while it is doing what we want adequately there is no need to upgrade. If I did, then I would spend most of my time sending updates to the ships and doing little else!

Linux has proven itself extremely reliable. The cheapness of the hardware has meant that we could and did build a network onboard the ships vastly superior to one that would have been made based around Sun equipment. The extra facilties of email over telex, and the relative cheapness of the connection has meant that shipboard data can be sent to the office in a more usable form.

Since Inmarsat A costs are approximate US $1 for 6 seconds, message size becomes an important consideration! We want to develop a system that is 8 bit clean, allows routing of traffic via various media (direct uucp logins, internet email, modem dialup etc) and provides a sophisticated “receipt” system so the

sender knows when the message has reached the destination (e.g. we don't want the ship to generate a receipt because it will cause extra ship-shore traffic (expensive) so the shore based hub will generate it once it knows the ship has collected it). Because all routing will be by my software, it can ensure the message survives whatever the transport restrictions are—e.g. with direct uucp it could gzip the file for sending, email it could uuencode, etc.

Randolph: *When can I get a tour? I live in Seattle.*

Stephen: Our ships (when they go to the USA) are mainly the other side—Galveston (for lightering, so they're offshore and you'd need helicoptor ride to reach them) and LOOP (Louisiana Offshore Oil Port—not easy to get to), so a tour of the ship is highly unlikely, I'm afraid. Of course, most of the time they spend away from any visible land travelling across the Atlantic Ocean, but sometimes they go to European ports (e.g. Rotterdam's Europort).

Randolph: *Besides sending Linux to sea, what other claims to fame can you make?*

Stephen: I'm the originator of the “I hate Linus Torvalds” thread in c.o.l where I said I hated him for making me upgrade my PC so I could run the excellent OS he had written (and was flamed by lots of people who never read the second page! Linus recognised it as a joke thankfully!)

I never did get round to sending him the postcard that was requested in the early release notes...

Randy Bentson has been programming for money since 1969—writing more tasking kernels in assembly code than he wants to admit. His first high-level language operating system was the UCSD P-system. For nearly 14 years he has been working with UNIX and for the last year he's been enjoying Linux. Randy is the author of the Linux driver for the Cyclades serial I/O card.

Archive Index Issue Table of Contents

Advanced search

Introduction to Lisp-Stat

Balasubramanian Narasimhan

Issue #16, August 1995

If you do heavy-duty statistical computing and have been looking for a powerful statistical package that runs under Linux, Lisp-Stat may be just what you need.

Although I installed Linux on a 80 Meg partition on my Gateway 33Mhz PC over a year ago, I really did not make serious use of Linux for my scientific work, mostly because I lacked disk space. Recently, I bought a new 1-Gig drive and that excuse went away. So I decided to install **Lisp-Stat**, a program that I use most for my statistical Computing.

Written by Luke Tierney at the University of Minnesota, Lisp-Stat is a powerful, interactive, object-oriented statistical computing environment based on the the Xlisp dialect of Lisp. It runs on under Microsoft Windows, Macs, and Unix based X11 systems almost uniformly. It has good graphical facilities for both static and dynamic graphics along with functions for common statistical computations.

Furthermore, using the foreign function interface, one can call C and Fortran programs from within Lisp-Stat. A byte-code compiler is available to speed up your programs once you have debugged them. Of course, one needn't use Lisp-Stat for statistical computing alone; I routinely use for all kinds of things: as a calculator, for figuring out grades of my students, as an engine for hypertext illustrations as well as matrix manipulations.

In this article, I will introduce you to some of the capabilities of Lisp-Stat. Although I use Lisp in this article, I will not get into the details of Lisp programming unless it impinges on our discussion. If you are new to Lisp, you might want to read article on Scheme by Robert Sanders, *Linux Journal*, March 1995, as Lisp and Scheme are closely related. Some of the comments I make apply actually to Lisp, but it serves no useful purpose to delineate what is the Lisp and what is the Stat part. You need not know Lisp to use any of the examples or to follow the article. If you get seriously interested in Lisp-Stat you should probably get a copy of Tierney's book titled **Lisp-Stat**, ISBN

0-471-50916-7, published by John Wiley. Besides being the canonical reference for Lisp-Stat, it provides a quick and practical introduction to Lisp.

Figure 1: An example of a Lisp-Stat session

A Quick Tour

Assuming that you have installed Lisp-Stat successfully, just type **xlispstat** to invoke the program. To quit the program, just type **(exit)**. Figure 1 shows a simple session. Case does not matter and the **>** you see in the figure is Lisp-Stat's prompt. The data I have used is the number of requests a WWW server honored during each of the *24* hours in a day. The **def** macro binds a variable name **requests** to the list of values.

As the example shows, calculation of summary statistics like the mean and standard deviation are trivial. Since Lisp-Stat is based on Lisp, you have all the power of Lisp for data manipulation. A rich set of data types is available including vectors, sequences, strings, matrices. In figure 1 the variable *A* is defined to be a 3x3 matrix and *B* is a list of three numbers. The example solves *Ax=b* for *x* by computing *A<+>-1<+>b* yielding the solution *[2, -6, 1]*. Note that *b* is a list while *A<+>-1<+>* is a matrix, yet the Lisp interpreter takes care of the types and in effect computes the product of a matrix and a vector.

Many of Lisp-Stat's functions operate on sequences which may be lists or vectors and they are *vectorized*, meaning that these functions can be applied to arguments that are lists and the result is a list of the results of applying the function to each element of the list. Some other functions are *vector reducing*, meaning that they can be applied to a list of arguments but they return a single number. In figure 1, the function **mean** is an example of a vector-reducing function; it treated the list of lists as a single long list and returned the mean of the long list. On the other hand, the function **normal-cdf** is a vectorized function and invoking it on a list of three numbers produces a list of three answers. Of course, if we do wish mean to behave in a vectorized fashion, the statement **(mapcar #'mean (list (normal-rand 10) (normal-rand 20)))** will do it.

Figure 2: A histogram and a Line Plot

A picture is worth a thousand words, particularly in statistics. Lisp-Stat boasts excellent graphical tools. The graphical system is based on an object-oriented paradigm. Functions that create graphical windows or plots return an object as the result. The returned object is just another data type much like a number or a list and it can be used in appropriate computations.

Commonly used graphical functions are **histogram** for constructing histograms, **plot-points** for plotting *(x,y)* pairs, **plot-lines** for joining *(x,y)* pairs by means of lines, **plot-function** for plotting a function of one variable, **spin-function** for plotting a function of two variables, and **spin-plot** for 3-d plots.

All the **spin** functions provide controls for yawing, pitching and rolling in the graph they create. Figure 2 shows a plot of the number of requests versus each of the *24* hours. The plots were produced using the following lines of code.

```
(histogram requests)
(def time (iseq 24))
(plot-lines time requests)
(send * :add-points time requests)
(send ** :point-symbol (iseq 24) 'diamond)
```

Just drawing the lines alone is less than satisfactory since the exact location of the points is lost. So, after constructing the plot, we send a "message" to the plot using the **send** function asking the object to add-points to the graph resulting in the graph shown. The ***** in the send function refers to the result of the previous command, i.e., the plot object. The ****** refers to the result of the command before the previous one. In the example, I have asked that the plotting symbol be a diamond instead of the default circle. The user has a choice of quite a few plotting symbols.

In each plot there is a menu button that has further useful options. One can select or deselect points with the mouse, highlight certain points, save the plot as a postscript file etc. I will only discuss a single feature, that of *linking*. Linked plots are a way of sharing information between plots. Consider for example, figure 2, where we have a plot of **requests** versus **time** as well as a histogram of **requests**.

If you enable linking by choosing the **Link View** item in the menu in each plot, selecting a vertical bar in the histogram by dragging the mouse with the button pressed causes the corresponding points in the line plot to be highlighted. You might have to peer at the figure to see that the point where the highest peak occurs is highlighted since it corresponds to the highlighted histogram bar. Linking is extremely useful in viewing multidimensional data since one can get a better idea of how the same group of points can be projected in different views.

Online documentation for Lisp-Stat is available via the functions **help**, **help*** and **apropos**. For help on the **mean** function, type **(help 'mean)**. The use of the quote is essential, otherwise the interpreter would assume that mean is a variable and try to evaluate it. However, in many situations, one does not know what the function is named.

For example, is the function that multiples two matrices **mat-mult** or **matrix-multiply**? Typing **(apropos 'mult)** will print a list of all symbols that have the word "mult" in them. This might help you narrow down the search. On the other hand, if you know that the function you are looking for contains the word **matrix** in it, **(help* 'matrix)** will return help on all symbols that contain the word **matrix**. The help facility as it exists now is less than optimal and several people are developing a more elaborate help system.

I usually read the newsgroup sci.stat.math and almost always there is someone out there who wants to know how to calculate an *F*-probability or how to generate a normal random variable. Lisp-Stat has distribution functions and generators for all of the commonly used distributions. For example **(normal-cdf 1.645)** will give you the probability to the left of *1.645* which is about *0.95*. The statement**(def x (normal-rand 100))** will define x to be a list of *100* standard normal variates. Similar functions exist for Students-*T*, Gamma, Beta, Chi-squared and *F* distributions as **(help* 'cdf)** or **(help* 'rand)** will show.

Lisp-Stat has many functions for input and output. For dealing with files, I've rarely needed to go beyond using the two functions **read-data-file** and **read-data-columns**. The statement **(read-data-file "foo.dat")** returns the whole file contents as one long list, while **(read-data-columns "foo.dat")** returns a list of columns of the file. One can specify the number of columns in the data file as a second argument to **read-data-columns**. Otherwise, it guesses the number of columns based on the first line. The function **format**, which is similar to C's **sprintf()**, is a versatile function for formatted printing.

Programming in Lisp-Stat

Lisp-Stat's graphical system and regression models are implemented using a prototype-based object system. This is different from the class-based object system used by languages like C++ or the approach used by Common Lisp Object System (CLOS). Briefly speaking, there is a root prototype object from which instances of all other objects are created. Objects can have slots to hold information and they respond to *messages* which are dispatched to the object using the **send** function. Messages are typically keywords, words that begin with a colon–**:add-points** in figure 2 is an example.

The code that actually implements the action is called a *method* for the message. The macros **defproto** and **defmeth** make the process of constructing objects and writing methods easier. Lisp-Stat would be less interesting if all it provided were objects for building statistical models. The windowing system provides objects for building user interfaces like menus, dialogs, slider controls etc. So one can construct nice dialogs to go with the computations.

Figure 3: A 2-D Plot with a Least Squares Line Superimposed

A Simple Animation

Figure 3 shows an example of dynamic animation using a slider dialog. The function *sin2pi x/n* is plotted. The slider allows the user to see the plot change as *n* is changed. The code to perform this is below.

```
(setf n 1)
(defun f (x) (sin (/ (* 2 pi x) n)))
(def sine-plot (plot-function #'f -5 5))
(defun change-n (x)
  (setf n x)
  (send sine-plot :clear :draw nil)
  (send sine-plot :add-function #'f -5 5))
(sequence-slider-dialog (iseq 1 20) :action #'change-n)
```

The function **sequence-slider-dialog** creates a slider. Initially, the global variable *n* is *1*. Every time the user moves the slider-stop using the mouse, the function **change-n** gets called with the value of *n* corresponding to the slider-stop. In our example, *n* can be any integer from *1* to *20*. The function **change-n** sets the value of *n* and redraws the plot.

Figure 4: A 2-D plot with a Least Squares Line Superimposed

An object-oriented Programming Example

In order to keep the discussion tolerable, I chose a simple example that is probably not too useful. For serious programming, one needs to know about the built-in prototypes and functions of Lisp-Stat discussed in Tierney's book. I shall introduce what I need as I go along.

We will create an object that accepts a list of *(x,y)* values and draws a plot with the least-squares line superimposed on it. We will also require that the equation of the least-squares line be displayed in the plot. We begin by defining a new prototype. It is only natural that our prototype be a descendent of the built-in prototype **scatterplot-proto** which “knows” all about drawing 2D plots.

```
(defproto least-squares-plot-proto '(intercept slope) ()
          scatterplot-proto)
```

Notice that our prototype has two slots for holding the intercept and the slope of the least squares line. We will need to access the values in these slots later, so it is best to define two simple methods using the **defmeth** macro that return the slot values.

```
(defmeth least-squares-plot-proto :slope ()
 "Returns the slope of the least squares line."
  (slot-value 'slope))
(defmeth least-squares-plot-proto :intercept ()
```

```
  "Returns the intercept of the least squares line."
   (slot-value 'intercept))
```

We have provided a documentation string for the methods; the documentation can be retrieved by means of a command such as **(send least-squares-plot-proto :help :slope)**.

In order to use our prototype, we must define a **:isnew** method that initializes an instance of the prototype. Our **:isnew** method must calculate the least-squares line and store the slope and intercept. It should exploit its lineage as a descendant of **scatterplot-proto** by invoking the inherited methods to do the plotting tasks. Some space must be created in the margin to display the equation for the least-squares line.

Finally, the *x,y* points must be plotted, the axes labeled, and the window redrawn to reflect the changes. Here is the method.

```
(defmeth least-squares-plot-proto :isnew (x y &key (title "LS Plot"))
  (let* ((m (regression-model x y :print nil))
         (beta (send m :coef-estimates)))
    (setf (slot-value 'intercept) (select beta 0))
    (setf (slot-value 'slope) (select beta 1)))
  (call-next-method 2 :title title)
  (send self :margin 0 (+ (send self :text-ascent)
                          (send self :text-descent)) 0 0)
  (send self :add-points x y)
  (send self :variable-label 0 "X")
  (send self :variable-label 1 "Y")
  (send self :redraw))
```

We have used the **regression-model** function to compute the least-squares line. The **call-next-method** function calls the **:isnew** inherited method of **scatterplot-proto**–this is what actually creates a plot-window. The argument *2* just refers to the number of variables that will be plotted. At this point, the plot-window is actually blank. Using information about the font in use, a margin area is created. Then the points are plotted. In the body of a method the variable **self** is bound to the object receiving the message. The method concludes by giving some meaningful names to the variables and redrawing the window.

All the above code will do is plot the points. How can we ensure that least-squares line and its equation are also displayed? We use the fact that any window is actually drawn using a **:redraw** message. By writing a new **:redraw** message, we can ensure the results we want. In actuality, the **:redraw** message itself is executed via three other messages **:redraw-background**, **:redraw-content** and **:redraw-overlays**. We really only need to write a **:redraw-content** method since only the content of the plot is affected. So here we go.

```
(defmeth least-squares-plot-proto :redraw-content ()
  (call-next-method)  ; Let the scatterplot do its things.
  (send self :adjust-to-data :draw nil) ; make sure scale is ok.
  (let* ((limits (send self :range 0))
         (intercept (send self :intercept))
         (slope (send self :slope))
```

```
            (info-str (format nil "y = ~5,3f + ~5,3f x" intercept slope)))
      (send self :draw-string info-str
            10 (+ (send self :text-ascent) (send self :text-descent)))
            ; Display the equation in the margin.
      (send self :add-function ; Draw the LS line.
            #'(lambda (x) (+ intercept (* slope x)))
            (car limits)
            (cadr limits) :draw nil)))
```

Notice that the keyword argument **:draw** is **nil** to avoid infinite loops in the redrawing process. If **:draw** is not nil, the **:redraw** method gets invoked again. The line is actually drawn using the **:add-function** method of **scatterplot-proto**. We need not worry about drawing the points since that is the responsibility of **scatterplot-proto** once we have added the points in the **:isnew** method.

Figure 4 shows the results of using this code with the following program.

```
(def x (normal-rand 20))
(def y (+ 5 (* 2 x) (normal-rand 20)))
(def m (send least-squares-plot-proto :new x y))
```

Final Remarks

Compiling a Lisp-Stat program is straightforward. The statement **(compile-file "foo")** in Lisp-Stat will compile the file foo into foo.fsl. When you load the file foo later, the compiled file is loaded if it exists and is newer than the uncompiled file. Debugging can be accomplished via the **debug**, **baktrace** and **trace** functions. A stepper is also available to step through lines of code.

There are many interesting dynamic animations can be constructed in Lisp-Stat. This article has only scratched the surface. Lisp-Stat continues to evolve and **Xlisp** itself continues to move closer and closer to Common Lisp due to the efforts of many, particularly Tom Almy and Luke Tierney. The available body of applications and software for Lisp-Stat is also growing; see the sidebar “Getting Lisp-Stat” for more information.

> Balasubramanian Narasimhan teaches Statistics at Penn State Erie, The Behrend College. His interests include classical western music, Seminole football and the history of India. He may be reached at naras@euler.bd.psu.edu

Getting Lisp-Stat

Lisp-Stat is freely available on the net. The primary distribution site is ftp.stat.umn.edu. Look under pub/xlispstat for xlispstat-3-44.tar.gz. The file is about 1.2 Megabytes, which means that it fits nicely on a 3.5-inch disk. It compiles out of the box on Linux, but to use the foreign-function interface, you must first install the GNU **dld** library, available from tsx-11.mit.edu under pub/linux/binaries/libs as dld-3.2.5.bin.tar.gz. For those who don't want the adventure of building from scratch, you can obtain a binary from

euler.bd.psu.edu under pub/lj/xlispstat. Follow the instructions in the README file. The file xlispstat-3.44-bin.tar.gz is the whole binary.

There is a mailing list for Xlisp-Stat users. To join the mailing list send a message with your e-mail address saying that you want to subscribe to stat-lisp-news-request@stat.umn.edu.

The Usenet newsgroup comp.lang.lisp.x is devoted to XLisp, however it is a low-volume newsgroup averaging 2-3 articles a day.

The site ftp.stat.ucla.edu has a good amount of Lisp and Lisp-Stat related stuff. For a look at some hypertext applications, look at the sites euler.bd.psu.edu and www.stat.ucla.edu.

Archive Index Issue Table of Contents

Advanced search

Linux in the Rugged Field

Sid Hellman

Issue #16, August 1995

Linux makes field seismology simpler, less expensive, and more efficient. Sid Hellman explains why having the source available helped make Linux the best choice for a portable workstation for field data analysis.

Traditionally, seismologists have used laboratory-based Sun computers to analyze data recorded in the field. But the advent of inexpensive Intel-based laptops and Linux has allowed us to take workstations out of the lab and into the field. This has involved the routine porting of code across OS platforms as well as kernel level development. After a year of hard work, scientists affiliated with the PASSCAL Instrument Center have been using Linux as a portable workstation for seismic studies from Missouri to Massachusetts and from Micronesia to Tanzania.

Contrary to the implications of our name, we are not a group dedicated to or affiliated with the Pascal programming language. PASSCAL, or the Program for Array Seismic Studies of the Continental Lithosphere, is a member program of IRIS (the Incorporated Research Institutions for Seismology), which is a consortium of 90 U.S. universities (there are a number of affiliate members, both foreign and domestic). We have been located at Columbia University's Lamont-Doherty Earth Observatory since 1989, while a second instrument center was established at Stanford University in 1992.

Kobe Earthquake Seismic Data Displayed on a Linux System

In seismology, it is common for a scientist to run an experiment requiring the use of many expensive instruments for a relatively short period of time. Having individual scientists spend hundreds of thousands of dollars on equipment, only to have it collect dust on a shelf for 80% of its life, makes little sense in this era of fiscal belt-tightening. In addition, few scientists have the time to master and integrate equipment from a variety of manufacturers into a reliable data acquisition system.

PASSCAL Team Member Bob Busby in Pakistan

Therefore, the PASSCAL Instrument Center was organized to service the seismic community—providing a powerful and flexible portable instrument along with expertise that can only be obtained via experience. We supply logistic and technical support, provide for system integration and maintenance, and deliver comprehensive feedback to the manufacturers. Since experiments are returning from the field with Terra-Bytes of data, an extensive suite of programs have been developed to assist the seismologists with the acquisition and archiving of data and for assuring data quality. The popularity of this program has grown to the point that experiment scheduling for 1997 is currently underway.

Historically PASSCAL experiments have utilized two separate computer systems, one for data acquisition and one for data analysis. The data acquisition system (DAS) is comprised of a series of weather-tight grey boxes (CPU, SCSI disks, batteries), peripherals (solar panels, GPS receiver), and a hand terminal with an LCD touch-screen to program the DAS and retrieve data. A standard Sparcstation running SunOS, which may or may not be located anywhere near the seismic station, is used for data analysis and quality assurance.

The portable aspect of the PASSCAL instrument explains the program's popularity, at the same time it also introduces logistic difficulties. Investigators frequently erect stations in remote locations, making access difficult and/or expensive. Hence station visits are infrequent, with users allowing the DAS to run for months before they return for data retrieval. Retrieval is accomplished by copying to tape (or spare disk), which is then brought back to the lab for quality control, data analysis and archiving. Detecting problems with the instrument at this stage requires a return trip. (At one of our stations in Micronesia, there is only one flight a week to the island. You fly in, work for 4 hours, wait a week, fly back, check your data, and, if there were any problems, fly back a week later.)

In November of 1993, I installed Linux at home simply to have a operating environment similar to the one I used at work. I got X up and running and discovered the XView libraries sitting there serendipitously on my hard disk. One major piece of code for which I was responsible uses X (via XView) for displaying seismic traces, and approximately 17 milliseconds later I decided to port this program to Linux, just so I could work at home.

One day I dragged my PC into work to show everyone what Linux could do. A member of our team remarked that if we could run this on a portable computer, we could actually look at the raw data and perform spectral analyses

of the data in the field, while there is still time to correct mistakes. That led to the thought that if we had a computer with X-windows in the field, we could even write some user-friendly and powerful software to program the DAS, instead of using the hand-terminal.

At the time we were actually experiencing a dilemma concerning the programming of the DAS's. Epson manufactures the hand terminal, which runs a simple basic program and interfaces to the DAS via a serial port. This hand terminal is fairly user-friendly: Just point with your finger and push. The problem is that only rudimentary diagnoses can be made with the hand terminal. There is a DOS-based alternative to this program that is more powerful (i.e. has many options), but “user friendly” is not a phrase we use to describe it.

Since many of our users only perform field work every few years, and many are novice grad students, user friendliness is of utmost importance to guarantee data integrity. The idea of combining the power of the DOS program with the ease of use of the hand terminal into a beautiful X-windows program was compelling.

With a dream and a need, we did what many of you have done: We went looking for money. Selling the laptop idea was actually quite easy; selling Linux was another matter. Since all of our existing code was written to run under SunOS and X Windows we did not even entertain the idea of porting our code to MS-Windows or OS2.

However I was not actually sold on Linux, myself. My first PC-Unix experience was with Interactive Unix, back when it was owned by Kodak. I really had no objection to going with them again (or SCO, or whomever). We investigated the alternatives, and as many of you have also discovered: chances are good that Linux provides better support than the other PC-Unixes for any given piece of hardware. After demonstrating that to ourselves and our boss, we went with Linux.

We purchased our first two laptops in early 1994, primarily to study the feasibility of running Linux laptops in the field. When my co-workers Paul Friberg and John Webber and I began porting major pieces of our code, we hit our first bottleneck in very short time: SCSI. All of our data is acquired and stored on SCSI devices, and much of our code required data just to be tested. Our quick-fix was to purchase a docking station and install a SCSI adapter here. Obviously this was neither a permanent nor portable solution, but it was enough to get us started.

We looked into PCMCIA SCSI, but no support existed at that time. We reasoned that with the infancy of the technology, support could not be expected immediately, but would be coming along shortly. Foolishly, we decided to wait for it while proceeding with the development of all the other code we would need.

Because we are also heavily entrenched in Tcl/Tk, we decided to write a Tcl/Tk-based replacement for the hand terminal. We managed to combine the hand terminal's point-and-click ease of use with the functionality of the DOS program. The BLT extensions to Tcl/Tk made this program downright fancy.

We also wrote a program to display seismic data in real time, allowing users to examine the background signal of the seismometer before walking away and leaving it for a few months. (The alternative has always been to acquire data to disk, run some data conversion programs, load up the data into the viewer, make changes to the instrument and repeat the procedure. This, of course, assumes that you have a workstation near the seismometer.) The new option of acquiring and displaying in real-time while making adjustments can easily save time and effort.

At this point we had assembled a sizeable suite of programs, and all that remained was getting the data. Again we asked around the net, but no-one knew of a Linux supported PCMCIA SCSI card. Gennady Pratusevich, our engineer, decided to write a driver for Linux, and we decided to go with the Trantor/Adaptec card. Since most of the other Adaptec cards were well-supported, we figured that Adaptec would provide us with the programming information. We were quite wrong.

We started searching around for SCSI cards from companies that would supply us with programming information. We found a small producer of such cards in Colorado, initially he seemed interested in Linux, but he never got back to us again. We considered developing our own card when some Linux newsgroup wisdom indicated that the New Media Bus Toaster used the AIC6360 chipset. This is the same chipset used by the Adaptec 1522 board. We purchased a Bus Toaster and, armed with the source code for the Adaptec 1522 driver, David Hind's PCMCIA code, and a few assorted manuals, Pratusevich managed to create a fully functional SCSI driver. With the subsequent introduction of loadable SCSI modules, the real power of PCMCIA was realized, allowing us to swap disks on the fly. We were finally ready for prime time.

I would like to point out here that if we had chosen an operating system that did not release the source code, this would have been the tragic end of our story.

Last December we attended the fall meeting of the American Geophysical Union in San Francisco, having just taken delivery of 5 additional laptops. Usually we have our Sun Sparcstations on prominent display, but this year they were hidden behind our Linux laptops (as if a full-sized workstation could hide behind a 7-pound laptop). Linux was a hit, and everyone wanted one.

The second week of January saw us shipping four laptops into the field. Preliminary reports are encouraging. The laptops work and are making life easier for our users. The people using these machines are not Linux experts, but the machines are easy enough to use that people would rather utilize these than any currently available DOS alternative.

We did not originally expect Linux to be the way to go for us. Obviously porting to a Sparc laptop would have required much less work, but they are prohibitively expensive. We also looked into Solaris for PC's, but the minimal volume of hardware that Solaris supports turned us away. If any of you are trying to convince your supervisors to go with Linux, first sell the idea of Unix, and then have them compare Linux to Solaris and the like. The best decision becomes obvious at this point.

> The work described in this article was supported by the Incorporated Research Institutions for Seismology and by the National Science Foundation under Cooperative Agreement No. EAR-9023505. Any opinions and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Shortly after **Sid Hellman** earned his masters in physics, a knee injury diverted him into programming. A Systems Analyst at Columbia University's Lamont-Doherty Earth Observatory since 1990 (seismology since 1993), his duties have ranged from programming to electronics to field work. The PASSCAL Instrument Center can be reached at passcal@ldeo.columbia.edu or via the WWW at www.ldeo.columbia.edu/Passcal/passcal.intro.html

Archive Index Issue Table of Contents

Advanced search

Linux Programing Hints

Jim Shapiro

Issue #16, August 1995

Perl is often considered a scripting language for systems administrators. Jim demonstrates that it is useful to applications and scientific programmers as well—as a prototyping tool.

If you are like many Linux users you may have heard of Perl, but have been reluctant to learn another language. This was my situation several months ago. A friend suggested I give Perl a try. Since I already knew C, Perl was a snap to learn. I soon found myself doing all sorts of text reformatting using Perl. My friends and coworkers were impressed, but skeptical. Could Perl cut the mustard on big files where a ton of data had to be read, massaged and written?

Their skepticism subsided, however, when I wrote a Perl program (I prefer to call them programs rather than scripts just to separate them from shell scripts) to load over three and a half million lines of US Postal data. Now I actually teach Perl (but I digress).

In this article I would like to suggest a use for Perl which is often overlooked—Perl as a prototyping tool. Most C programmers spend a fair amount of time managing memory, and I am no exception. Memory management is a necessary function, especially if you want to keep your C programs tight—not using more memory than necessary—and well behaved—not crashing with the resulting core dumps. The problem with managing memory yourself is that it can divert attention from the program's purpose, which is typically to get an algorithm running.

With Perl, not only do you get solid memory handling routines, you get them in an interpreted/compiled environment and, thanks to the Free Software Foundation, they will not cost you a penny. In fact, if you have Linux you probably already have Perl as well. Let me illustrate how Perl can be used as a prototyping tool with two examples, a simple Monte Carlo calculation and a more substantial geometric problem.

A Monte Carlo Estimate Of pi

Monte Carlo techniques use probabilistic methods to make estimates. Typically, one or more random numbers are substituted into a function and the resulting value is tested for validity. The program keeps track of both the number of satisfactory tests and the total number of tests. The result is the ratio of satisfactory to total tests and this ratio is monitored as the number of tests is increased.

```
#!/usr/bin/perl
# Monte Carlo calculation of pi
srand(time | $$);
for($i = 1, $inside = 0.0; $i <= 1000000; $i++)
   {
   $x = rand;
   $y = rand;
   $inside++ if $x * $x + $y * $y < 1.0;
   printf "After %7d points pi ~= %9.7lf\n", $i, 4.0 * $inside / $i if
      ($i % 10000) == 0;
   }
```

Figure 1. A monte Carlo Calculation of pi

Let us get our feet wet with a short and simple Perl program. Figure 1 is a program which estimates pi using a Monte Carlo calculation. Consider a circle inscribed inside a square of side two centered at the origin. The area of the square is four and, since the circle has a radius of one, its area is pi. The ratio of the area of the circle to that of the square is thus pi / 4, and this is also the chance that a random point within the square is also inside the circle. This program repeats a loop a million times, each time calling Perl's **rand** function twice, once for **x** and once for **y**. The distance of the **x**, **y** pair from the origin is calculated and, if it is less than one, it is counted as a successful test. (Technically, this program uses only the parts of the circle and the square in the first quadrant for simplicity, but, by symmetry, the ratio of areas is the same as if the whole of both figures had been used.)

A Linux tip—if you give your Perl programs a unique extension, like .pl, it is easy to make them stand out in ls type listings. Adding the line:

```
.pl 01;33 # perl programs (yellow)
```

to the file /etc/DIR_COLORS will make the names of all files with .pl extensions in listings appear in yellow. See the man pages for **ls** and the file /etc/DIR_COLORS for details.

Note, first of all, how short the Perl program is. Also, note how much it resembles a C program, especially the **for** loop and the **printf** function. There are important differences, however. Variables, like **$i**, **$x**, etc. are used when needed without a specific declaration. It is not even clear to the casual observer what the types of the variables are. And what is with these if statements **after**

the statements they control? And **rand** is used like a function, but there are no parentheses—maybe it is a keyword.

All of these differences are features of Perl. Perl keeps track of your variables for you. The variables are really strings internally, but they get converted to doubles when needed such as when the distance of the random point from the origin is calculated and compared to one. You needn't concern yourself with any of these details, however. The **if** test at the end of a line is a handy equivalent to C's **if** block (the C style is OK in Perl also) but a lot shorter. You will find yourself using Perl's **if** style all the time once you get the hang of it. Perl's **if** has three relatives (**unless**, **while** and **until**) which also can be used before a block or at the end of a series of comma separated statements. Finally, **rand** really is a function—in this case the parentheses are optional. This is the case with many functions, including the **printf** at the bottom of the **for** loop.

If you run the program in Figure 1, you will get an estimate for pi after every 10,000 tests. I call the program m_c.pl and run it by typing its name. The first line is the path to the Perl program on my system. Change this path if yours is different. You can also test a program for syntax errors with a **-c** command line switch, i.e.

```
perl -c m_c.pl
```

Perl will also provide warnings, such as when you assign to a variable and never use the variable again. Use the **-w** command line switch to turn on warnings. This is a handy way to uncover spelling errors which can easily crop up in an environment without explicit variable declarations. I usually use both tests simultaneously during development, i.e.

```
perl -cw m_c.pl
```

Point In Polygon

Next let us consider a more difficult problem. How do we test whether a point is inside a polygon? This problem is not as simple as it may first appear, especially when you take into account the special cases—such as when the point is on the boundary of the polygon. Is that inside, outside, or do you want to count it as a separate case—on the boundary? Let's break the problem down and start by writing a Perl subroutine to test whether a point is on a line.

A not-too-efficient but easy-to-code routine works as follows. If a point is on a line the sum of the distances from the point to each of the line endpoints is the same as the distance between the endpoints, i.e. the length of the line. You might want to reread the previous sentence and make a little sketch to convince yourself that this is indeed the case. We will first need a routine (in

Perl they are called subroutines) to find the distance between two points. C has a built-in function, hypot, but the one-liner in Figure 2 is the Perl equivalent. The **sub** keyword denotes a subroutine and the subroutine's name follows. There is no parameter list in the definition of a subroutine. Those are comments following the subroutine name in Figure 2. We will supply the parameter list when we call the subroutine, as you will see shortly.

```
sub hypot # (x, y) returns sqrt(x * x + y * y)
{
sqrt $_[0] * $_[0] + $_[1] * $_[1];
}
```

Figure 2. Perl Subroutine to Find Distance Between Points

When you call a subroutine in Perl all of the parameters are automatically put into an array **@_** (The **@** denotes a standard array). In this case the array has two elements, the differences in the **x** and **y** coordinates of the two points. Note that the elements are referenced by address so that we need to be careful. Any modifications to these variables would change the values outside this subroutine. In Perl you can also create local variables within a subroutine, or inside of any block for that matter. Normally I would do so, but since this subroutine is so short and is likely to be called many times, I avoided the overhead of local variables. Note again, that a Perl function, **sqrt**, was called without any parentheses. The return value of a subroutine is the last thing calculated (or you can overrride this behavior with a specific return value). Here the Pythagorean result is the last value calculated, so no explicit **return** is necessary. You will probably find, as I have, that explicit **return** statements are rarely needed.

Now we need to tell our program what points and areas are. In C we seem to have the advantage of structures at our disposal. We would probably set up something like the **typedef** and declaration in Figure 3.

```
typedef struct
   {
   double x,
          y;
   } POINT;
POINT *polygon_p;
```

Figure 3. Typedef and Declaration

A point has **x** and **y** coordinates and a polygon has three or more points. Every time we need a polygon we can malloc the space for it and fill it with points—and free the space when we are done with it. This is where Perl shines.

Perl has neither explicit memory allocation nor structures. The good news is that we do not have to concern ourselves with memory—it is there when we need it. The bad news is that we have to come up with some system like C's

structures. This turns out to be trivial. Let us put everything into strings. Perl takes care of strings of any length, relieving us of the pesky problem of counting characters. You do not even have to add one for the terminating null character—Perl does not use a terminator.

Our point will simply be a string containing two doubles joined by a comma; our line will be a string containing two points joined by a colon; and our polygon will be a string containing three or more points joined by colons, like so:

```
$point = "1.0,2.0";
$line = "0.0,0.0:0.0,1.0";
$polygon = "1.0,2.0:6.0,5.0:0.0,3.1";
```

These “structures” turn out to be easy to scan visually and very easy to manage, thanks to Perl's join and split functions. Figure 4 is the **point_on_line** subroutine I developed using the above scheme.

```
$epsilon = 1.0e-10;
sub point_on_line #(point, line) returns 0 or 1
{ # if on - sum of distance to ends should be distance between ends
local($point, $line) = @_;
local($p_x, $p_y) = split(",", $point);
local($l_b, $l_e) = split(":", $line);
local($l_b_x, $l_b_y) = split(",", $l_b);
local($l_e_x, $l_e_y) = split(",", $l_e);
&fabs(&hypot($p_x - $l_b_x, $p_y - $l_b_y) +
      &hypot($p_x - $l_e_x, $p_y - $l_e_y) -
      &hypot($l_e_x - $l_b_x, $l_e_y - $l_b_y)) < $epsilon;
}
sub fabs # (x) returns absolute value of x
{
local($rv) = @_;
$rv = -$rv if $rv < 0.0;
$rv;
}
```

Figure 4. point-on-line Subroutine

This subroutine is called with two strings, i.e.

```
&point_on_line($point, $line);
```

and returns one if the point is on the line, and zero otherwise. Note that in this case I put the calling parameters into local arrays for safety and ease of maintenance—**$point** and **$line** are easier to remember next week than **$_[0]** and **$_[1]**. The **split** function's use should be obvious. The **point** string gets split into two coordinate strings by the comma, and the **line** string gets broken into two point strings by the colon. Then the points at the beginning and end of the line, **$l_b** and **$l_e**, get broken down into their respective coordinates. Finally the coordinates are passed to our **hypot** function. Note how the **&** sign is used to prefix a subroutine for the call. Perl has no equivalent to C's **fabs** so I quickly

rolled my own in Figure 4. I used the **$epsilon** variable to avoid floating point roundoff problems.

We have been building this thing from the bottom up. So far we have a routine to test whether a point is on a line. Our goal is a routine to test whether a point is within a polygon. Let me give you an overview of how we are going to solve the problem, provide you with two more Perl routines (which you should be able to read now), and then show you the external “workhorse” routine that does the final inside/on/outside determination.

Our polygon is a closed figure and every point is either inside, outside, or on the boundary (there is actually a mathematical statement called Jordan's Curve Theorem that proves this!). If we can establish a point that is guaranteed to be outside of our polygon, we can test the “insideness” of any point by connecting this test point to our “outside” point by a straight line and counting the number of times the line crosses the polygon. Clearly if the line never crosses the polygon, the point is outside; if it crosses once the point is inside; and, in general, if the line crosses the polygon's boundary an odd number of times, the point is inside. An even number of crossings means the point is outside. To put it another way, every time the line crosses the polygon it changes from being either inside to outside or vice-versa. Starting from the outside point our first crossing (if any) puts us inside; the next crossing (if any) puts us back outside; etc., odd-inside, even-outside. So, it looks like we will need a routine to test whether two lines intersect.

One way of testing for line intersection is to make sure both endpoints of each line are on opposite sides of the other line. (Reread and make a few pictures, as before.) This leads to our final routine. Now we will test whether, when we walk along a line from the beginning point to the opposite end and then turn to go straight to another point, we turn counterclockwise or clockwise. This discussion and the resulting routines are modeled after Sedgewick's in “Algorithms in C”, pages 349-354 (Robert Sedgewick, Addison Wesley, 1990). Let us start with the counterclockwise subroutine, **ccw** in Figure 5.

```
sub ccw # (three points) return -1, 0, or 1
{
local(@points) = @_;
local($rv) = 0;
local($dx1, $dx2, $dy1, $dy2, $p0x, $p0y, $p1x, $p1y, $p2x, $p2y);
($p0x, $p0y) = split(",", $points[0]);
($p1x, $p1y) = split(",", $points[1]);
($p2x, $p2y) = split(",", $points[2]);
$dx1 = $p1x - $p0x;
$dy1 = $p1y - $p0y;
$dx2 = $p2x - $p0x;
$dy2 = $p2y - $p0y;
switch:
   {
   $rv =  1, last if $dx1 * $dy2 > $dy1 * $dx2;
   $rv = -1, last if $dx1 * $dy2 < $dy1 * $dx2;
   $rv = -1, last if ($dx1 * $dx2 < 0.0) || ($dy1 * $dy2 < 0.0);
   $rv =  1, last if ($dx1 * $dx1 + $dy1 * $dy1) < ($dx2 * $dx2 + $dy2 * $dy2);
```

```
    }
$rv;
}
```

Figure 5. Counterclockwise (ccw Subroutine)

The **ccw** routine accepts three points and compares the slope of the line from the second to the third with the slope of the line from the first to the second. It is carefully constructed to handle vertical lines and even the case of collinear points. Note how the input points are collected into a local array, **@points**, avoiding side effects and making the program easy to maintain and understand. The points each get split into their respective coordinates with the test being carried out in a block labeled **switch** (for obvious reasons). By now you have probably guessed that Perl allows labels, just like C. There is no specific **switch** construct in Perl, however. The block in **ccw** above is Perl's way of building a switch. The **last** keyword immediately exits the block and sometimes the Perl interpretor is smart enough to convert the block to a C switch statement internally (although not in this case).

```
sub intersect # (two lines) returns 0 or 1
{
local($l1, $l2) = @_;
local($l1_b, $l1_e) = split(":", $l1);
local($l2_b, $l2_e) = split(":", $l2);
&ccw($l1_b, $l1_e, $l2_b) * &ccw($l1_b, $l1_e, $l2_e) <= 0 &&
&ccw($l2_b, $l2_e, $l1_b) * &ccw($l2_b, $l2_e, $l1_e) <= 0;
}
```

Figure 6. Intersection Subroutine

Figure 6 is the intersection routine. It is very straightforward. The lines are split into endpoints and the endpoints are tested for “sideness”.

We now have all the building blocks to construct our “inside” subroutine. First we connect our test point with an “outside” point via a straight line. Then we walk around the polygon, testing each polygon side in turn, accumulating the intersections of the sides with the test line. If the number of intersections is odd the point is inside, and vice-versa.

```
$big_float = 1.0e7;
sub lower_left_index # (polygon) returns index of lower left corner
{
local($polygon) = @_;
local($index) = 0;
local(@vertices) = split(":", $polygon);
local($x_min, $y_min) = split(",", $vertices[0]);
local($i, $x, $y);
for($i = 1; $i <= $#vertices; $i++)
   {
   ($x, $y) = split(",", $vertices[$i]);
   if(($y < $y_min) || (($y == $y_min) && ($x < $x_min)))
      {
      $x_min = $x;
      $y_min = $y;
      $index = $i;
      }
   }
```

```
$index;
}
sub inside # (point, polygon) returns 0 or 1
{
local($point, $polygon) = @_;
local(@vertices) = split(":", $polygon);
local($index) = &lower_left_index($polygon);
local($last_index) = $index ? $index - 1 : $#vertices;
local($count, $holding_point) = (0, 0);
local($i, $lp, $lt, $vertex, $x, $y, $big_x_point);
local($check_index) = $index; # true if index is not zero
OUTER: for(;;)
   { # one pass loop
   for($i = 0, $vertex = $vertices[$#vertices]; $i <= $#vertices; $i++)
      {
      $lp = join(":", $vertex, $vertices[$i]);
      $vertex = $vertices[$i];
      if(&point_on_line_2($point, $lp))
         {
         $count = 1;
         print "Point on boundary\n" if defined $verbose;
         last OUTER;
         }
      }
   ($x, $y) = split(",", $point);
   $big_x_point = join(",", $big_float, $y);
   $lt = join(":", $point, $big_x_point);
   for($i = 0; $i <= $#vertices; $i++)
      {
      if(&point_on_line_2($vertices[$index], $lt))
         {
         $holding_point = 1;
         }
      else
         {
         if(!$holding_point)
            {
            $lp = join(":", $vertices[$last_index], $vertices[$index]);
            $count++ if &intersect($lp, $lt);
            }
         elsif($holding_point &&
               (&ccw($point, $big_x_point, $vertices[$index]) !=
                &ccw($point, $big_x_point, $vertices[$last_index])))
            {
            $count++;
            }
         $last_index = $index;
         $holding_point = 0;
         }
      $index++;
      if($check_index && ($index == @vertices))
         {
         $check_index = 0;
         $index = 0;
         }
      }
   last;
   } # one pass "loop"
$count & 1;
}
```

Figure 7. lower-left-index and inside Subroutines

The pair of subroutines in Figure 7 are Perl implementations of functions suggested by Sedgewick in the section “Inclusion in a Polygon” (pages 353-355), although more complete. The **lower_left_index** subroutine returns the index of the polygon point with the smallest x coordinate among all points with the smallest y coordinate. This is necessary to account for the special cases when a polygon vertex is on the test line. Note how, in the third line within the block, the **@vertices** array is automatically constructed by splitting the **$polygon** string

with colons. Each element of the **@vertices** array is a pair of coordinates separated by a comma, one for each vertex. Whenever we need individual x, y pairs the split function is called, as we have seen before. This occurs before the **for** loop to initialize **$x_min** and **$y_min**, and inside the loop to generate a new test pair **$x**, **$y**. The upper limit in the loop variable **$i** is **$#vertices**, which is the index of the last member of **@vertices**. Perl automatically keeps one of these variables for each array. The last statement in this routine, **$index;** simply establishes the return value.

The inside subroutine is our "workhorse" function. It is admittedly fairly complicated, but you should be able to follow the logic if you have read this far. Here is some help. The variable **$index** is used to walk from vertex to vertex around the polygon starting from the index returned by **lower_left_index**. If this index is anything other than zero, it will need to be reset after the vertex with the largest index is encountered. The variable **$check_index** keeps track of both whether this resetting will be necessary and, if so, whether it has been done yet. The variable **$last_index** is the index of the last vertex that was not on the test line. Generally this is the index of the vertex "behind" **$index**.

The **OUTER** label takes advantage of one of Perl's handiest features, the ability to exit effortlessly from nested loops. You can do this in C if you like **goto**. In the present example the polygon sides are created via the **join** function using the colon as the separator. The sides are stored in **$lp**. If the test point is on a polygon edge, there is no need to test further, so the **OUTER** loop is exited immediately. Note that setting **$count** to one in this case is equivalent to counting the boundary as inside the polygon, since one is an odd number. It would be trivial to count the boundary as outside or even as a special case by modifying the if block containing the statement: **last OUTER;**

If the point is not on an edge, a horizontal line from the test point to an outside point with an x coordinate equal to **$big_float** is constructed and stored in **$lt**. The remainder of this function tests for either line intersection or "sideness" depending on whether the previous vertex was on the test line, incrementing **$count** as appropriate. The return value of the inside function is 1, if the point is inside, and 0 if the point is outside.

```
#!/usr/local/bin/perl
while(<DATA>)
   {
   chop;
   $polygon .= $_ . ":";
   }
chop $polygon;
for(;;)
   {
   print "Enter x and y separated by a comma (q to quit): ";
   chop($point = <STDIN>);
   last if $point =~ /^[qQ]/;
   print("No comma!  Try again.\n"), redo if $point !~ /,/;
   $point =~ s/ +//g;
```

```
    print "Checking point: $point\n";
    printf "%s\n", &inside($point, $polygon) ? "inside" : "outside";
    }
__END__
5.0,4.0
0.0,0.0
10.0,5.0
0.0,10.0
```

Figure 8. Driver Routine for Inside Subroutine

A simple driver routine for the inside subroutine is presented in Figure 8. This routine reads its data from the end of the perl program itself. Anything following the line:

```
__END__
```

is considered data and is accessed through the (automatically opened) **DATA** file handle. Two new operators introduced in this driver are "**.**" which concatenates two strings and "**.=**" which appends one string to another. That is, "**.=**" is to "**.**" as "**+=**" is to "**+**". The **chop** function removes the last character from each element of its argument list. Note how the line:

```
chop $polygon;
```

trims the final colon from the polygon string, so that it is a legitimate polygon. Replace my data with your own if you want to run this driver, but be sure to put a comma between the x and y coordinates.

Summary

You have seen two examples of how Perl can be used for prototyping. I hope that from these examples you have gained a feel for Perl's syntax. More importantly, I hope that you have seen how using Perl can free you from concentrating on programming specific details, like memory allocation. Instead, you can direct your efforts toward getting your algorithm up and running. I have discovered that, in many cases, the Perl prototype was sufficient for my purposes, saving me the time of coding the program in C or C++ at all!

When I do recode a prototyped algorithm from Perl to another language, I have found that it is easy to change gears. The logic is behind me, freeing me to concentrate on C specifics, memory allocation/deallocation, input/output, error reporting, etc.

My suggestion to the reader is to program a simple application in Perl and see for yourself how this very elegant and powerful language works. You may not save any time with the first program or two, but it will not be long before the benefits of Perl appear. If you feel ambitious, try writing a routine to replace my **point_on_line**. I mentioned earlier that my algorithm for testing whether a point

is on a line is not very efficient. Another, more efficient scheme, is to first check whether the point's x coordinate is within the x range of the line and, if so, whether the point's y coordinate satisfies the equation of the line. Vertical lines are special cases.

Among the many algorithms I have prototyped in Perl are LZW data compression (the same as used in the UNIX compress utility), RSA encryption, many matrix operations including eigenvalue/eigenvector determination and a code generator that outputs C code from a database. I even have a little program called “perls” that reads a database of perl programming tips and prints a random tip to the screen. [I can provide this program to The *Linux Journal* and/or its audience via Internet. Let me know if you are interested.]

[Yes, we are. We would like to put it on our web site, perhaps even in a cgi script.]

> Jim Shapiro is a consultant specializing in programming mathematical algorithms. He is presently developing a GIS system for a telecommunications company. When he isn't on his Linux system hacking away in C or Perl he can often be found on the squash courts. Jim is a founding member of LUGOR, the Linux User's Group Of the Rockies.

Perl Resources

Programming Perl by Larry Wall and Randal L. Schwartz, O'Reilly & Associates, Inc., 1992. If you are serious about learning Perl, this is the book to read. It is all here, including some very sophisticated examples. Not recommended for beginners, however.

Learning Perl by Randal L. Schwartz, O'Reilly & Associates, Inc., 1993. A tutorial divided into lesson sized chapters.

Teach Yourself Perl in 21 Days by David Till, SAMS publishing, 1995. My personal favorite. Looks more daunting (841 pages) than it is. I got so excited I read it in seven days. Read this one, then “Programming perl”, and you will soon be an expert.

The “man” pages. Not bad if you want to get the flavor of the language, but mine seem dated.

Archive Index Issue Table of Contents

Advanced search

PracTCL Programming Tips

Stephen Uhler

Issue #16, August 1995

This month, we inaugurate a bimonthly column for Tcl/Tk programmers. Stephen Uhler will cover some useful but perhaps poorly-known or poorly-understood features of the Tcl language and the Tk windowing toolkit.

For those new to the Tcl language, the name of this column might be confusing: Tcl is pronounced “tickle”. This column will be mainly for those who already know Tcl and Tk and wish to learn how to improve their programming skills. For readers who would like to learn about Tcl and Tck, they are amply covered in John K. Ousterhout's book *Tcl and the Tk Toolkit*, and *Linux Journal* printed an introductory article in the December 1994 issue as well.

User interfaces are created in Tk with two steps. First, the user interface elements (called widgets) such as buttons or scollbars are created using the appropriate widget commands. Next, they are arranged on the display with a geometry manager.

Tk comes with several different geometry managers to choose from. The “packer” and “placer” are general purpose geometry managers, whereas the “text” (in tk4.0) and “canvas” widgets can also operate as geometry managers by positioning other widgets inside themselves. Additional geometry managers are available in some of the many extension packages, such as “blt_table” from the BLT extensions by George Howlett.

Traditionally, the “packer” is called upon as the primary geometry manager in Tk applications, because of the powerful constraint based layouts it supports, whereas the “placer” is reserved for beginners who have not yet mastered the packer's intricacies. John Ousterhout, in *Tcl and the Tk Toolkit* spends 15 pages describing the “packer”, but a single paragraph on “place”, suggesting “the placer is only used for a few special purposes”. In fact, the “placer” is an essential tool for the power user because it affords exact control over widget

positioning. I'll demonstrate two example uses of the “placer” that hint at its true power.

How to be a Pane in Tk

For the first example, we'll write a special purpose geometry manager entirely in TCL, using the “placer”, to construct a Motif-like “pane” widget. The “pane” widget divides a window into two halves, or panes, and provides a handle the user can use to dynamically change the relative size of each half.

```
frame .top
frame .bottom
frame .handle -bd 2 -relief raised -bg red \
    -cursor sb_v_double_arrow
```

We'll start off by creating two frames, **.top** and **.bottom**, and a resize **.handle** to change the relative size of **.top** and **.bottom**. In this example, we'll put our “pane” widget in the top level **.**, but it could be used almost anywhere.

```
place .top -relwidth 1 -rely 0 -height -1 \
    -anchor nw
place .bottom -relwidth 1 -rely 1 -height -1 \
    -anchor sw
place .handle -relx 0.9 -width 10 -height 10 \
    -anchor e
. configure -bg black
```

These 3 widgets are arranged by the “placer” in two steps. First we specify the options to **place** that won't change. Both **.top** and **.bottom** will span the entire width of the window (**-relwidth 1**), with **.top** anchored to the top (**-rely 0 -anchor nw**) and **.bottom** anchored at the bottom (**-rely 1 -anchor sw**). The option **-height -1** (a new Tk4.0 feature) decreases the height of **.top** and **.bottom** by 1 pixel, which leaves a “gap” between the windows, so the root window will show through as a black (**.<\!s>configure -bg black**) line between the 2 panes. Finally, we'll place **.handle** near the right edge.

```
bind . <Configure> {
    set H [winfo height .].0
    set Y0 [winfo rooty .]
}
```

To calculate the relative placement of **.top** and **.bottom** we'll need to know the position (**Y0**) and size (**H**) of the root window, which we'll compute any time either could change, by binding the computation to a **<Configure>** event. Since the height (**H**) will be used as a floating point number, we'll tack on a **.0**.

```
bind .handle <B1-Motion> {
    adjust [expr (%Y-$Y0)/$H]
}
```

When the user moves the handle by dragging it with the mouse, we'll compute the fraction of the way down the root window the mouse is, and call **adjust** to

move the windows accordingly. We need to use **%Y**, the mouse position in "root" coordinates, because **%y** is relative to the **handle** and not the root window, **.** .

```
proc adjust {fract} {
    place .top -relheight $fract
    place .handle -rely $fract
    place .bottom -relheight [expr 1.0 - $fract]
}
```

The procedure **adjust** takes a fraction between 0 and 1, changes the the height of **top** and **bottom** windows, and updates the position of the handle. Only the **place** options that may have changed need to be updated. That's all there is to it.

```
proc stuff {root file} {
    text $root.text -yscrollcommand \
        "$root.scroll set"
    scrollbar $root.scroll -command \
        "$root.text yview"
    pack $root.scroll -side right -fill y
    pack $root.text  -fill both -expand 1
    $root.text insert 0.0 [exec cat $file]
}
```

To test it out, we need something to put in the top and bottom halves. We'll create a procedure **stuff** that displays the contents of a file in a text widget with a scroll bar.

```
adjust .5
stuff .bottom $env(HOME)/.login
stuff .top $env(HOME)/.cshrc
```

Now fill each pane, **adjust** the two halves, and off you go.

What a Drag

For our second example, we'll use the placer to permit a user to interactively "drag and drop" a widget within a window. When the user selects a widget, by clicking on it with the mouse, we'll lift it up so it hovers over the window, and casts a shadow as we drag it around. Releasing the mouse drops the widget in its new location.

```
label .label -text "drag me"\
  -borderwidth 3 -relief raised
frame .shadow -bg black
lower .shadow .label
place .label -x 50 -y 50
set hover 5
```

We'll start by creating a widget to drag, **.label**, and its shadow **.shadow**. We'll use **lower** to make sure the shadow is always "below" the widget, and we'll start **.label** in an arbitrary location, 50 pixels from the top left corner of **.** . The

variable **hover** controls how high we'll lift the widget above its window as we drag it.

```
bind .label <1> {
    array set data [place info .label]
    place .label -x [expr $data(-x) - $hover]\
                 -y [expr $data(-y) - $hover]
    place .shadow -in .label -x $hover -y $hover \
                  -relx 0 -rely 0 -relwidth 1 \
                  -relheight 1 -width -2 -height -2 \
                  -bordermode outside
    set Rootx [expr %X - [winfo x %W]]
    set Rooty [expr %Y - [winfo y %W]]
}
```

When we first click on **.label**, we need to lift it up, add its shadow, and compute where it is relative to the root window so we can figure out how to move it. The **array set** command (new in tcl7.4), takes name-value pairs and creates an array from them. Fortunately, the **place info** command happens to report the current **place** options in the form of name-value pairs, permitting us to access and modify individual place options using array accesses. The first **place** command simply moves the widget up and to the left **$hover** pixels as we first press the mouse. I think the second place command, which positions the shadow, uses every available place option!

The **-in** option, which would more accurately be described as “relative to”, causes all locations specified in **.shadow** to be relative to **.label**, instead of **.**, which would be the default. The **-x** and **-y** options, when added to **-relx** and **-rely**, position the shadow where **.label** was before we **$hover**ed it. The **-relwidth** and **-relheight** options make **.shadow** the same size as **.label**, and then the **-width** and **-height** options make the shadow a little smaller, so it will appear farther away. Finally, the **-bordermode** option instructs the placer to include the border of **.label** when computing the sizes for **-relwidth** and **-relheight**.

Finally, we compute the location of the mouse cursor, in pixels, relative to the top left corner of the root window (**Rootx**, **Rooty**), so it will be easier to figure out how to track **.label** with the mouse.

```
bind .label <B1-Motion> {
    place .label -x [expr %X - $Rootx] \
                 -y [expr %Y - $Rooty]
}
```

As the mouse moves, we reposition the widget to follow along. Because we “placed” the shadow relative to the widget (using **the** -in option), it tags along all by itself.

```
bind .label <ButtonRelease-1> {
    array set data [place info .label]
    place .label -x [expr $data(-x) + $hover] \
                 -y [expr $data(-y) + $hover]
    place forget .shadow
}
```

When we release the mouse button, the same **array set** trick as before is used to “drop” the widget back on the window, then remove the shadow.

As you can hopefully see from these two simple examples, the “placer” can be a powerful tool for the dynamic placement of widgets in Tk.

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

Archive Index Issue Table of Contents

Advanced search

The Trade Shows

Randolph Bentson

Arnold Robbins

Issue #16, August 1995

While *Linux Journal* went to DECUS in Washington DC, Randy Bentson attended the Internet World Show and Arnold Robbins took in the sights of the East Coast Comdex.

Comdex and Internet World

While Linux Journal went to DECUS in Washington, DC, Randy Bentson attended the Internet World Show and Arnold Robbins took in the sights at East Coast Comdex.

by Randolph Bentson and Arnold Robbins

EAST COAST COMDEX

East Coast Comdex is held annually, with the Windows World show, at the Georgia World Congress Center in downtown Atlanta. This is one of the world's largest convention centers. Tens of thousands of people attend, literally from all over the world, with a pretty high concentration of people from the Southeast US. The show lasts four days, from Monday through Thursday.

If you come, *don't* attempt to drive to downtown Atlanta and park. Instead, take MARTA, the local rapid transit system, to the CNN-Omni station, and follow the crowds; it's a short walk.

Getting free tickets to see the exhibits is usually pretty easy. Chances are, if you know a vendor displaying there, they can get you tickets. Later in the week, you may even find free passes on the tables when you walk in. I had a press pass, courtesy of *Linux Journal*, although both my company and my wife's company had booths.

It definitely pays to go on a day other than the first day. The lines to get your badge are almost non-existent, which speeds up the process enormously.

Comdex is just enormous. Imagine two huge halls, each of which is bigger than several football fields, packed end to end with display booths and people wandering through trying to see everything. Imagine, if you will, *thousands* of techno-nerds. Not just any nerds, but PC and Windows nerds. Along with the size and the crowd comes the *noise*. The vendors often put on little stage shows and demos. They all have microphones. They all have sound effects. They all have music. It's almost impossible to think while wondering around Comdex, much less have a conversation with your wife (particularly if your wife is like mine, and talks softly).

It also pays to bring your lunch; the food is expensive there. The new owners of Comdex gave out coupons for a free (small) drink, which was a nice touch.

What's Hot

There were very few, if any, really new or interesting items to be seen this year. Comdex was more of the same old thing; Intel-based systems of various sorts being predominant.

Some cute things: Panasonic had a laptop with a keyboard that raised up to reveal a CD-ROM drive. There was another company showing a “remote” mouse that could be used from up to 40 feet away.

The “hot” items were: 1) Windows 95. I'd say that over 95% of the vendors were showing their product(s) working on Windows 95. Not too suprising at the “Windows World” show, but discouraging to an old Unix hacker like myself. 2) Internet. If a product could even be remotely tied into the Internet somehow, it was. The BellSouth ISDN booth was jumping like hotcakes. IBM's booth had a picture of a nun saying “I'm dying to surf the net.” The Internet has arrived, and the masses are diving in. America Online even had a group of about 20 workstations all lined up so people could “test drive the Internet”. Gack. 3) CD-ROM. You name it, somebody's got it on CD-ROM.

Fortunately, Comdex made a lot of the vendors clean up their acts; there weren't any pornographic CD-ROMs visible (unlike last year, which generated a lot of complaints).

It was also pretty clear that OS/2-Warp is an also-ran. I don't think I saw more than 5% of the vendors touting their product running on OS/2.

Perhaps suprisingly, there were almost no PowerPC products. The ones I saw were in the Apple booth, and there were PowerPC systems in the IBM booth.

There was some Linux at the show, but not much. A few vendors with Linux CD-ROMs and such. The most visible Linux was at the bookstore in the lobby, with several Linux books prominently displayed when you walked in, and on the shelves. The O'Reilly books were the nicest looking.

Editorializing

Comdex has changed over the years. It used to be a show for the entire computing industry. Once upon a time, you could find mainframe and mini-computer vendors showing (anyone remember Vax VMS? DG AOS?). You could find Unix vendors showing, like Sun. You also used to get lots of nice freebies.

It has changed. First of all, very few people were giving out toys, much less nice ones. :-(. Secondly, it's a PC and Microsoft world out there. There were fewer Macintoshs than I've ever seen, and much less Unix than usual.

Those companies who were trying to sell server based systems were selling Windows NT, with Unix “also” available. This included companies like Pyramid, and even DEC was pushing NT on the Alpha. It is no wonder that the Unix vendors formed COSE, there really is something to be afraid of. It's a terrible shame that Marketing Hype can sell so *much* Mediocrity to the public, but that seems to be the reality.

The conclusion I'd draw for the Linux world is that if WINE is going to go anywhere, it ought to concentrate on Windows 95 compatibility; i.e., some way to run a Win32 binary under Linux. Skip the Win16 stuff; it'll be dead within a year.

All in all, this year's Comdex was a disappointment. I usually enjoy it, but this year I didn't. It was also a bit depressing to see so much Microsoft stuff everywhere. I have high hopes for the free software world, but it's sobering to see what a small part of the “industry” consists of free software.

On the other hand, there was some Linux. Great things can come from small beginnings, and we're just at the beginning of the growth curve for self-contained, reliable, usable, free software based systems. I hope that in five years, my report from Comdex will tell a much different story.(Arnold Robbins)

THE INTERNET WORLD SHOW

Depending on your measure, Spring Internet World 95 was a great success or a slight disappointment. With respect to its stated goal of presenting what is happening on the Internet, it was excellent. While it can't yet boast the attendence numbers of Comdex, 20,000 visitors and 190 exhibitors made this a respectable conference and may well justify the claim of being the “world's

largest Internet conference." It was held April 10-13 at the heart of Silicon Valley in the San Jose Convention Center, thus it should come as no surprise to learn that the registration was flooded with people who just took some time off from work to check out the show.

Many of the major hardware and software vendors were there. Present were the the traditional vendors: Apple, Dell, DEC, IBM, Microsoft, NCD, Silicon Graphics, Sun, and Tandem. In addition, there were also strictly network product vendors: 3Com, FTP Software, Rockwell, Telebit, Wollongong, and ZyXEL. (I know I'm going to catch flak for this—I'm sure to have missed someone in my list—but I'll press on.)

What made this different from other trade shows I've attended was the focus on user-level access the Internet and the newly discovered World Wide Web.

Everyone was touting some feature of their product that allowed one to compose, access, view, process, or control access to the web. I was amazed by the number of hypertext and HTML text preparation tools that were being displayed. I'll refrain from citing who did what, but some of the low end products were far from magic—I keep seeing the guy behind the curtain. If I were a cynic I would claim the pencil manufacturers were there to show how pencils could be used to prepare hypertext, but that would be an exaggeration.

Since I've worked with mark-up languages such as troff and TeX, I don't find HTML at all difficult. In fact, I find it is missing many of the features I've come to expect of such a language. (Apparently so do others. The proposed HTML 3.0 is a move in the right direction.) This viewpoint renders a lot of the HTML preparation tools rather ho-hum.

Still, there were some products which were close to magic and some other products which will have increased importance in our lives. Specifically, I found the integration of database searches with a web server to be a most natural outcome. Most applications have two elements: the user interface and solving the problem itself. The Common Gateway Interface is an elegent method of using the http daemon and the WWW browser to implement a GUI interface for user application. There were a number of vendors who offered WWW integrated database products.

Another class of products were security related. As we become more thoroughly connected to the network we also become more exposed to malicious intrusions on our systems. By my count there were eight vendors whose sole product was firewall or other security hardware or software. In addition, products such as Secure HTTP, will be integrated by Netscape to give secure interactions over the network.

Of course, since this was a networking exhibition, there were plenty of representation by Internet service providors. There were a number of nation-wide and international vendors present: America Online, BBN Planet, Netcom, Prodigy Services, PSInet, SPRY, and UUNET. It was also interesting to note the number of local providers at the show. If this is any indication, you should have lots of choices for Internet service in any major city.

Finally, I was gratified to see the number of traditional publishers who have recognized that the Internet is a marketplace that they should serve. Present were divisions of Dun & Bradstreet, MacMillan Publishing, San Jose Mecury News, O'Reilly & Associates, Random House, Van Nostrand Reinhold, and John Wiley & Sons.

So what was the disappointment? The poor Linux visibility. I guess this is a side effect of the commercial nature of the show. After all, how can the developers of Linux, TCP/IP, Lynx and Mosaic justify the expense of renting booth space?

This led me on a quest—that I didn't complete—looking for Linux inside. I started at one end of the exhibition hall and went from booth to booth asking about Linux support. I found some solace in the responses. A number of vendors said it wouldn't be long before a Linux version of the product would be available -- after all, that's what the developers inside the companies were using at work or at home. There's a chance that next year even more products will have “Linux” in their list of supported systems, Randolph Bentson

Randolph Bentson (bentson@grieg.seaslug.org) has been programming for money since 1969-writing more tasking kernels in assembly code than he wants to admit. His first high-level language operating system was the UCSD P-system. For nearly 14 years he has been working with Unix and for the last year he's been enjoying Linux. Randy is the author of the Linux driver for the Cyclades serial I/O card.

Arnold Robbins (arnold@gnu.ai.mit.edu) is a professional programmer and semi-professional author. He has been doing volunteer work for the GNU project since 1987 and working with Unix and Unix-like systems since 1981.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

What's GNU

Arnold Robbins

Issue #16, August 1995

This month's column takes a brief look at the GNU coding standards, a document that describes how to write and package GNU software.

What is it that makes a GNU program a GNU program? What makes GNU software “better” than other (free or non-free) software? The most obvious difference is the GNU General Public License (GPL), which describes the distribution terms for GNU software. But this is usually not the reason you hear people saying “Get the GNU version of **xyz**, it's much better.” GNU software is usually more robust, and performs better, than standard Unix versions. We're going to look at some of the reasons why, and at the document that describes the principles of GNU software design.

The *GNU Coding Standards* describe how to write software for the GNU project. It covers a range of topics. As of this writing, related chapters are not grouped together, so we'll look at the chapters by topics, not in the order they appear.

You can find the *GNU Coding Standards* in the Autoconf distribution, currently **autoconf-2.3.tar.gz**, from your nearest GNU mirror site. An ASCII copy (**standards.txt**) should also be available as a standalone file from your nearest GNU mirror site, as well.

Intellectual Property Rights

The first issue discussed has to do with intellectual ownership. If you're GOING to write a GNU program that re-implements a Unix utility, *don't* look at the Unix source code! (Source code licenses are harder and harder to get these days, so this is less of a problem than it was 10 years ago.) The other issue has to do with copyright assignment. If you're going to write or work on a GNU program, you have to either declare your work to be in the public domain, or assign the copyright in it to the FSF. (Small changes don't have to do this, so don't be scared off by this if you want to submit a bug fix. On the other hand, if you

enhance GNU [cw]find[ecw] so that it can read **cpio** archive tapes, you probably would have to do paperwork. Even this is usually painless.)

You can, of course, write a program from scratch, release it under the GPL, and keep the copyright. You may also generate your own changes to a program for which the FSF owns the copyright, and distribute your version separately from the FSF's version, under the GPL. Assigning copyright to the FSF is only a necessity when you want your changes to be folded back into the main distribution of a GNU program.

Program Design

A number of chapters provide general advice about program design. The four main issues are compatibility (with standards and Unix), what language to write in, whether to rely on non-standard features of other programs (in a word, “don't”), and what “portability” means.

Compatibility with ANSI, POSIX, and Berkeley Unix is an important goal. But it's not an overriding one. The general idea is to provide all necessary functionality, with command line switches to provide a strict ANSI or POSIX mode.

C is the preferred language for writing GNU software, since it is the most commonly available language. In the Unix world, ANSI C is only now becoming common (sad but true), so K&R C is still the most widely portable dialect. This is changing rapidly though, with C++ becoming more commonplace. One widely used GNU package written in C++ is **groff** (GNU troff). With GCC supporting C++, it has been my experience that installing groff is not difficult.

The standards state that portability is a bit of red herring. GNU utilities are ultimately intended to run on the GNU kernel with the GNU C library. But since the kernel isn't finished yet, and users are using GNU tools on non-GNU systems, portability is desirable, just not paramount. The standard recommends using Autoconf (about which I one day hope to write a column) for achieving portability among different Unix systems.

Program Behavior

The next group of chapters provides general advice about program behavior. We will return to look at one of these chapters in detail, below. These chapters focuses on how to design your program, how error messages should be formatted, how to write libraries (make them reentrant), and standards for the command line interface.

Error message formatting is important, since several tools, notably Emacs, use the error messages to help you go straight to the point in the source file or data file where an error occurred.

GNU utilities should use a function named **getopt_long** for processing the command line. This function provides command line option parsing for both traditional Unix style options (**gawk -F: ...**) and GNU style long options (**gawk --field-separator=: ...**). All programs should provide **--help** and **--version** options, and when a long name is used in one program, it should be used the same way in other GNU programs. To this end, there is a rather exhaustive list of long options used by current GNU programs.

Writing C Code

The most substantive part of the manual describes how to write C code, covering things like formatting the code, comments, how to use C cleanly, how to name your functions and variables, and how to declare, or not declare, standard system functions that you wish to use.

Code formatting is a religious issue; many people have different styles that they prefer. I personally don't like the FSF's style, and if you look at gawk, which I maintain, you'll see it's formatted in standard K&R style. But this is the only variation in **gawk** from this part of the coding standards (other variations will go away in **gawk** 3.0, coming this year).

Nevertheless, while I don't like the FSF's style, I consider it of the utmost importance, when modifying some other program, to stick to the coding style already used. Having a consistent coding style is more important than which coding style you pick.

What I find important about the chapters on C coding is that the advice is good for *any* C coding, not just if you happen to be working on a GNU project. So, if you're just learning C, or even if you've been working in C (or C++) for a while, I would recommend these chapters to you, since they encapsulate many years of experience.

Documenting Programs

Two chapters cover writing documentation for your program. The preferred way is to write a manual using Texinfo, which was discussed in an earlier column (Issue #6, October 1994). There is some nice advice in here about writing manuals. And, as described earlier, Texinfo is an enjoyable language in which to write documentation.

How to make releases

Finally, there are three chapters devoted to the mechanics of making a release. These chapters discuss the conventions to use for Makefiles, how configuration should work, and other generalities about how a release should work.

These chapters, together with the Autoconf manual, provide the needed information for packaging up a program and making the final released **tar**] file.

What Makes A GNU Program Better?

We'll take a look now at the chapter entitled *Program Behavior for All Programs*. This chapter provides the principles of software design that make GNU programs better than their Unix counterparts. We will quote selected parts of the chapter, with some examples of where these principles have paid off.

> Avoid arbitrary limits on the length or number of *any* data structure, including file names, lines, files, and symbols, by allocating all data structures dynamically. In most Unix utilities, "long lines are silently truncated". This is not acceptable in a GNU utility.

This is perhaps the single most important rule in GNU software design, "no arbitrary limits." All GNU utilities should be able to manage arbitrary amounts of data.

While this makes it harder for the programmer, it makes things much better for the user. I have one gawk user who runs an awk program on over 650,000 files (no, that's not a typo) to gather statistics. gawk grows to over 192 Megabytes of data space, and the program runs for around seven CPU hours. He would simply not be able to run his program using another awk implementation.

> Utilities reading files should not drop NUL characters, or any other nonprinting characters (including those with codes above 0177). The only sensible exceptions would be utilities specifically intended for interface to certain types of printers that can't handle those characters.

It is also well known that Emacs can edit any arbitrary file, including files containing binary data!

> Check every system call for an error return, unless you know you wish to ignore errors. Include the system error text (from perror or equivalent) in *every* error message resulting from a failing system call, as well as

> the name of the file if any and the name of the utility. Just “cannot open foo.c” or “stat failed” is not sufficient.

Checking every system call provides robustness. This is another case where life is harder for the programmer, but better for the user. An error message detailing what exactly went wrong makes finding and solving any problems much easier.

> Check every call to malloc or realloc to see if it returned zero. Check realloc even if you are making the block smaller; in a system that rounds block sizes to a power of 2, realloc may get a different block if you ask for less space.

> In Unix, **realloc** can destroy the storage block if it returns zero. GNU **realloc** does not have this bug: if it fails, the original block is unchanged. Feel free to assume the bug is fixed. If you wish to run your program on Unix, and wish to avoid lossage in this case, you can use the GNU **malloc**.

> You must expect **free** to alter the contents of the block that was freed. Anything you want to fetch from the block, you must fetch before calling **free**.

In three short paragraphs, Richard Stallman has distilled the important principles for doing dynamic memory management using **malloc**. It is the use of dynamic memory, and the “no arbitrary limits” principle that makes GNU programs so robust and more capable than their Unix counterparts.

> Use **getopt_long** to decode arguments, unless the argument syntax makes this unreasonable.

Long options were mentioned earlier. Their use is intended to make GNU programs easier to use and more consistent than the Unix versions. The **getopt_long** function is a nice one; it provides you all the flexibility and capabilities you may need for argument parsing. As a simple yet obvious example, **--verbose** is spelled exactly the same way in *all* GNU programs. Contrast this to **-v**, **-V**, **-d** etc.

Finally, we'll quote from an earlier chapter that discusses how to write your program differently than the way a Unix program may have been written.

> For example, Unix utilities were generally optimized to minimize memory use; if you go for speed instead, your program will be very different. You could keep the entire input file in core and scan it there instead of using stdio. Use a smarter algorithm discovered more recently than the Unix program. Eliminate use of

> temporary files. Do it in one pass instead of two (we did this in the assembler).

> Or, on the contrary, emphasize simplicity instead of speed. For some applications, the speed of today's computers makes simpler algorithms adequate. Or go for generality. For example, Unix programs often have static tables or fixed-size strings, which make for arbitrary limits; use dynamic allocation instead. Make sure your program handles NULs and other funny characters in the input files. Add a programming language for extensibility and write part of the program in that language.

An excellent example of the difference an algorithm can make is GNU **diff**. My computer's previous incarnation was an AT&T 3B1; a system with a MC68010 processor, a whopping two megabytes of memory and 80 megabytes of MFM disk.

I did (and do) lots of editing on the manual for gawk, a file that is currently over 17,000 lines long (although at the time, it was only in the 10,000 lines range). I used to use **diff -c** quite frequently to look at my changes. On this slow system, switching to GNU **diff** made an extremely noticeable difference in the amount of time it took for the context diff to appear. The difference is almost entirely due to the better algorithm that GNU **diff** uses.

Summary

The *GNU Coding Standards* is a worthwhile document to read if you wish to develop new GNU software, enhance existing GNU software, or just wish to learn how to be a better programmer. The principles and techniques it espouses are what make GNU software the preferred choice of the Unix community.

Epilogue

As mentioned, the released version of the standards covers its topics in a rather haphazard order. As a result of working on this column, I volunteered to re-organize them into several related chapters. This new version may be available by the time you read this article; keep an eye on your nearest GNU mirror site.

Arnold Robbins is a professional programmer and semi-professional author. He has been doing volunteer work for the GNU project since 1987 and working with UNIX and UNIX-like systems since 1981. Questions and/or comments can be sent by e-mail to arnold@gnu.ai.mit.edu

Archive Index Issue Table of Contents

Advanced search

Sendmail: Theory and Practice

Phil Hughes

Issue #16, August 1995

Yes, the answer is there, but the book is more than many people need or want to get into - sort of like reading the National Electrical Code book to find out how to replace a fuse.

- Author: Frederick M. Avolio & Paul A. Vixie
- Publisher: Digital Press
- ISBN: 1-55558-127-7
- Price: $29.95
- Reviewer: Phil Hughes

Many of us who have had to configure Sendmail think of it as something similar to fixing the plumbing in our house - it has to be done, we sort of know how to do it and we wish we could ignore it and it would go away. Yes, Sendmail is powerful and, for most systems, necessary, but it is complicated. To make matters worse, you don't have to make changes often enough so that you actually learn how to do it right and remember it.

When Eric Allman's book on Sendmail was published, I got a copy and was immediately intimidated. Eric is the author of Sendmail and his book is thorough - tipping the scales at almost 800 pages. Yes, the answer is there, but the book is more than many people need or want to get into - sort of like reading the National Electrical Code book to find out how to replace a fuse.

If this same thing happened to you, *Sendmail: Theory and Practice* may be the right answer. In 262 pages Avolio and Vixie address just what the book title says: theory and practice. It takes the fear out of Sendmail configuration by first explaining the practical considerations involved in electronic mail transfer and then goes on to show how to configure Sendmail to accomplish the tasks.

The first 90 pages cover practical information about addressing over networks, including the problems of mixed-type addresses (that is, a combination of a uucp address and domain address). These pages also cover mail user agents, their interface to Sendmail and how aliases work in Sendmail. Or, more correctly, how to use Sendmail aliases to do what you need in a reliable and secure fashion. Again, the emphasis is on practical application.

The next chapter offers the basics of macros and rules. This is presented in a practical and non-threatening manner with an emphasis on what you need rather than a lengthy look at all the capabilities.

The next chapter addresses the IDA Kit extension to Sendmail. It does a good job of showing how the DBM tables of the IDA kit tie into the rules in the sendmail.cf file. While I personally had hoped I didn't need to know this, the book gives enough information to help you understand this without getting bogged down in theory.

Even if you have what looks like a working Sendmail, the chapter on “Maintenance and Administration” will help you feel a lot better about your relationship with Sendmail. After going through all the files related to Sendmail and all the command line options, it looks at things you can check, why you might want to check them, and how to check them. For example, a section on queued mail offers five steps to help identify why mail is remaining in the queue and what to do to get it on its way.

The book ends with a series of appendicies that offer resources or pointers to resources that you need. These include summaries of the options and mailer flags for the sendmail.cf file, sample sendmail.cf files, logging and debugging information, and even the form to be sent in to the InterNIC to register a domain.

The main shortcoming of the book is that it does not address Release 8 of Sendmail. The authors claim that the philosophy of R8 is the same as R5. Certainly the added functionality in R8 is not covered. They may be absolutely right in their decision, however. This book contains plenty to keep you thinking, and adding complications of R8 could have detracted from the book's usability.

If you are afraid of Sendmail but have to deal with it, this is the right book to get. It doesn't tell you everything but it does tell you more than most systems administrators need to know and it is presented in a very practical manner from the point of view of two people who see Sendmail as a tool and have learned to use that tool to accomplish their tasks.

Archive Index Issue Table of Contents

Advanced search

Letters to the Editor

Various

Issue #16, August 1995

Readers sound off.

Monte Told Amy . . .

I'd like to congratulate you on getting the job of planning the layout of *Linux Journal*....and I'd like to also say that it was much more enjoyable to read this time around. Thank you. —Monte Corbit monte@intellinet.com

And We Have Content . . .

I also wanted to compliment you guys on the excellent content level of *LJ*. — Tom MorseLernout & Hauspie Speech Products tmorse@lhs.com

Can't Please Everyone

It is seldom that an editor asks an opinion of their “art director”. Understandably since you have a layout person doing the work that an art director should your requests for an opinion is a cry for help.

Judging from the recent changes this cry of help seems justified. Your magazine was brought to my attention by my husband, one of your subscribers. His response was, “Look how amateurish this is.”

I would like to start with the visual on page 19. This is a bad shot. It may have been the only one that you had. Why wasn't it cropped differently or the background replaced? Did Amy like the hair sticking out and the UFO by his right shoulder?

The lime green? Was it inspired by the well designed ad on page 29? Do you feel it works as well as on page 6 & 7? Was the Linux color on page 3 chosen from the ad on page 59? Do you feel the two colors complement each other well on page 3?

In newspapers rough sketches lend well to the 80 line screens. In magazines as in your case the rough thumbnail sketches appear badly drawn and quite wobbly. Are you trying to give the impression that your magazine can not afford an illustrator?

Page 10. Why is 'Stop the Presses' so big? Is it to compete with the even larger horsey Headlines throughout your magazine? Doesn't by Phil Hughes look so small in comparison floating in all that white space. It gets a bit lost just hanging there. Shouldn't the three be married in a design unity? Stop the presses? (see page 116 of June Wired Magazine)

Why is the logo of your Magazine treated differently between the cover and page 3? Why is the date under Journal? Was the page intended to be smaller or did Amy feel the white space and the line added something? Why is the XBase lines so big on the cover? Do you feel that the cover breaths well of is it a clutter on design elements thrown on the page to fill up all space?

My suggestion is send Amy on a Design course or two. Perhaps if she didn't race through the design she may have done a better job. But remember you get what you pay for. —Most sincerely,Cinna cinna@interport.net

The Final Vote ...

Kudos on the new look for *LJ*! Pass on my appreciation to Amy Wood. Keep it up. —Andy Cook andy@anchtk.chm.anl.gov

Archive Index Issue Table of Contents

Advanced search

ELF Released for Linux

Michael K. Johnson

Issue #16, August 1995

What does that mean to the average Linux user? What is ELF, anyway?

The ELF tools for Linux have just been publicly released, after months of testing. What does that mean to the average Linux user? What is ELF, anyway?

ELF is a advanced new binary file format for executables, libraries, and object files (which are used to create libraries and executables). It is the native binary file format of Unix System V Release 4, and is far more powerful and flexible than the original binary file format used on Linux, which is often called “a.out”.

Why do I care?

The people who really *care* about this are the programmers who make it possible for you to run a Linux system. They care because ELF makes several things much easier for them to do, and even makes some things that were previously practically impossible easy. This will allow them to develop better and more interesting software—which every Linux user gets to use.

Also, the ELF shared libraries are much more flexible than the old a.out DLL shared libraries which have been the Linux standard for a few years now. Backwards compatibility will be much easier to maintain with the ELF libraries, which will give all Linux users a more stable platform. “Wizardry” should no longer be required to upgrade libraries or compilers, once you are using ELF.

Do I want to upgrade?

Probably not yet. If you use Linux because it is an adventure, or to learn, then you probably do want to upgrade. However, if you just want to get work done, then you probably do not want to upgrade quite yet.

There are two ways to upgrade. One is to simply install the ELF libraries, and if you compile your own kernels, make sure that your kernel has been compiled with ELF binary support. This will allow ELF binaries to run on your system with no further work on your part. The first problem with this is that using both the older a.out shared libraries and the ELF shared libraries at the same time will use more memory than using all binaries of one type or the other, and therefore will almost certainly slow your system down. Therefore, this is only recommended it you want to be able to run occasional ELF binaries. The second problem with this is that you are required to rearrange some files in order to install the ELF libraries.

The other way is to wait until your favorite distribution of Linux is available in an all-ELF form, and re-install your system. As drastic as this sounds, it will lead to higher performance, especially if you want to run ELF binaries regularily. Hopefully, most distributions will make this easier than it sounds, perhaps by having an option to only install binaries. Distributions with package maintenance options should give you the option to uninstall your old packages and install new ELF-based ones without a fully reinstalling your system.

You will probably want to wait to reinstall your system until your favorite distribution has provided a (relatively) easy way to upgrade. If you want to use a new program available only in ELF (i.e., play with a new toy), your time will probably be better spent simply installing the new libraries. Unfortunately for normal users, it is not as easy as making a directory and dropping a few files in it. It requires installing a new **ld.so** and moving libraries. There has been some talk of a script to automate the process, and it will probably have been written by the time you read this, but I can't tell you where to get it because as I write this it doesn't yet exist.

This isn't wizardry?

The unfortunate fact is that to get to the state where it is so much easier to upgrade will require harder work initially. If you are patient, someone else may do the work for you; either someone who writes a generic upgrade script (this is only really possible thanks to the Linux File System Standard, which was explained in the July issue of *LJ*) or the maintainer of your favorite distribution. But there is no free lunch; either you have to do the work, or someone else does. The difference with ELF is that once that difficult work has been done once, the *next* time you need to upgrade your shared libraries, it will be easier than it ever has been—for whoever does the work.

Update on Linux/Alpha

For those following the Linux/Alpha port, Jim Paradis of DEC recently announced that the networking code in the Alpha port is starting to work, a few

weeks after work on networking commenced. This rapid progress, which has been characteristic of the Alpha port, suggests that DEC is likely to meet their projected fall release of the full system.

Archive Index Issue Table of Contents

Advanced search

Novice to Novice

Dean Oisboid

Issue #16, August 1995

As part of a regular series by and for novices, Dean gives a novice's point of view on some of the many spreadsheet and text editing programs that run under Linux.

This series has aimed at the Unix beginner who has experience with MS-DOS. I have assumed that such a person is exploring Linux primarily to learn Unix easily, cheaply, and conveniently. This person may buy or download Linux, get it running, and probably focus on the main veins—Networking, X-Windows, GNU C/C++, etc.—without really exploring some of the **other** offerings such as **sc**, **oreo**, or **xfractint**. I originally dismissed these programs, thinking they couldn't have much substance being freeware and certainly couldn't compare to what exists in DOS or Windows.

Then I realized that these freebies weren't necessarily equal to the **latest** versions of commercial programs like Lotus-123 or WordPerfect but perhaps would be comparable to older versions. And I realized that, just like in DOS/Windows, these freeware and shareware programs served to fulfill an applications void or to present inexpensive alternatives to their commercial counterparts or came into existance because they were fun and a challenging exercise to develop.

This article will be the first of a small journey to seek out these other programs and see what they have to offer. I will concentrate on three main topics: spreadsheets and text editors, databases, and seredipity. The general focus will be towards business applications and things that would impress someone visiting from a DOS/Windows environment.

For the majority of these programs, a major advantage over commercial rivals is that the source code is included. Linux lovers already know of that luxury. Unfortunately, that advantage usually dissipates with the need to compile the source, a task that brings anguish to many novices. As a note of reassurance to

other novices, I've found that recompiling isn't always a headache. With a swap file active to boost memory, I've had few problems with compiles. Many of the glitches I've had occur when the make files expect certain files in certain places; that is, when file locations are hardcoded in. And, of course, it helps to read the README files.

A point to remember: many of these programs undergo constant revision and by the time this article sees the light of print newer versions may have been released. With the updates, changes may be made to installation routines or requirements. Consequently the procedures or problems I describe during installing or bugs that I find may not always carry over to newer releases.

My goal is to see how these programs compare to known DOS/Windows counterparts, not to imply that either operating system is better, but simply to provide a frame of reference.

Finally, if no installation option was available from Slackware I unarchived these programs under /usr as those more experienced than myself in Unix have recommended.

Let's snoop around, shall we?

Interlude: The Agonies of Obsolescence

I realized that my copy of Linux from October 1994 was old. Factor in the time for publishing preparations and I was committing a grave disservice. It was time to upgrade to Linux 1.2.1.

Although I had been using Morse Telecommunications' Slackware Professional 2.1 I decided to try another Linux offering. Highly recommended, and what I tried, is InfoMagic's Linux Developer's Resource. This 4-CD monster is an amazing bargain. Archives of Sunsite, TSX-11, and GNU, “live” Slackware, tons of everything else, and best of all: a great price. For the novice, the package may be too overwhelming—there's only a little manual for installation. (Other documentation is on the main CD. InfoMagic offers a beginning package —“Linux Toolbox”—that looks pretty good and comes with a variety of printed help.)

I have two complaints. The first is when using the Windows Boot/Root disk maker routine of the Distribution. The manual says that the program will let you choose UMSDOS as a Root option but I didn't see one and I had to create one manually from the CD. Not a big deal and certainly not heart attack inducing.

The second complaint, not specific to InfoMagic's product, is far worse, but it only applies, I believe, to those having a Sony cdu31a/cdu33a CD-ROM. The auto-detect function was removed or disabled from the kernels. This means that even when you specifically choose the cdu31a kernel from the “Q” disks or recompile the kernel your CD-ROM still will not be recognized. At least it did not for me. Why the decision to remove this chunk of code which worked in earlier versions, I don't know. But I went through Linux-novice hell to get Linux working again.

[The reason that cdu31a/33a autodetect support was removed is that the only possible way to autodetect these drives is so dangerous that detecting it has the potential to hang the computer at boot. This was causing immense troubles for thousands of people without Sony CD-ROM drives, and so the autodetect capability had to be removed, and it is no longer part of the standard Linux kernel. —ED]

A positive change is in the X-Windows configuration routine. Now it does all the dirty work. You do **not** have to manually edit any files afterwords. As before, however, you will need all the information about your monitor and video card, but the entire process is easy and quick. Kudos!

Spreadsheets

sc (6.21, on disk set 'AP') and **oleo** (v0.03.2, from Sunsite) are both simplified spreadsheets similar to early version of Lotus-123 or Excel. Both programs have a fair variety of functions but lack the **bang!** of their MS-DOS counterparts. I had no problems installing either one because binaries were included in the archives. Both ran without fault. The problems I did have were primarily related to cursor movement. I'm spoiled; I like to use the cursor keys and PageUp and PageDown. Both programs used keys and combinations that may be familiar to Unix users, especially Emacs users, but aren't familiar to this Novice. For example, the Lotus-123 '/' command didn't bring up the menu. Also you can't just type a number into a cell; you must first hit the '=' key (like Excel). I received an out of memory error in sc when I tried to get the cursor to (g)o to cell zz3000. Sure I was running sc through X-Windows but the swapfile was active. Like I said, I'm spoiled.

xspread (1.1L2, from Sunsite) is an X-Windows spreadsheet based on sc. Like sc the data is entered via '='. Unlike sc '/' does bring up a Lotus-123-like menu. If you had to use any of the freeware spreadsheets mentioned this would get my recommendation, simply because of the Lotus-like menu.

Text editors

vi and **Emacs** (GNU version) are the two most popular and renowned text editors in Unix. Most of the following appeared on the main Slackware CD and had installation scripts available.

vi is on most every system and is a useful compact program, though I don't care for it personally. The commands and three operating modes aren't the most intuitive—not that DOS's **edlin** fares better by comparison (though DOS's newer **edit** does shine). My advice to novices is to learn it just in case. Again, this program is everywhere so it helps you to have at least a basic understanding of how to work it.

GNU emacs (19.27, on disk "E") is not a program, it's a lifestyle. Either you love it because it can do almost anything or you hate it for its obscureness. I know it overwhelms me. The X-Windows version is amazing. It has a menu bar to make things easier and the menu choices are bizarre and unique. You can read your mail or news, learn from the tutorial, search by most anything. Choosing calendar lets you play with Mayan, Islamic, Hebrew, and some other dates. "Moon phases" is an option. A diary is available. As for word processing I didn't think it so hot. In X-Windows, I expected the ability to change fonts or typesizes and there was no obvious way to do so. Comparing emacs to Microsoft's Word for Windows will show even greater inconveniences for the would-be Unix word processor. But emacs can apparently do so much for so many types of people —programmers, writers, e-mail readers—that to focus on just one area doesn't quite give it a fair appraisal.

[The reason that it doesn't compare well with work processors is that it isn't one. Even in the DOS and Windows, world, programmers don't use Word for Windows to write the programs, nor do they use Brief to write theses, as a general rule—Ed]

A mild warning: if you don't have the option to run it from CD, expect emacs to use a little over 20 megs of your hard drive.

And that's all I will say about vi and emacs. If you want more in depth information about any of the programs look for HOW-TO's, FAQ's, and books. *Linux Journal* had an article about customizing emacs in the September 1994 issue. A good basic intro to vi is in Matt Welsh's "Linux Installation and Getting Started" guide.

A quick aside: I don't care for vi but I use it; I like the X-Windows version of emacs but I don't use it yet. In general, I find the non-X-Window command structure of the emacs programs too arcane. In part, this is because I'm coming over from DOS where most text editors have available the Wordstar-compatible

commands or where most everything is mouseable. Since I use a “modern” word processor, many of the Unix programs I found seem outright brutal. For me <F1> should always be the help key, not some ^h-? combination requiring three hands.

Emacs clones

Lucid Emacs (19.10) is a clean variant of GNU Emacs and is very nice. Unlike the GNU version it doesn't bombard you with a variety of unusual options. It appears to focus on basic text editing and that's it! Cursor key and mouse movement and all the other basics are there. The one immediate nitpick I discovered is that when you change the font or type size for a sentence, word, or a block, all the text in the file gets changed. [This is because Lucid Emacs, like GNU Emacs, is a text editor, and you are changing the font that the whole file is displayed in—Ed] Like GNU Emacs, Lucid will eat up about 20 megs of hard drive space so you may want to choose between the two and/or run it from the CD.

jove (4.14.10, on disk “AP”) stands for “Jonathan's Own Version of Emacs”. Unfortunately I couldn't explore whatever differences exist between this and the real emacs since I'm not an expert user of emacs. One obvious difference is the amount of hard drive space used. **jove** doesn't come close to the over-20 megs that emacs consumes. Other than that, the same annoying obscure commands are present.

vi clones

vim (3.0, on disk “AP”) stands for Vi IMproved. I've already stated that I'm not enthused about vi. Sadly, vim doesn't improve my opinion. It may be improvement over vi but it still doesn't make the commands any more intuitive or the vi idea any more palatable.

fpted is another clone. Unfortunately I couldn't really try it because it crashed. It looked like, at least, a help menu was easily available.

celvis required compiling, which produced a multitude of warnings. Yet it ran without any apparent problems. A unique aspect to this clone is that with the proper terminal you can write in Chinese.

vile and **xvile** for X-Windows are yet more vi clones. I liked these two because of the highlight bar at the bottom which, in part, clearly tells the command for help. I keep focussing on help because for novices it is perhaps the most important command to know. Considering that many of these programs do not include help files or tutorials, a menu bar saying that help is ^h-? or something is a great benefit.

nvi (1.03) is—guess what—yet **another** clone of vi but without a helpful menu bar at the bottom. [It's the official “new vi” from Berkeley—Ed]

xvi needed compiling which I passed on. My guess is this program is yet another variant on vi and will run under X-Windows.

Other editors

jed (0.96, also on disk “AP”) is a program that I like. It uses the cursor keys for easy movement. There's a menu at the top and another available that appears at the bottom when you hit ^h-?. However, one warning about that: option 6 will not exit you from the secondary menu but from entire program! [Jed can also use Emacs, EDT, and Wordstar keys, and the curser keys work the same no matter what keyset you are using, so most people will be comfortable with it one way or another—Ed]

joe (2.2, disk “AP”) is a program I also like but not quite as much as jed. The screen is cleaner, not as cluttered with a help menu, just a simple bar at the top. **joe** uses Wordstar-like commands so I had no problems with cursor movement and other commands. The help was a bit tricky. ^k-h to turn it on but to page through required an awkward ^[-. or ^[-, (for backwards).

ed stands for “standard editor” and I couldn't stand it. This is a very small program with no obvious help command. In fact, to figure out the commands you have to read the source code; fine for hackers, not so good for novices. [Even most hackers avoid this like the plague. It complains about errors with only a question mark; the manual says, “Experienced users will usually know what is wrong.” —Ed]

ez (7.0, ATK 6.3.1) provides a clean X-Windows Word Processor that apparently can be used standalone or part of the larger Andrew system. This program was covered in *Linux Journal*, issues 4 (August, 1994) and 5 (September, 1994). **ez** certainly lives up to its name with mouse and cursor key control and a good word processor with multimedia options. Despite its ease it didn't have any obvious way to change fonts or type sizes, unusual for a program of this quality. I discovered a window that had the information about fonts and whatnot listed but again no obvious way to make changes.

[**ez** can edit documents of many different types transparantly. In order to access the word processing features, you have to open the right kind of document by giving the file the correct extension. Files ending in .d or .doc will turn on ez's word processing capability, which will show up in obvious menus; for instance, font changes are accessed through the “Font” menu—Ed]

crisp (v2.2e from the Sunsite CD) is an editor that is like emacs in that it has great expandability, obscure commands, and can be used for programming. Untarring the binaries produced a barrage of files including the two main executables cr and xcr (for the X-Windows version). Upon startup of either you get a window with an information bar at the bottom. For novices there is no obvious help command. ^-h didn't work. crisp was the last package I looked at and I was determined to at least figure some of it out. I used **mc** and finally discovered user and programmer help files under /usr/local/lib/crisp/help/org. The user.hlp file said Alt-h would give help but back in crisp it didn't; it gave gibberish that told me I probably had an emulation problem. Enough, enough, I resign. For Crisp this is apparently the last freebie version and they now have a snazzy graphical offering. I hope it has an obvious help command.

Whew!!!

I probably missed a few spreadsheets and text editors while scrounging through the CD's. My general impression is that for the spreadsheets the level of the freeware is comparable to an early Lotus-123 or Excel. But the programs work and will get you through your basic spreadsheet needs.

There are quite a few choices of text editors available with the majority of them clones of either vi or emacs. My personal favorites are jed, joe, either X-Windows version of emacs, and Andrew's ez. They get the job done fairly easily and Emacs can apparently handle most anything. To conserve hard drive space, I would probably choose GNU Emacs over Lucid or maybe have both runnable off the CD. But to summarize my feelings: I guess I expected programs that were more for word processing and for that I gained disappointment. Maybe the WordPerfect demo would have met my expectation could I have installed the beast. For word processing I would stay with what I have in DOS or perhaps purchase a good commerical UNIX word processor. As a writer, I need advanced features and none of the above programs really satisfied that criterion, instead being better for producing ASCII files and for programming environments.

[Projects are now underway to provide more word processing programs. We will cover them in *Linux Journal* when they are ready for public consumption—ED]

Dean Oisboid, owner of Garlic Software, is a database consultant, Unix beginner, and avowed Diplomacy addict. He can reached at 73717.2343@compuserve.com.

Archive Index Issue Table of Contents

Advanced search

New Products

LJ Staff

Issue #16, August 1995

Interactive UNIX Guide, SEDIT and S/REXX for Linux and more.

Interactive UNIX Guide

Sunni Micro Lab announced The Interactive UNIX Guide for Linux. This is a computer-based UNIX training system.

It includes 72 interactive tutorial sesssion covering 90 Unix utilities. $110 Cdn/ $80 USD.

Contact: Sunni Micro Lab, 1300 Britannia Road East, Suite 208, Mississauga, ON L4W 1C8 CANADA. Phone: 905 795-9292. Fax: 905-795-9291.

Price CAN$110, US$80

SEDIT and S/REXX for Linux

Benaroya announced a Linux version of their SEDIT, S/REXX and S/REXX Graphical Debugger products. The Linux version currently in beta test is available for download along with versions for the other currently supported UNIX™ platforms.

SEDIT is a UNIX text editor patterned after IBM's XEDIT mainframe editor. It operates with a GUI under X windows or in character mode when X is not suitable.

S/REXX implements all REXX language features described in the second edition of Mike Cowlishaw's book, “The REXX Programming Language”, except that numeric digits are limited to 15.

The downloadable software *and* a flat ASCII file version of the WWW document is available via anonymous ftp from directory pub/sedit at

ftp.portal.com. See either seditsrexxinfo.txt.gz or seditsrexxinfo.txt (same content) for the descriptive material.

Introductory pricing for SEDIT or S/REXX starts at $99 or both for $160.

Contact in North America: Dave Morris, Barili Systems Limited,10873 W Estates Drive. Cupertino, CA 95014, E-mail: sedit@shell.portal.com, url: www.portal.com/~sedit

Outside North America: Benaroya, 31 Rue de Constantinople, 75008 Paris, France, +33-1-47 33 33 24, FAX: +1 47 22 06 17

Price: SEDIT or S/REXX start US$99 each or both for US$160

Linux Internet Archives

Yggdrasil Computing announced a new “Linux Internet Archives”, a new four CDROM containing the latest snapshots of the Linux FTP archive sites from the internet, including: Slackware 2.2.0.1* **, Debian .93 beta, MCC 1.0+, mini-linux, Jurix, Xdenu 2.0 and SLS, sunsite.unc.edu:pub/Linux*, tsx-11.mit.edu:pub/linux*, ftp.x.org X11R6 archive*, prep.ai.mit.edu:pub/gnu, JE Linux (Japanese Extensions), and Linux X software.

The disks also contain a snapshot of DEC Alpha Linux port (not a runnable system), and the Internet RFC standards.

The disks contain Boot floppies with fixed version of fdisk for Slackware 2.2.0.1 (in addition to the original boot floppies). 4 CD Set - $19.95.

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

Price: US$19.95 for the four-CD set

Archive Index Issue Table of Contents

Advanced search

Memory Allocation

Michael K. Johnson

Issue #16, August 1995

Memory allocation of some sort is required in practically any program, but in the Linux kernel it is more complex than in user-lever code—for good reason.

Memory allocation in the Linux kernel is complex, because there are significant constraints involved—and different ways of allocating memory have different constraints. This means that anyone writing Linux kernel code needs to understand the various ways of allocating memory, including the tradeoffs involved. This makes for for more efficient use of memory and CPU time—you can specify exactly what you need—but it also makes for more demanding programming.

There are essentially five different ways of allocating memory in the kernel. That's a white lie, but it is close enough to the truth for anyone who needs to read this article to learn about kernel memory allocation. Three (which provide dynamic allocation) are generally useful, and two (which provide static allocation) are deprecated, and are mostly historical artifacts that should not be used. We will discuss the advantages and limitations of the useful ways first, and will only briefly mention the two deprecated ways at the end of this article so that you know what to avoid.

Memory Allocation Strategies

There are a few rules that apply no matter what form of dynamic kernel memory allocation you attempt to do. Whenever you attempt to allocate memory in kernel space, you **must** be prepared for an allocation error. Always check the value returned from the allocation function, and if it is 0, you will need to handle it cleanly, somehow. User-space code can be terminated with a segmentation violation if it ignores memory allocation errors, but the kernel can easily crash, bringing down the whole system.

There are several common error-handling strategies. One strategy is to attempt to allocate critical memory at the top of a function, where you are less likely to have committed yourself and can more likely return an error cleanly. This is usually the best way to handle the problem.

Another strategy, usually used together with allocation at the top of the function, is to allocate an "easy" amount of memory for the memory management system to provide, and then parcel it out for various purposes during the life of the function, effectively doing its own memory management. Several subsystems in the kernel do this, such as the high-level SCSI drivers and the network code. Both include special memory allocation functions which are only supposed to be used in those subsystem. These are not documented here, under the assumption that documentation for those subsystems should document subsystem-specific memory allocation routines.

Yet another strategy, which will only work if you are not in "critical" sections of code, is to allow the kernel to schedule another process by calling **schedule()** and then to try again later, when schedule returns. Note that some kinds of allocation are not safe to call even once from within critical code; that will be covered when we discuss the individual functions.

The fundamental rule is not to write algorithms that commit themselves to completing without having been guaranteed the resources they need in order to complete. Memory is one of the scarcest and most commonly needed of the resources that must be guaranteed, and the only way to guarantee that memory will be available is to allocate it.

Kmalloc

The **kmalloc()** function allocates memory at two levels: it uses a "bucket" system to allocate memory in units up to nearly a page (4Kb on the i86) in length, and uses a "buddy" system on lists of different sizes of contiguous chunks of memory to allocate memory in units up to 128Kb (on the i86) in length. Only in recent kernels has it been able to allocate memory in units over 4Kb in length, and allocating large amounts of memory with kmalloc is very likely to fail, especially in low-memory situations, and especially on machines with less memory.

Kmalloc is very flexible, as demonstrated by its calling convention:

```
void * kmalloc(unsigned int size, int priority);
```

Note the **priority** argument: this is what makes kmalloc so flexible; it is possible to use kmalloc in very constrained circumstances such as from an interrupt handler. Interrupt-driven code, or code that cannot be pre-empted, but still

needs to allocate memory, can call kmalloc with the **GFP_ATOMIC** priority. This will be more likely to fail, because it cannot swap or do anything else which would cause implicit or explicit I/O to occur. Code with relaxed requirements, which may legitimately be pre-empted, should instead call kmalloc with the **GFP_KERNEL** priority. This may cause paging and may cause **schedule()** to be called, but has a higher chance of success.

In order to dynamically allocate memory that can by accessed via DMA, the **GFP_DMA** priority should be used. It does stress the memory system, particularily if large amounts of memory are requested, and is quite likely to fail. Try again. It should be noted that **GFP_DMA** is only likely to fail on system with severe limitations on DMA transfers—such as computers using the common ISA bus. Not all platforms are affected by this problem.

Memory allocated with **kmalloc()** is freed with **kfree()** (or **kfree_s()**).

Vmalloc

For allocating large areas of virtually contiguous memory that do not have to be physically contiguous for interfacing with hardware, the new **vmalloc()** function (with the same calling convention as conventional **malloc()**) will cause less stress on the memory subsystem. It allocates possibly non-contiguous blocks of free memory, and maps them into one contiguous space in high memory. It is less likely to fail than kmalloc in many situations. It does not take a priority like kamlloc does. It cannot be called from within an interrupt, and it may implicitly cause pre-emption to occur.

Memory allocated with vmalloc is not DMA-able, even on systems without DMA restrictions, because DMA under Linux assumes a 1-1 logical-physical page mapping. This simplifies memory management in several ways, and is not a severe restriction, because kmalloc provides a way to get DMA-capable memory.

Just because it is addressed virtually does not mean that this memory is subject to paging to disk, despite rumors to the contrary. The “virtual” in vmalloc refers only to the addressing, which is not a 1-1 mapping from virtual to physical address space, unlike the rest of the kernel. Swapping may be initiated to **provide** the memory during a call to **vmalloc()**, but the vmalloced memory will not then be swapped out.

Memory allocated with **vmalloc()** is freed with **vfree()**.

get_free_pages

Now we learn what the **GFP** above stands for: **get_free_page** (well, perhaps **__get_free_pages**) and simply specifies how exactly this function goes about attempting to find free pages of memory. As you may guess, the same **GFP_*** values are used for these functions as well.

This is the way to request an amount of memory that is easy for the memory subsystem to allocate. This is the lowest-level—and therefore the lowest-overhead—way of dynamicly allocating memory. If you need a chunk of memory larger than half a page but no larger than a page (when deciding this, be aware that page size varies from architecture to architecture; it is 4Kb on the i86 and 8Kb on the DEC Alpha, for instance), especially if you only need it for the duration of the current procedure, this can be the right way to go. Also, if you are working on subsystem-specific memory management, you almost certainly want to allocate your memory this way.

If you want only one page, call **get_free_page(*priority*)**, where ***priority*** is one of the **GFP_*** values. Of course, the same rules about which **GFP_*** value is correct apply as for kmalloc. If you only want one page and don't care if it has been cleared (set to all zero values), use **__get_free_page(*priority*)** instead, since most of the overhead of allocating a page with **get_free_page** goes to clearing the page.

If you need to allocate more than one consecutive page, you can do so, although this is more likely to fail than allocating a single page, and the more pages you wish to allocate, the less likely you are to succeed. You can only allocate a number of pages which is a power of two. **__get_free_pages(*priority*, *order*)** is called with the same ***priority*** argument; the ***order*** argument gives the size according to the following formula: **PAGE_SIZE**<\!s>*<\!s>2<+>***order***<+> so an ***order*** of 0 gives one page, of 1 gives two pages, of 2 gives four pages, and so on (in current kernels, at least) up to 5, which gives 32 pages, which on the i86 architecture is 128Kb. **PAGE_SIZE**, as you may have guessed, is the standard macro for the number of bytes in a page.

The **__get_dma_pages()** function works exactly like **__get_free_pages()**, except that it allocates pages which are capable of being used for DMA, and it puts more stress on the memory allocation system.

Pages allocated with **get_free_page()** or **__get_free_page()** are freed with **free_page()**, and pages allocated with **__get_free_pages()** and **__get_dma_pages()** are freed with **free_pages()**.

Device initialization

Now we approach the deprecated strategies. They may be useful in some circumstances, mostly in situations where they are the “easy way” to get a driver working. In those cases, it is usually best to eventually find another way to do the same thing, as neither strategy is very flexible. Neither strategy is applicable to loadable modules.

When a device which is compiled into the kernel is initialized, it is passed a pointer to available memory. It is then required to return a pointer. If the pointer it returns is higher than the pointer it got, the memory in between the two pointers is reserved for the device. That memory will be in the first few megabytes of memory. The exact location will depend on how the kernel is booted. This is (perhaps unfortunately...) documented in the *Linux Kernel Hackers' Guide*.

Once allocated, that memory cannot be freed.

Memory initialization

This particular method is extremely deprecated, and is architecture-dependent as well. It is possible to add a function call within the body of **mem_init()**, which resides, for the i86 platform, in the file arch/i386/mm/init.c. In the middle of this function, two functions for initializing SCSI and sound-driver memory are provided. Also, arch/i386/kernel/head.S provides another platform-dependent way to allocate memory. This is where initial memory management is set up.

If you understand these well enough to muck with them, you don't need my help. These are last resorts for memory allocation, and you need to know *exactly* what you need to do, and *why* the dynamic allocation strategies will not work for you, before considering these “hacks”.

Michael K. Johnson is the Editor of *Linux Journal*, and pretends to be a Linux guru in his spare time. He can be reached via email as info@linuxjournal.com.

Archive Index Issue Table of Contents

Advanced search