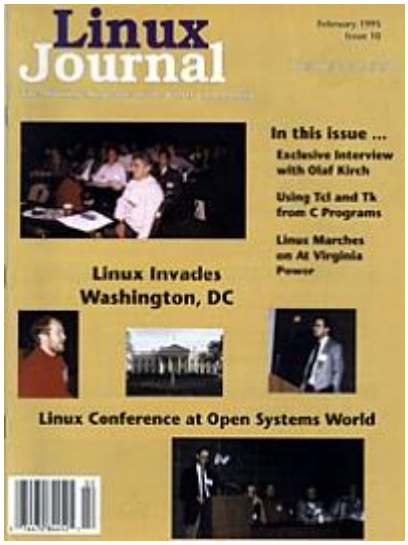
LINUX JOURNAL

Advanced search

Linux Journal Issue #10/February 1995



Features

A Conversation with Olaf Kirsch
: The author of the Network Administrator's Guide tells us a little something about his life and the NaG

Using Tcl and Tk from Your C Programs *by Matt Welsh*
: This month we'll show you how to use Tcl and Tk from your C programs.

Linux Conference at Open Systems World/FedUNIX 1994 *by Belinda Frazier*
: A remarkable conference with developers, support persons, resellers and end-users.

SCADA-Linux Still Hard at Work *by Vance Petree*
: Time marches on, Linux marches on, and one of the cardinal rules of the universe manifests itself.

News & Articles

Report on Comdex 1994 *by Belinda Frazier*

What Your DOS Manual Doesn't Tell You About Linux *by Liam Greenwood*

What's GNU? *by Arnold Robbins*

Columns

Letters to the Editor

Stop the Presses *by Phil Hughes*

New Products

Kernel Korner : Block Device Drivers: Interrupts *by Michael K. Johnson*

Archive Index

Advanced search

A Conversation with Olaf Kirch

LJ Staff

Issue #10, February 1995

The author of the Linux Network Administrator's Guide tells us a little something about his life and the NAG.

***Linux Journal*:** Tell us a little bit about yourself. How old are you? Where did you go to school and what did you study? What are your hobbies?

Olaf Kirch: I'm 28 and right now I'm working as a developer for a small company in the CAD/CAM business. I studied math and computer science at Darmstadt University, Germany, and graduated about one year ago.

One of the things I do in my spare time is, of course, tend Linux boxes, but I also read and paint a bit. And

I like bicycling. On my holidays, I love to go hiking with a backpack—the farther away from any terminal, the better.

***LJ*:** How did you start using Linux?

Olaf: I installed my first Linux system from an MCC Interim distribution some time back in 1992. Before that, my home box was an Atari running Minix, which was a little painful. When I heard of Linux, I instantly junked the Atari and bought a PC. I got the MCC release from some kind soul who offered a gratis copying service on German Usenet. You only had to send him seven floppies...

***LJ*:** As the principal author of the Linux Network Administrator's Guide, you have helped a lot of Linux users. How did you get started?

Olaf: I got into the whole project almost by accident. Initially, I meant to contribute only a UUCP chapter to the System Administrator's Guide. When I followed up with a chapter on smail and released it to the DOC channel on

Linux Activists, I mused aloud “Wouldn't it be nice to have an entire Networking Guide?” “Hey, great,” everyone said, “I'd say, go for it!” I was trapped.

Early on, progress was rather slow, because I wasn't as comfortable with English as I thought. But the people on the DOC channel were very helpful, and I got lots of reviews. My most important reviewer was Andy Oram at O'Reilly, who got involved in the book in late 1993.

Unfortunately, I also had to write my MS thesis, but in the end it worked out well. I'm only glad my prof never caught me in the terminal room.

***LJ*:** You say you “trapped” yourself into writing the NAG. Were you a networking guru when you started writing it?

Olaf: No, not at all. I had hacked on UUCP and various mailers like smail2 and umail, but my ideas about TCP/IP networking, etc., were rather foggy. To put it more bluntly, I was just another clueless newbie then. But I was very curious, so I got myself some books, pestered developers with stupid questions and spent endless hours browsing source code. BTW (by the way), that's one of the main lessons I learned from the whole thing: If you don't know how something works, read the source. It really helps.

***LJ*:** Do you have a network at home or just a single computer?

Olaf: Most of the time, I have only one computer at home, so I have to test a lot of things using only local loopback. I do have some friends, however, who are running networked Linux boxes and they always call me when they have a problem or want to install new applications. This way I can try out everything on their networks; this has the added benefit that if anything goes wrong, their machines go down, not mine.

***LJ*:** What is your connection to the Internet?

Olaf: I get mail and news via UUCP from brewhq, our domain's main hub that has an Internet uplink via ISDN. For interactive things like FTP, I use a SLIP link, but I hope to have ISDN, too, some time soon.

***LJ*:** What future do you see for the NAG? Do you intend to keep revising it to keep it up to date?

Olaf: Yes, I do, at least for a while. The basic network administration issues in Linux don't change that rapidly at the moment, so I think I'll release updates every few months.

Of course, I'm aware there's a lot going on that I didn't cover in the NAG, like IPX, ham radio and so on.

I also had offers from people who wanted to write something on sendmail V8, INN and the BSD automounter. On the other hand, I feel the book is already rather hefty, so

I'll probably not add any new sections. Maybe there will be a sequel, but don't hold your breath.

***LJ*:** Maybe a Basic Linux Network Administrator's Guide and an Advanced Linux Network Administrators Guide?

Olaf: I'm thinking more of a collection of papers, a little like the management documents in the 4.4BSD System Manager's Manual. I would want to make it less closed than the NAG itself, so that different people can contribute more easily. I had very firm ideas about how detailed the networking guide should be, up to the point that some people considered it “dumbed down”. Vince Skahan, who wrote the sendmail chapter never complained, but I think I badgered him quite a lot. For the sequel, I would lend people a hand at writing something, without imposing my views on them.

***LJ*:** What suggestions do you have for those people who are interested in learning about topics you decided not to cover? How can they learn about them?

Olaf: That depends. For some packages, like INN, a quite exhaustive FAQ is distributed on the Net. For sendmail V8, you can always get the bat book from O'Reilly. It's about the size of a brick, but very useful. The IPX and AX.25 stuff is still largely undocumented, so your best bet is to read the sources.

***LJ*:** What has Linux done for your professional life?

Olaf: Difficult question. People usually don't roll out the red carpet for you just because you say, “Hey, I wrote a book on Linux, why don't you hire me?” On my current job,

I don't get involved with network administration a lot. I'm mainly coding C++ and Motif applications in an HP environment, but I'm quite happy with it.

***LJ*:** In the preface of the NAG, you say that one of your favorite sports is “doing things with sed that other people would reach for their perl interpreter for.” Do you have a favorite sed hack you would like to tell us about?

Olaf: First, let me say that I don't believe perl is evil or anything. I just think that sed is more fun, just the way the Obfuscated C Code Contest is. My favorite sed hack is a short script I wrote that computes prime numbers: If you give it a number n on standard input, it will print all primes smaller than 2^n on standard out.

***LJ*:** Will you give any hints as to how you got that to work?

Olaf: There's nothing magic about it. The script is a simple sieve algorithm. The only tricky thing is incrementing and decrementing numbers. You can do that by shifting a marker from right to left, very much like a carry flag. Say I have 7890@ as input and want to decrement it. Then I replace 0@ with @9, and continue. Any other digit is decremented by one and the marker removed, i.e., 9@ becomes 8, 8@ becomes 7, etc. Quite silly, I admit.

***LJ*:** Thanks for taking the time to do this interview!

Archive Index Issue Table of Contents

Advanced search

Using Tcl and Tk from Your C Programs

Matt Welsh

Issue #10, February 1995

In the December issue, we introduced X Window System programming with Tcl and Tk, showing you how to write wish scripts for simple X-based applications. This month, we'll show you how to use Tcl and Tk from your C programs.

Tcl was originally designed to be an “extension language”--that is, an interpreted script language to be embedded in another program (say, one written in C) and used to either handle the mundane tasks of user customizations, or, with Tk, more complex tasks such as providing an X Window System interface for the program. The Tcl interpreter itself is simply a library of functions which you can invoke from your program; Tk is a series of routines used in conjunction with the Tcl interpreter. Although you can write Tcl/Tk programs entirely as scripts, to be executed via **wish**, this is only one side of the story. To really make this system shine you need to utilize Tcl and Tk from other programs.

Ousterhout's book, *Tcl and the Tk Toolkit*, contains exhaustive material on linking the Tcl interpreter with your C programs. What this generally entails is having your program produce or read Tcl commands from some source and pass the commands, as strings, to the Tcl interpreter functions, which return the result of evaluating and executing the Tcl expressions.

While this mechanism is certainly useful, there are several drawbacks. First of all, it requires the programmer to learn the details of interfacing their C code with the Tcl interpreter. While this is not usually difficult, it means that the programmer must not only work partly in C and partly in Tcl (which may be an unfamiliar language at first), but also learn the details of using the Tcl library routines. In most cases this requires the program to be reorganized to some extent—for example, the program's **main** function is replaced with a Tcl “event loop”.

The other drawback is that the Tcl and Tk libraries are literally huge—linking against them produces executables over a megabyte in size. Although there are now Tcl and Tk shared libraries available, this is a design concern for some.

The basic paradigm presented by this approach is that one implements new Tcl functions in C, and those Tcl functions can be called from a script which uses your program as a Tcl/Tk interpreter—a replacement for **wish** for your particular application.

My solution to this problem is perhaps less powerful, but also much more straightforward from the point-of-view of the programmer. The idea is to fork an instance of **wish** as a child process of your C program and talk to **wish** via two pipes. **wish**, being a separate process, isn't linked directly to your C program. It is used as a "server" for Tcl and Tk commands—you send Tcl/Tk commands down the pipe to **wish**, which executes them (say, by creating buttons, drawing graphics, whatever). You can have **wish** print strings to its standard output in response to events (say, when the user clicks a button in the **wish** window)--your C program can receive these strings from the read pipe and act upon them.

This mechanism is more in line with the Unix philosophy of using small tools to handle particular tasks. Your C program concerns itself with application-specific processing, and simply writing Tcl/Tk commands to a pipe. **wish** concerns itself with executing these commands.

This solution also gets around the problem of having a separate **wish** replacement for each application that you write using Tcl and Tk. In this way, all applications can execute the same copy of **wish** and communicate with it in different ways.

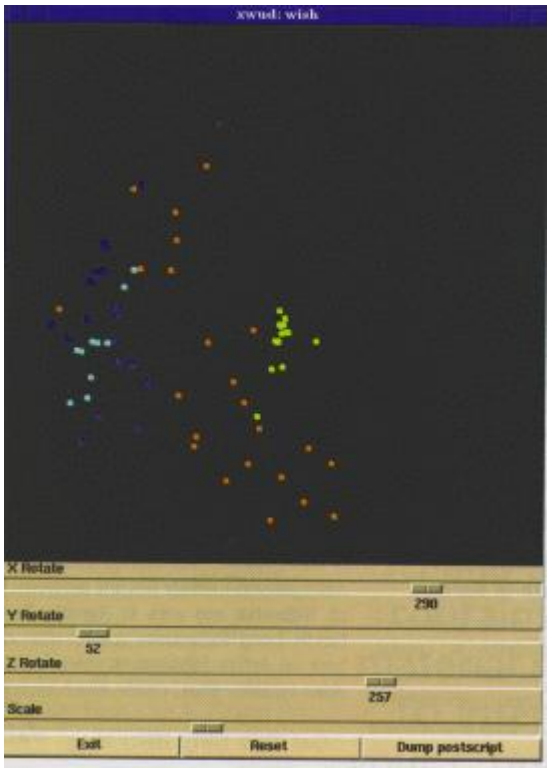


Figure 1

This month, I'm going to demonstrate a “real world” application which uses these concepts. My machine vision research at Cornell required me to visualize three-dimensional point sets. (For the curious, the problem dealt with feature classification: for each region in an image, five features were quantified, such as average intensity, Canny edge density, and so forth. The problem is to classify like regions by treating each region as a point in a five-dimensional feature space, and group regions together using the k-nearest neighbor clustering algorithm. I needed to take a 3D slice of this 5D space, assign a type to each point, and view it in realtime by rotating, scaling and so forth. This would allow me to verify that my features were clustering well.) Essentially, it's a simple scientific visualization program for the task at hand; this was much easier to write, using Tcl and Tk, than working with the large visualization packages that were available. Additionally, I could customize it to taste.

This program reads in a datafile consisting of 3D coordinates, one per point. Each point is also assigned a “type”, which is an integer from 0 to 6. Each point is given a simple 3D-to-2D transformation and plotted with a different color, based on the type. A **wish** canvas widget is used to do the plotting; **wish** provides scrollbars to allow you to rotate and scale the dataset. Figure 1 (above) shows what the program looks like on a sample dataset of about 70 points.

Note that the original version of this program contained other features, such as the option to display axes. I have trimmed down the code considerably in order for it to fit here.

The first thing that we need is some way to start up a child process and talk to it via two pipes. (Two pipes are used in this implementation: one for writing to the child, and one for reading from it. In the end I found this simpler than synchronizing the use of a single pipe.)

Here is the code, which I call **child.c**, to do this:

```
/* child.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/time.h>
#include "child.h"
/* Exec the named cmd as a child process, returning
 * two pipes to communicate with the process, and
 * the child's process ID */
int start_child(char *cmd, FILE **readpipe, FILE
        **writepipe) {
   int childpid, pipe1[2], pipe2[2];
   if ((pipe(pipe1) < 0) || (pipe(pipe2) < 0)) {
        perror("pipe"); exit(-1);
}
   if ((childpid = vfork()) < 0) {
     perror("fork"); exit(-1);
   } else if (childpid > 0) {  /* Parent. */
     close(pipe1[0]); close(pipe2[1]);
     /* Write to child is pipe1[1], read from
      * child is pipe2[0].  */
     *readpipe = fdopen(pipe2[0],"r");
     *writepipe=fdopen(pipe1[1],"w");
     setlinebuf(*writepipe);
     return childpid;
   } else {  /* Child. */
     close(pipe1[1]); close(pipe2[0]);
     /* Read from parent is pipe1[0], write to
      * parent is pipe2[1].  */
     dup2(pipe1[0],0);
     dup2(pipe2[1],1);
     close(pipe1[0]); close(pipe2[1]);
     if (execlp(cmd,cmd,NULL) < 0)
        perror("execlp");
     /* Never returns */
} }
```

If you're familiar with Unix systems programming, this is a cookbook function. We use vfork (fork would do as well) to start a child process, and in the child execlp the command passed to the function. The command passed to start_child must be on your path when using this function; also, you can't pass command-line arguments to the command. It's easy to add the code to do this, but we don't show this here for sake of brevity.

We use dup2 to connect the child's standard input to the write pipe, and the child's standard output to the read pipe. In this way anything that the child prints to stdout will show up on readpipe, and anything the parent writes to writepipe will show up on the child's stdin. In the parent, we use fdopen to treat the pipes as stdio FILE pointers, and setlinebuf to force the write pipe to be flushed whenever we send a newline. This saves us the trouble of using fflush each time we write strings to the pipe.

The header file, child.h, simply contains a prototype for start_child. It should be included in any code which uses the above function.

```
#ifndef _mdw_CHILD_H
#define _mdw_CHILD_H
#include stdio.h
#include sys/types.h
#include sys/time.h
extern int start_child(char *cmd,
        FILE **readpipe, FILE **writepipe);
#endif
```

Now, we can write a C program to call start_child to execute **wish** as a child process. We write Tcl/Tk commands to writepipe, and read responses back from **wish** on readpipe. For example, we can have **wish** print a string to stdout whenever a button is pressed or a scrollbar moved; our C program will see this string and act upon it.

Here is the code, **splot.c**, which implements the 3D dataset viewer.

```
/* splot.c */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include "child.h"
#define Z_DIST 400.0
#define SCALE_FACTOR 100.0
/* Factor for degrees to radians */
#define DEG2RAD 0.0174532
typedef struct _point_list {
   float x, y, z;
   int xd, yd;
   int type; /* Color */
   struct _point_list *next;
}  point_list;
static char *colornames[] = { "red",
        "blue", "slateblue", "lightblue",
        "yellow", "orange",
        "gray90"
};
inline void matrix(float *a, float *b,
                float sinr, float cosr) {
        float tma;
        tma = *a;
        *a = (tma * cosr) - (*b * sinr);
        *b = (tma * sinr) + (*b * cosr);
}
void plot_points(FILE *read_from, FILE *write_to,
        point_list *list, char *canvas_name,
        float xr, float yr, float zr,
        float s, int half) {
point_list *node;
float cx, sx, cy, sy, cz, sz, mz;
float x,y,z;
xr *= DEG2RAD; yr *= DEG2RAD; zr *= DEG2RAD;
s /= SCALE_FACTOR;
cx = cos(xr); sx = sin(xr);
cy = cos(yr); sy = sin(yr);
cz = cos(zr); sz = sin(zr);
for (node = list; node != NULL;
        node = node->next) {
   /* Simple 3D transform with perspective */
   x = (node->x * s); y = (node->y * s);
   z = (node->z * s);
   matrix(&x,&y,sz,cz); matrix(&x,&z,sy,cy);
   matrix(&y,&z,sx,cx);
   mz = Z_DIST - z; if (mz < 3.4e-3) mz = 3.4e-3;
   x /= (mz * (1.0/Z_DIST));
```

```
   y /= (mz * (1.0/Z_DIST));
   node->xd = x+half; node->yd = y+half;
}
/* Erase points */
fprintf(write_to,"%s delete dots\n",canvas_name);
for (node = list; node != NULL;
        node = node->next) {
   /* Send canvas command to wish... create
    * an oval on the canvas for each point. */
   fprintf(write_to,
        "%s create oval %d %d %d %d " \
        "-fill %s -outline %s -tags dots\n",
        canvas_name,(node->xd)-3,(node->yd)-3,
        (node->xd)+3,(node->yd)+3,
        colornames[node->type],
        colornames[node->type]);
}
}
/* Create dataset list given filename to read */
point_list *load_points(char *fname) {
   FILE *fp;
   point_list *thelist = NULL, *node;
   assert (fp = fopen(fname,"r"));
   while (!feof(fp)) {
        assert (node =
            (point_list *)malloc(sizeof(point_list)));
        if (fscanf(fp,"%f %f %f %d",
                &(node->x),&(node->y),&(node->z),
                &(node->type)) == 4) {
            node->next = thelist;
            thelist = node;
        }
}
   fclose(fp);
   return thelist;
}
void main(int argc,char **argv) {
   FILE *read_from, *write_to;
   char result[80], canvas_name[5];
   float xr,yr,zr,s;
   int childpid, half;
   point_list *thelist;
   assert(argc == 2);
   thelist = load_points(argv[1]);
   childpid = start_child("wish",
                &read_from,&write_to);
   /* Tell wish to read the init script */
   fprintf(write_to,"source splot.tcl\n");
   while(1) {
        /* Blocks on read from wish */
        if (fgets(result,80,read_from) <= 0) exit(0);
                /* Exit if wish dies */
        /* Scan the string from wish */
        if ((sscanf(result,"p %s %f %f %f %f %d",
                canvas_name,&xr,&yr,&zr,
                &s,&half)) == 6)
           plot_points(read_from,write_to,thelist,
   else
        fprintf(stderr,"Bad command: %s\n",result);
   }
}
```

To build the above program (call it splot) you can use the command:

```
gcc -O2 -o splot splot.c child.c -lm
```

You should find splot to be fairly straightforward.

The first thing we do is read the data file named on the command line, using the load_points function. This function reads a file which looks like the following:

```
-50 -50 -50 0
 50 -50 -50 1
-50  50 -50 2
-50 -50  50 3
-50  50  50 4
 50 -50  50 5
 50  50 -50 1
 50  50  50 2
```

(This particular dataset defines a cube. The fourth column indicates the type, or color, of each point.) **load_points** reads each line and returns the values as a linked list of type **point_list**. Next, we use **start_child** to fire up wish. Anything written to **write_to** will be read by wish as a Tcl/Tk command. First we send the command source **splot.tcl**, which causes wish to read the script **splot.tcl**, shown below.

```
# splot.tcl
option add *Width 10
# Called whenever we replot the points
proc replot val {
        puts stdout "p .c [.sf.rxscroll get] \
                           [.sf.ryscroll get] \
                           [.sf.rzscroll get] \
                           [.sf.sscroll get] 250"
        flush stdout
}
# Create canvas widget
canvas .c -width 500 -height 500 -bg black
pack .c -side top
# Frame to hold scrollbars
frame   .sf
pack .sf -expand 1 -fill x
# Scrollbars for rotating view. Call replot whenever
# we move them.
scale  .sf.rxscroll -label "X Rotate" -length 500 \
 -from 0 -to 360 -command "replot" -orient horiz
scale   .sf.ryscroll -label "Y Rotate" -length 500 \
 -from 0 -to 360 -command "replot" -orient horiz
scale   .sf.rzscroll -label "Z Rotate" -length 500 \
 -from 0 -to 360 -command "replot" -orient horiz
# Scrollbar for scaling view.
 scale .sf.sscroll -label "Scale" -length 500 \
  -from 1 -to 1000 -command "replot" -orient horiz \
  -showvalue 0
 .sf.sscroll set 500
# Pack them into the frame
pack  .sf.rxscroll .sf.ryscroll .sf.rzscroll \
      .sf.sscroll -side top
# Frame for holding buttons
frame .bf
pack    .bf -expand 1 -fill x
# Exit button
button .bf.exit -text "Exit" -command {exit}
# Reset button
button .bf.sreset -text "Reset" -command \
   {.sf.sscroll set 500; .sf.rxscroll set 0;
    .sf.ryscroll set 0; .sf.rzscroll set 0; replot 0}
# Dump postscript
button .bf.psout -text "Dump postscript" -command \
   {.c postscript -colormode gray -file "ps.out"}
# Pack buttons into frame
pack  .bf.exit .bf.sreset .bf.psout -side left \
   -expand 1 -fill x
# Call replot
replot 0
```

Nearly everything in this script was introduced in the December issue; if you can't follow it, check the Tcl/Tk manpages for scrollbar, button, and so forth (or order back issues).

After telling wish to read **splot.tcl**, the program goes into a read loop, using fgets to read lines from the read pipe. This causes splot to sleep until there is data on the pipe to be read. If you wanted your program to continue running while waiting for output from wish, there are several alternatives. You could call select to poll for pending data on the pipe, or you could set the pipe to use non-blocking I/O (see the man page for **fcntl**). Any book on Unix systems programming can help.

Whenever the scrollbars are moved, they call the replot function within **splot.tcl**. This prints a line beginning with the letter "p", followed by the name of the canvas widget to draw to, the values of the rotation and scale scrollbars, and the half-height of the canvas widget. This latter is used to center the image in the canvas when it is drawn.

Note that we must flush stdout after writing a command to it. Otherwise the commands would be buffered and not sent immediately to **splot**.

Once splot receives this line, it uses **sscanf** to parse the values and calls **plot_points**. This function implements a very simple, but relatively fast, 3D perspective transform, and applies it to each point. For each point, we send wish a **canvas** command to create an oval object based upon its 2D location after the transform. The variable half is used to center the point set on the canvas. The **colornames** array is indexed with the type field of each point structure to set the color.

There you have it! A complete visualization program in a few kilobytes of C and Tcl code. Try it out: Enter the above code, compile it, and run the program as **splot cube.dat** where **cube.dat** contains the dataset for the 3D cube given above. You should be able to tumble and scale the cube in the wish window. On my systems, this is remarkably fast—I can view datasets of several hundred points with very little noticeable lag.

However, the idea here is to code all of the speed-critical parts of the program in C, and allow wish to handle just the user interface. Remember that Tcl and Tk passes everything around as scripts, so the tighter your Tcl code, the better. For example, note how we do the degree-to-radian conversion and point scaling in the C code. Using a **Tcl expr** command to do the same would require greater overhead.

There are many possible extensions to this program. For example, you could add buttons or additional scrollbars to **splot.tcl** which would cause other kinds of commands to be printed to wishs **stdout**. The read loop in splot, for example, could do a switch based on the first character of the line received

from wish and perform different actions based on that. As long as your C code and Tcl script agree on the command format used, you're “cooking with gas”.

Please feel free to get in touch with me if you have questions about this code or problems getting it to work for you. I suggest picking up a copy of John Ousterhout's book *Tcl and the Tk Toolkit*, from Addison-Wesley, as well as a book on Unix systems programming, which will cover the details of using pipes for interprocess communication.

Until next time, happy hacking.

Matt Welsh (mdw@sunsite.unc.edu) Matt Welsh is a systems hacker and writer, working with the Linux Documentation Project.

Archive Index Issue Table of Contents

Advanced search

Linux Conference at Open Systems World/FedUNIX'94

Belinda Frazier

Issue #10, February 1995

A remarkable conference with developers, support persons and resellers results in a successful, information-packed event.

Open Systems World/FedUNIX'94 in Washington DC, in its sixth year, included several conferences and classes such as the FedUnix Sessions, Motif/COSE Users Conference, Novell AppWare Developers, SCO, Solaris, Windows NT, and World Wide Web/Mosaic conferences. For the first time, on December 1 and 2, Open Systems World offered a Linux International Users and Developers Conference and a one-day Linux Tutorial.

The suggestion to include a Linux conference at Open Systems World was made by Thomas Sterling, the Acting Director of CESDIS (Center of Excellence in Space Data and Information Sciences). Alan Fedder, the director of Open Systems World, made the choice to bring Linux to Open Systems World/ FedUNIX'94.

Linux Journal developed and sponsored the Linux conference with immeasurable help from the speakers and other volunteers. One article cannot begin to include all the information presented at the sessions and the tutorial; future articles will include a more in-depth and technical report about each session.

Approximately 50 people signed up for the conference and the tutorial, but there were 67 people, including speakers, present during the one-day conference. Included in the conference were several panel discussions interlaced with presentations from experts on particular subjects.

Panel: What Should the Relationship Be Between Linux Resellers and the Linux Development Community?

Robert Young of ACC Bookstore moderated four panel members: Eric Youngdale and Donald Becker representing developers and Dan Irvin of Linux Systems Laboratories and Mark Horton, a support person for InfoMagic, representing resellers.

Eric Youngdale

Audience during panel discussion.

The opinions of all the panelists seemed to be that a cooperative relationship existed, and would continue to exist, between developers and resellers. There was mild dissension about Linux support when the moderator commented that the user couldn't get, or shouldn't expect to get, Linux support for the low price of a Linux distribution on CD; this was disputed by Mark Horton, support person for InfoMagic, who replied that the InfoMagic CD-ROM included support, and discussed the kind of support they were giving. Horton added that, in general, the support wasn't getting abused/overused by the end user.

There was a feeling of good will between developers and resellers. This was definitely a panel discussion, not a panel argument.

Panel: The Commercial Future of Linux

Moderated by Michael K. Johnson, the four panelists included Marc Ewing of Red Hat Software, Mark Komarinski who writes the “System Administration” column for *Linux Journal*, Ross Biro of Yggdrasil Computing and Mark Bolzern of WorkGroup Solutions.

Mark Horton, Mark Komarinski and Phil Hughes

They discussed the current commercial products available for Linux, obstacles to the development of commercial products, and how commercial vendors might be persuaded to port their product to Linux and market it.

There are already about 50 commercial applications available for Linux, including Tecplot, ISE Eiffel, Genplot, dBMan, ESQLFLEX and JustLogic Database Manager. Although Linux users expect relatively inexpensive applications, it's not profitable for commercial marketers to produce such low-cost applications due to development and marketing costs. Distribution channels can be a significant cost factor in inexpensive applications, as well as the cost of advertising, which can, for example, run $3,500 for a one-time quarter page ad in a popular Unix magazine.

Mark Bolzern said that while Linux is becoming “the Unix of choice”, Linux is not yet trusted to be mission-critical. Bolzern anticipates, however, that over the next year, Linux will be used in this capacity, such as the application currently in use at Virginia Power for real-time data collection.

Ross Biro stressed the importance of hardware in the commercial future of Linux. Except for Cyclades who makes serial boards, there aren't enough hardware vendors making their products' specifications available for Linux developers. There was general agreement among audience members, the panel and the moderator about the need for more hardware support for Linux. (However, it's rumored that half of Cyclades' domestic sales of one of their serial boards is to Linux users.) Mark Komarinski noted that having most of the kernel written in C (as it is now) will help with porting to hardware.

Some panelists and audience members added positive comments about the two ports for the DEC alpha chip in progress: one being done by DEC, the other by Linus Torvalds on a machine loaned to him by DEC.

Mark Bolzern talked about how his company was investing in its Linux application now and pricing it far below the actual price the development and marketing costs would allow for such a product, because they anticipate that the future volume of sales to the Linux market will make this worth their investment.

Bolzern advised that his company is paying a PR firm to promote Linux, with the idea that increased sales of their product will follow as Flagship demos become available on many more of the different Linux distributions on CD-ROM. To help with the PR work, he asked for success stories, such as how Linux users replaced their whole network with Linux systems, or did something with Linux that could not be done any other way. Please e-mail to mark@linuxmall.com

Dr. Greg Wettstein

Linux and NASA: Project Beowulf

Don Becker, who wrote most of the Ethernet drivers for Linux, is the principal investigator on a new project at NASA called Beowulf, a cluster of Linux processors, connected by parallel Ethernets. He discussed the project with an enthralled audience.

Other Topics

After lunch, participants returned for an inspiring talk on How To Convince Your Boss/Employer/Customer To Use Linux. Dr. Greg Wettstein from the Roger

Maris Cancer Center (see Issue #5 of *Linux Journal* for his article about their Linux system) discussed a planned, reasonable way to present Linux to someone as a solution. He noted you should identify a specific problem that Linux can fix, explain how Linux can fix it, emphasize Linux advantages (for example, having source code available so you can make changes, its built-in networking and its support community). Don't try to replace an entire working system with Linux in one fell swoop—he emphasized, “Evolution, Not Revolution”.

Other subjects in the conference were: WINE presented by Bob Amstadt, Linux and The X Windows System presented by Przemek Klosowski and Linux and iBCS2 Compatibility by Eric Youngdale. iBCS2 defines a common object program format—a standard for PC Unix executables. The iBCS2 compatibility libraries will allow existing PC Unix applications to run on a Linux platform.

Panel: Commercial Use of Linux

This panel discussion included Vance Petree, who is using Linux for real-time data collection at Virginia Power; Russell Carter, Sandia Labs, using Linux for a super workstation; Greg Wettstein, of the Roger Maris Cancer Center, who uses Linux for a Patient Information System, written using Perl and Tcl/Tk; Donald Becker, NASA, who is developing a cluster of Linux stations; Paul Tomblin formerly of Gandalf, who is using Linux to build test tools for testing Gandalf's networking products.

The audience at all the talks was attentive. The one-day tutorial included experts speaking on their particular area of expertise and will be covered in other articles. All in all, the conference felt like a big success with an amazing amount of information presented.

Letters about the Conference

Archive Index Issue Table of Contents

Advanced search

Linux in the Real World

Vance Petree

Issue #10, February 1995

Last month I described the considerable success Virginia Power has enjoyed using Linux as the basis for a distributed data collection and archiving system. Well, Time marches on, Linux marches on and one of the cardinal rules of the Universe manifests itself: Anything that works tends to get used. A lot. And frequently in unexpected ways. Linux, blissfully so, is no exception.

This article describes an exciting and important new system that is being built on Virginia Power's Linux platform—a virtual SCADA (Supervisory Control and Data Acquisition) system that will provide a cost-effective and flexible alternative to traditional SCADA connectivity. Whereas last month's story was an epic software adventure with a cast of several, this month's story is more of a documentary of the new additions being added to our Linux house. As this is being written, the frame is up, the roof is on, and the drywall is in place. However, the joists are still visible and a good bit of plumbing has yet to be added. Don't worry—with virtual hard hats in place and source code hammers handy, we should be able to visualize the finished rooms easily. (Besides, as you read this, the system is finished. Magazines are wonderful time machines.)

But first, for those of you who may have missed last month's article, a little foundation on SCADA. As far as electric utilities are concerned, SCADA means the retrieval of real-time analog and status data from various locations in the service territory through remote terminal units (RTUs) installed in substations. This information is obtained by central master computers, where it is stored, analyzed, and presented to system operators who are responsible for maintaining the integrity and reliability of the transmission and distribution grid. When necessary, these operators can also remotely operate field devices like line breakers and capacitor banks by sending control commands from the master computer out to the RTUs. The master computers themselves even contain feedback algorithms that automatically operate some devices based on system conditions.

At Virginia Power, the standard medium of communication between RTUs and SCADA master computers is the dedicated serial line, often leased from the local phone company. Several RTUs can be multi-dropped off a single dedicated line (up to 16, a limitation imposed by our currently-used SCADA protocol), but geographical limitations tend to prevent as much sharing of dedicated lines as might be desirable. At the risk of over-simplifying, we can imagine one dedicated line per RTU, giving us a traditional SCADA system something like that shown in Figure 1 (see below).

The advantages of a dedicated connection are pretty obvious: constant data availability and quick response when system conditions require a control action of some sort (such as opening or closing a breaker or capacitor bank). In the case of generation stations or large, high-voltage substations, any other type of monitoring is unthinkable.

Yet, there are other likely monitoring sites, often in remote locations (the exact technical phrase is “in the middle of nowhere”) which are not quite so high profile (or high pressure). As a matter of fact, in a data acquisition sense, these potential sites are downright *prosaic*: two or three analog points, a couple of status points, and perhaps a single control point. Such modest monitoring needs don't justify the constant watchfulness a dedicated serial line provides, but the information does need to be retrieved; the control capabilities do need to be available when needed.

Figure 1. Traditional SCADA Connectivity

Figure 2. A Hybrid SCADA System

Over the years some partial solutions have been implemented. In many cases, intelligent electronic devices such as digital relays can monitor a small number of analog and status devices and supply sufficient control capabilities. These relays usually implement a simple serial-based protocol; installed in remote sites along with modems, they can be interrogated from the SCADA master computer centers using stand-alone PC packages, resulting in a hybrid SCADA system as shown in Figure 2 (see above).

Such a hybrid system, while it may provide (in one form or another) all necessary data and control capabilities, fails to provide the system operators with a unified picture of the system they are operating. These folks have enough responsibility on their hands without having to run through mental gyrations along the lines of: “I'd better check the voltages in the Berry district. Oops—the Cranberry substation has to be dialed, so I'll just walk over here to the dialup PC and...oh, heck. Ted's using the PC to dial the Mineral Water substation! Guess I'll try later...”

Of course, a second dialup PC could be purchased and then a third and a fourth and—Whoa—just a minute! What about those eminently reliable and flexible Linux systems in each and every SCADA master computer center? (I know you saw this coming.) Not only are those Linux systems handling averaged analog data for the Asset Management database system, but they are also an eminently reliable and flexible dialing subsystem (I blush to admit this, but there you are) which can potentially talk to any type of device with a byte-oriented protocol!

The dialing subsystem has no set limit on the number of phone lines it can handle. If some means could be found to move data back and forth between the SCADA master computers and the Linux systems, a more ideal SCADA system could be constructed, as shown in Figure 3 (see below). This would provide the system operators with a unified overview of their system—all information would be present in the SCADA master computer. Some of it would be retrieved via traditional means (dedicated lines) and the rest would be obtained via dialup connections through the Linux systems.

Figure 3. Ideal, Unified SCADA System

As you can probably guess, this last approach is pretty much the one we're taking, although a few sobering, but fortunately not insurmountable, realities of the Real World have intruded:

- Our SCADA master computers are older machines which are approaching the end of their digital careers. There are no spare processor cycles (or memory bytes) for any kind of special programs to accommodate talking with our Linux systems. In fact, the only feasible way to move data between our Linux systems and our SCADA computers is by using the same protocol as is used to communicate with our RTUs. Alas, this protocol is more than a little antiquated and uses special-purpose modems and encoding firmware.
- Dialing devices and retrieving data is all well and good, but sometimes system operators need to monitor data points continuously for a certain period of time.
- When operators perform control actions on remote devices, they need to see immediate feedback to determine the success or failure of the controls. Some actions can potentially affect several different data points, and these need to be updated in real time until the operators are satisfied as to the results of their control actions.

With regard to the first item, we are lucky enough to have available an RTU platform for which our group develops field-resident applications, such as closed-loop feedback controls and protocol translators for IEDs. This platform,

obviously, contains all of the requisite firmware and hardware for talking to our SCADA master computers and is fully programmable (in C, thank goodness). Stripped of all unnecessary peripheral hardware and loaded with a simple byte-oriented protocol to talk to our Linux systems over a null modem cable, this programmable RTU functions quite handily as a translator box: status and analog data can be delivered from dialup devices to the SCADA computer, and control requests can be delivered from the SCADA computer to the Linux system for appropriate action. Of course, all the SCADA computer knows is that it is scanning another RTU. The result is a slightly tempered ideal system as shown in Figure 4.

Figure 4. Unified SCADA System, Real World Edition

At this point, I'd like to mention a few details of software carpentry which will be important when we discuss the remaining items in our Real World reality list. The database of the translator box (i.e., the stripped-down programmable RTU which talks to the SCADA master computer) is organized as a set of arrays of data structures—one array for status points, another for analog points, etc. On the Linux side, a corresponding set of shared-memory partitions mirrors the arrays of data structures in the translator box—one partition for status points, another for analogs, etc. A daemon process in Linux talks to a counterpart process on the translator box and ensures that the corresponding instances of structure arrays remain consistent and up-to-date. This update process runs every few seconds.

“Every few seconds” may sound a trifle vague in connection with real-time data processing, but SCADA activity tends toward the leisurely side of real-time processing; RTUs are scanned once every 2 to 30 seconds, contact closures during control actions may be on the order of several hundred milliseconds to a second or two. So even though Linux (like any standard Unix system) is not strictly speaking a real-time system, it is more than responsive enough for the scale of real-time processing with which we're concerned.

Well, now—the important points to remember are these: A change to data in a shared memory segment in the Linux system will show up in the translator box, where it will be picked up and scanned by the SCADA master computer, eventually showing up on an operator display. Conversely, an operator-control action will change data in the translator box, which will show up on the Linux side and ring a (virtual) bell to cause some action to take place. From now on, we'll ignore the translator box and pretend that the SCADA master computer and Linux system are speaking directly to one another.

Which brings us, in a roundabout way, to the second Real World item. As noted before, continuous data monitoring is one of the advantages of dedicated-line

connectivity. Simulating dedicated-line access with regular phone lines is obviously why we're calling our new system a virtual SCADA system, and the basic principle is just as obvious: when continuous monitoring is needed for a dialup device, dial up and stay dialed up!

Of course, at any given time, it is only possible to continuously monitor as many dial-up devices as there are available phone lines—but more phone lines can always be added, should the need arise. We're starting with three dial-up serial ports per Linux machine; time and experience will tell if we need to add more. But some complications arise (as always) in the details. For example, what happens if an operator starts continuously monitoring a dial-up device, gets caught up in some other task, and forgets to release the device so the dial-up line can be used for some other purpose? For that matter, how does the operator start and stop monitoring a device in the first place?

To handle these details, each dial-up device has associated with it a number of pseudo status, analog and control points—points which have nothing to do with the data being monitored by the device, but rather are related to the device itself:

- A *timestamp* analog point, showing how old the device data is (i.e., the last time the device was called).
- A *connection* status point, showing whether the device is on-line or not.
- A *dial-up* control point. Toggling this control will cause the device to be dialed and a connection established.
- A *connect-time* analog point, showing how many minutes remain before the device is automatically disconnected.
- An *add-connect-time* control point. Toggling this point will add a fixed number of minutes to the connect-time analog, keeping the device on-line longer.
- A *disconnect* control point, to disconnect from the device immediately.

An additional pseudo-analog point reports the number of available dial-up lines. This analog point is displayed, along with the above-described pseudo points, for all dial-up devices on a SCADA master computer screen, allowing the system operator easy, centralized management of all dial-up devices.

As an example, let's replay our hypothetical scenario from a few paragraphs back: “I'd better check the voltages in the Berry district. Lessee—the Cranberry substation has to be dialed, so I'll just poke this control point right here at the comfort of my workstation...”

A little time passes while the device is dialed; the operator stays busy with other things. Then the connect pseudo-status changes state and dings an alarm beeper to attract the operator's attention: “Hmm...Cranberry's online now. I'd better keep an eye on those voltages for half an hour or so. I'll poke this add-time control a couple of times...There we go; now I've got 30 minutes of connect time.”

Okay, so it's not a perfect solution; the operator still has to perform some special actions to get his data, and has to know what's a dial-up device and what's not. But all of these extra activities can be done at the operator's regular workstation. And if dial-up devices are scheduled for periodic interrogation, some of these special actions may not even be necessary: “I'd better check the voltages in the Berry district. Say, it looks like the Cranberry substation was interrogated just 10 minutes ago—recently enough that I can use those values...”

As you might imagine, handling pseudo-points and connection timers involves much delightful software development on the Linux side, some of which is still in the blueprint stages and some needing only a final coat of symbol-stripping paint. The solution to the final item in our list of Real World realities—providing operator controls to dial-up devices—is still, pretty much, in the blueprint stage, but we can at least describe the basic ideas.

The main problem with dial-up device controls is providing sufficient generalization so that control actions are handled in a consistent manner. The standard method for controlling SCADA devices is a three-step select-verify-execute procedure: select the point to be controlled, verify the selection (usually by having the remote device echo the selection back to the master computer), and execute the desired control after a final go-ahead by the operator. The result of a control action is usually determined by monitoring an associated status point or one or more analog points.

Unfortunately, many of the intelligent end devices we are handling using virtual SCADA don't have clear- cut sequences of steps for performing control actions. One device, for example, uses an ASCII-encoded bitmap to select the device and execute the control, all in one step—so much for verification. Another device implements the usual 3-step procedure but with the added onus of sequence numbers to ensure no more than one outstanding control action at a time (actually, not at all a bad idea, but incompatible with our existing SCADA protocol). And there is the obvious prerequisite, that the device to be controlled must be on-line before any control is attempted.

A little poor-man's object-orientation seems to be in order, so we have abstracted the basic elements of a control request and, along the way, added a

few more pseudo-points per dial-up device (these additional pseudo-points are displayed on the same screen as all the other device pseudo-points):

- A *connection-in-progress* status point, which toggles true if the associated device is in the process of being dialed.
- A *control-in-progress* status point, which is true if a control is being performed on the associated device.
- A *control-success* status point, showing success or failure of the last control attempted.

The operator can perform a control on any dial-up device control point, just as he does with any other (dedicated-line) control point, with the understanding that his control action is actually a request for the control to be selected, verified, and executed on his behalf at some time in the (near) future. This distinction may seem cosmetic, but it is actually important, from an operational point of view.

Here's the general sequence of events: The operator controls a dial-up device control point (using the usual 3-step procedure, since he is communicating his request to the Linux system using the regular SCADA protocol), which toggles a database point in the Linux system, alerting the system that there's work to be done. The system sets the *control-in-progress* status point to ensure that only one control request per device is outstanding at a time. Since the number of separate control points per dial-up device is small, this restriction should not pose a problem.

If the device to be controlled is not online, it is dialed and a connection is established (the *connection-in-progress* status point allows monitoring of this process). If the device is already on-line, a set amount of time is added to its *connect-time* analog to allow for completion of the control request.

Device-specific software, knowing all the secrets for successful control actions on the device, performs the control requested by the operator, and reports the general success or failure through the *control-success* status point. The device remains on-line until its *connect-time* analog counts down to zero, allowing the operator an opportunity to observe any associated analogs or status points to verify that the control action has the desired effect.

Well, we've just about reached the end of our tour of this latest addition to our network of Linux systems, and I hope you've gotten a good idea of what the finished rooms will look like and the wonderful view we'll have of the increased efficiency and reliable operation of our SCADA systems. But the most remarkable feature of this new system hasn't yet been mentioned, although it has been implied in everything discussed so far.

Linux has become an integral, accepted part of the toolkit we use to craft solutions for the division and planning personnel who come to our group with problems and needs. During the design of our virtual SCADA system, no one suggested using some “other” operating system platform or questioned whether Linux would have enough horsepower to handle the new demands that would be placed upon it. A year of stellar, faultless Linux performance as our data-collection front ends has turned skepticism to happy acceptance and transformed the phrase “that PC Unix” to “our Linux systems.” Folks I've never met who work in our company, call up with Linux questions because they've heard good things about our systems.

Oh, we still have a skeptic or two—I'm sure we always will. But the surest way I've found to get them off my back, after they've expounded on the next release of “Ontario” or “Pookeepsie 96” or “Shangra-La”, is to cough politely and reply, “Well, Linux does that right now. And it works. Right now. See?”

The ones that come back, I tell `em how to get a good CD-ROM distribution. One more happy Linuxer can't hurt!

Vance Petree (vpetreeinfi.net) Although he began adulthood as a music composition major, Vance soon found computers a more reliable means of obtaining groceries. He has been a programmer for Virginia Power for the past 15 years, and lives with his wife (a tapestry weaver—which is a lot like programming, only slower) and two conversant cats in a 70-year-old townhouse deep in the genteel stew of urban Richmond, VA.

Archive Index Issue Table of Contents

Advanced search

Report on COMDEX '94

Belinda Frazier

Issue #10, February 1995

>First-time Linux representation, interesting application demonstrations, a vast array of materials and excited users—all contributed to a productive Comdex show.

Braving the crowds last November, I joined over 200,000 participants at COMDEX in Las Vegas, Nevada. The largest computer trade show in the world, COMDEX offered 2,200 exhibitors plying their new computer, multimedia and/or communication products at numerous convention sites and on multiple convention floors.

As if that wasn't enough, a PowerPC Pavilion brought together dozens of other exhibitors running their applications on PowerPC systems.

Surprisingly, given the almost overwhelming amount of information offered, there usually isn't much about Unix at COMDEX. This year, however, I was very pleased to find Linux represented at two booths at the show. Both Yggdrasil Computing, Inc., and Morse Telecommunication had Linux in their companies' banner.

I spent time at both of these booths to see first-hand the responses people had to Linux. The two most evident responses were easily categorized into the skeptics and the fanatics. The fanatics obviously knew about—and used—Linux. The skeptics were usually those who had barely tried Unix and didn't like it—but even so, they had trouble believing it could be free. I saw a few skeptics walk away with information about Linux products.

Dan Quinlan, Adam Richter and Corrine Butleau staffed the Yggdrasil Computing Booth at the Sands Expo and Convention Center. They demonstrated various applications on Linux, including the Microsoft Windows Solitaire game running under WINE (the Windows Emulator for Linux). They

also had the company's latest version of The Linux Bible, The GNU Testament, available for perusal and were also giving away copies of *Linux Journal*.

Adam said that he thought the most important news that week was that Linux was moving to ELF (Executable Loadable Format) as the binary format. ELF is the standard format used by most PC Unix implementations, such as SVR4. Yggdrasil's Winter, 1995, release of Plug & Play, which will be available by the time you read this, will have ELF binaries.

ACC Bookstore and Morse Telecommunication shared a booth, staffed by Michael Johnston, Robert Young and Pat Volkerding. Pat, who was interviewed in *Linux Journal* Issue #2, created the Slackware distribution which is being sold by many vendors. Morse Telecommunication produces Slackware Professional, a Linux package sold in many retail outlets.

ACC Bookstore and Morse had many products and publications about Linux displayed on their table and shelves, including a system demonstrating Abacus Software's MacEmulator for Linux, which had just been announced three days before COMDEX opened.

They also had flyers from other Linux vendors such as WorkGroup Solutions. Mark Bolzern of WorkGroup Solutions was often seen at ACC and Morse's booth as well. WorkGroup Solutions distributes Flagship, which was written by a German company, multisoft GmBH.

ACC Bookstore and Morse Telecommunication's booth. Pictured are Michael Johnston and Patrick Volkerding

Flagship is a compiler that compiles XBase code; it's similar to the MS-DOS product Clipper, but it runs on Linux. This means that you can port MS-DOS XBase applications to Linux.

Walnut Creek was also at Comdex showing their many CD-ROMs. When asked about how well Linux CD-ROMs were selling, one representative said, “Well, COMDEX is like being invaded by the Huns.” I think that meant Linux was not their best-selling product at the convention.

Andrew Grove, President and CEO of Intel Corporation, was one of the keynote speakers. Mr. Grove started with a score card of hits and misses based upon his speech and predictions at COMDEX three years ago. He gave himself hits for the rise of the PCI Local Bus, color notebooks, multimedia, messaging. He gave himself misses for his predictions on collaborative work with multimedia messaging and pen-based and wireless notebooks. (However, he wryly noted, “You can still draw a smiley face on a pen-based notebook.”)

Mr. Grove noted he had underestimated the processor performance vamp—there has been a 13-fold change in three years in terms of cost-effectiveness of processors. He also said that when he last spoke at COMDEX three years before, he had no idea how big the growth of the home personal computer would be, nor how much progress would have been made on the Information Highway.

I wonder what Andrew Grove will say in a few years about the growth of Linux on personal computers.

A customer picking up *Linux Journal*

Archive Index Issue Table of Contents

Advanced search

What Your DOS Manual Doesn't Tell You about Linux

Liam Greenwood

Issue #10, February 1995

A guide to discovering documentation (and other valuables) on your Linux system.

So, you decided to see what all the fuss was about. You installed Linux, logged in to your own Unix machine, and are ready to race. Hmmm, strange prompt:

```
george:~#
```

It's not quite **C:>** but it can't be that hard. You put in a floppy disk and type:

```
george:~# dir a:
ls: a:: No such file or directory
```

The commands and output of commands in this article assume you are using the bash shell. Also, you do not type in the prompt.

“I wonder how you look at floppies?” You type **help** at the prompt and get a screen full of cryptic command templates. Nothing on **dir** though. Ah! There in the second column it is—**dirs**. “No problem,” you think, as you confidently type:

```
george:~# dirs a:
```

...and get **dirs: unknown option: a:**. “Oh, that's right,” you mutter, “Unix doesn't have **a:** and **c:** drives. help again, and let's have a closer look. Here we go, it says: **help [pattern...]** so you enter:

```
george:~# help dir
dirs: dirs [-l]
   Display the list of currently remembered
   directories. Directories find their way
   onto the list with the `pushd' command;
   you can get back up through the list with
   the `popd' command.
   The -l flag specifies that `dirs' should not
   print shorthand versions of directories which
   are relative to your home directory. This
   means that `~/bin' might be displayed as
   /homes/bfox/bin
```

Getting a bit puzzled now you try:

```
george:~# dir
total 1
lrwxrwxrwx 1 root  root 8 Aug 2    21:39  INSTALL -> /var/adm/
lrwxrwxrwx 1 root  root 14 Aug 2   21:39  linux -> /usr/src/linux/
drwx------ 2 root  root 1024 Aug 3 18:05  mail/
```

Phew, something works. What does it all mean though? Different colours, arrows, your login name is there twice on each file and you still don't know how to list what's on a floppy. In fact, you're starting to wonder why you thought this Linux thing was a good idea in the first place.

Help Is on the Way

If Linux is your first foray into the Unix world then you're at a bit of a disadvantage compared to starting out on a commercial system: no manuals. Fortunately, Unix has a long history of carrying its documentation on-line. In fact, the first system I worked on, about eleven years ago, normally shipped with only the vendor-specific manuals in printed form and the majority of the documentation on-line. So here's a few tips on how to discover what you need to know without a set of printed manuals. A whistle-stop tour of **man**, **apropos**, **whatis**, **more** or **less**, **ls** and **find**. Where to look for information on Usenet other than comp.os.linux.help and where to look for information off Usenet, as well as in *Linux Journal*.

First stop is the helpful trinity of **man**, **apropos** and **whatis**. The most essential of these three is **man**. **man** is your access to the on-line manual set. **apropos** is a keywork searching tool for the manual set, and **whatis** is a quick reference to commands. **man** also has the ability to do both the keyword searching of **apropos** and the quick reference of **whatis**.

The best way to start is to try typing:

```
george:~# man man
```

which is asking the on-line manual for the manual page on itself. The header of the ”man“ man page has ”man(1)“ followed by a NAME entry, a SYNOPSIS entry and then the description. The NAME entry gives the name and a short

description of the things explained on this man page. The SYNOPSIS is a template for running the command which shows the required format, the options, and the parameters. The DESCRIPTION entry is the detailed description of the command, its calling conventions, options, parameters, and other details. Near the end of the man page are two very useful sections, the SEE ALSO and where appropriate the FILES section. The SEE ALSO section lists other man pages which are relevant to the one you have just looked up. The FILES section lists files which are relevant to the command you are looking at, for instance, configuration files.

There are three very useful options to the man program. They are **-a**, **-f**, and **-k**. The **-a** option tells the man program to not stop looking for pages at the first one it finds, but to scan all the man page sections and present you with all the relevant pages. You're probably now wondering what those sections are and why they exist. The man pages themselves can help us out with that one.

If you recall, when you typed man man the resulting man page was headed ”man(1)“. This means the page was found in section one of the manual. If you now enter:

```
george:~# man -a man
```

you will be presented with the ”man(1)“ man page again. Now type q to leave the man page viewer. Instead of being put back to the prompt, you will be given another man page to view, which is headed ”man(7)“. This is a description of the way to create man pages and is from section seven. On about the third screen of ”man(7)“ is a list of the various man sections. If you knew that you just wanted to see the page on how to create man pages, you could bypass the ”man(1)“ page by entering:

```
george:~# man 7 man
```

If you remember the name of a command but can't remember what it does, the **whatis** command (or **man -f**) is what you need. These commands search a special ”whatis“ database and return a list of matches with a short description.

```
george:~# whatis man
man (1)         - Format and display the on-line manual pages
man (7)         - Macros to format man pages
man.config (5)  - Configuration data for man
```

In this case, it also got us a pointer to another man-related man page on which we can do a man man.config or man **5 man.config**.

Often you'll know what you want to do, but won't know the command name. To do a keyword search of the ”whatis“ database, you can use either the apropos command or man **-k**. For example, to find out how to copy a file, neither man

copy nor whatis copy are of any help. Using either **apropos copy** or **man -k** copy returns 22 lines of matches, including the **entry cp(1) - Copy files**.

Another valuable way of finding information is to ”cruise the filesystem“. The tools you need are cd, ls, and either more or less. Another valuable little helper is the command **find**.

What do I mean by ”cruise the filesystem“? It just means to look through the filesystem for interesting files, reading any text files, and looking at the man pages for any executable files. You use **ls** to list the files and directories, cd to move into interesting looking directories, and the pager programs more and less to read any text files. So on my system I can **cd /usr** and one of the directories which I see in the **ls** output is called **doc**. When I do an **ls doc**, I see one of the directories is **faq**. I do an **ls doc/faq** and see one of the directories is called **howto**. An **ls** of the howto directory shows me that it's got a collection of Linux HOWTO documents. I **cd /usr/doc/faq/howto** and then I can, for example, **more README** or **less Mail-HOWTO**.

What about executables? Let's take an example a friend of mine had. He was having a look around when an **ls /bin** came up with a program called **dialog**. Thinking it looked interesting, he did a **man dialog** and found it was a program for creating dialog boxes for use in shell scripts. About three days later a *Linux Journal* arrived with an article on **dialog** boxes. I still want to know why my copy of man doesn't automatically cause *Linux Journal* articles to be written.

One important caveat about cruising a filesystem: don't randomly execute programs. Murphy's Law indicates that only data that's important and hard to replace will be lost when you try some innocuous-looking program. Find and read the man pages before trying the program. If there are no man pages, then look for other documentation. Use the **find** command to help you search for more information; it is a powerful and flexible command, allowing you to do searches through a filesystem on a number of file attributes. I'll just give you an example of searching by name. I can look for any filenames with the string readme in them and print the names on my screen with the following construct:

```
george:~# find / -iname `*readme*' -print
```

where **find** is the command, **/** is the place to start the search, **-iname** means do a case-insensitive filename search on the following string (***readme***) and **-print** means to display the result. If I were doing a case- sensitive search, I would replace **-iname** with **-name**.

Why don't I just do all my looking with the **find** command? You build a better roadmap of your system by walking the directories, and you find things you

didn't know to search for. How can you get find to search for the HOWTO documents when you don't know such documents exist?

A warning on the Linux documentation: all the documentation is provided by volunteer efforts and, since many people find writing documentation not as much fun as writing programs, inevitably there will be times when the on-line documentation isn't complete. There may be a program without a man page or other documentation. The documentation may be out of date, or there may be inaccuracies. So be prepared for some inconsistencies. Another thing to be aware of is that the placement of files in a distribution is entirely up to the person creating the distribution. Where they think the files should be, which is how it's documented in the man page, may not be where they actually are in your distribution. The name of the file is unlikely to have been changed, so it's simple enough to use find to resolve such differences.

Programs for which there are no man pages aren't so simple. However, Linux is a type of Unix. There are many books published on Unix and most of those would be relevant to a Linux system. There are books on learning to use Unix, Unix reference books, and Unix system administration books from publishers such as O'Reilly and Associates, The Waite Group, and others. In addition I would strongly recommend one of SSC's Unix command summaries. These summaries are like having your ”whatis" database extended to include both the synopsis and the options from the man page and to have them printed out.

If you have access to Usenet news, then you probably already look at one or more of the Linux discussion groups. In addition to those groups, there are many other Unix discussion groups in the comp.unix hierarchy. There are regular postings to news.answers of many Unix FAQs (Frequently Asked Questions); in particular, there is a Unix books FAQ. There are newsgroups on news readers (news.software.readers), on editors (comp.editors, comp.emacs), on mail (comp.mail hierarchy), and many other topics, so don't limit yourself to just the comp.os.linux groups.

Oh, and if you still can't get that directory listing on your A: drive, here's a hint: man mount.

Archive Index Issue Table of Contents

Advanced search

What's GNU?

Arnold Robbins

Issue #10, February 1995

This month's column discusses RCS, the Revision Control System.

What is RCS? RCS is the Revision Control System. Its job is to manage the process of revising and updating files. It can and should be used for program text and documentation, as well as for any other files that are revised on a frequent basis. RCS allows you to retrieve earlier versions of files, while keeping a log of what changes were made, who made them, and why. RCS makes it easy to compare any two versions of a file, and provides a mechanism for merging changes from two different development “branches” of a source file.

RCS was originally written by Dr. Walter F. Tichy, at Purdue University. Beginning in 1983, it received wide-spread use in the Unix community with its release as part of the User Contributed Software in 4.2BSD. It was described in an article in “Software—Practice And Experience” in July 1985.

Why Use RCS?

RCS provides a safety net for the software developer. When developing, fixing, and improving a program, changes are inevitable. By saving a stable version of your file in RCS, you can later return to a known state if a set of changes does not work out.

If more than one person is working on the same file, RCS allows you to “lock” a file, so that only one person will be allowed to make changes. Other people can still use the file, e.g., for compiling.

Besides keeping track of what changes were made to a file, RCS tracks who made the change and when. RCS also files a log message describing the change. This makes it easy to figure out who broke the program when the fatal bug is finally isolated.

Using RCS

The user interface is intentionally quite simple, consisting primarily of two commands, **ci** and **co**. To start with, make a directory to hold the program and **cd** into it. Then make a directory named **RCS**. Although not required, this is the cleanest way to do it; all RCS files will be kept in the **RCS** subdirectory. We'll also create the first version of the program.

```
$ mkdir hello
$ cd hello
$ mkdir RCS
$ cat > hello.c # editors are for wimps! :-)
#include <stdio.h>
int main(void)
{
   printf("hello, world\n");
}
^D
$ ls -l hello.c
-rw-r--r--  1 arnold       66 Nov  5 22:33 hello.c
```

We now have a C source file that is ready to go. When compiled and run, it prints the well-known, friendly greeting beloved by C programmers the world over.

After making sure it compiles, the first thing to do is “check in” the program with RCS. This is accomplished with **ci**.

```
$ ci hello.c
RCS/hello.c,v <-- hello.c
enter description, terminated with single `.' or end of file:
NOTE: This is NOT the log message!
> world famous C program that prints a friendly
message.
> .
initial revision: 1.1
done
$ ls
RCS
```

The first time a file is checked in, RCS wants a description of just what the file is. It reminds us that this is not the log message, thus, something like “initial revision” would be inappropriate here. The **>** is the prompt for information. Also note that the original file is removed. RCS has the file safely stored in the RCS file **hello.c,v** in the RCS directory.

Checking Files Out

Well, a file that we can't compile isn't of much use, so the next thing to do is get a copy so that we can actually compile the program and use it. This is done with **co**, which stands for “check out”.

```
$ co hello.c
RCS/hello.c,v -> hello.c
revision 1.1
done
$ ls -l hello.c
```

```
-r--r--r-- 1 arnold 66 Nov 5 22:43 hello.c
$ gcc hello.c -O -o hello; ./hello
hello, world
```

Note that the file is returned to us, but with read-only permissions. We are thus allowed to use the file, but not change it. In normal use, for instance, to build a whole source tree to install software, you would check out the files read-only, compile the programs, and then remove the source files.

Locking Files

What about when you want to change a file? Programs do evolve, so how do you get to the next revision? The first thing to do is to check out the file, but with a lock on the file. The lock says that you, and only you, are allowed to check in a new revision of the file. This is necessary if more than one person will be working with the source file, so that two people don't trash each other's work.

```
$ co -l hello.c
RCS/hello.c,v -> hello.c
 revision 1.1 (locked)
done
$ ls -l hello.c
-rw-r--r-- 1 arnold         66 Nov  5 22:51 hello.c
```

This checks out the file, and locks it. Note that the permissions now allow writing to the file. We can edit the file, and make our changes to it.

```
$ sam hello.c   # a nifty editor,
# watch for a future column on it.
$ cat hello.c
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
   if (argc > 1 && strcmp(argv[1], "-advice") == 0) {
      printf("Don't Panic!\n");
      exit(42);
   }
   printf("hello, world\n");
   exit(0);
}
$ gcc -O hello.c -o hello
$ ./hello -advice
Don't Panic!
$ ./hello
hello, world
```

Our program now has a new option, **-advice**, that prints a friendly piece of advice and exits with a well known, special value. The default behavior remains unchanged, except that **exit** is now used for the normal case, as well.

We can now check in the new version to RCS. Assuming that we will want to do further work on the file, **ci** also allows us to use the **-l** option. With this option, **ci** will perform the check-in and automatically do a **co -l** for us, so that we can continue to work on the file.

```
$ ci -l hello.c
RCS/hello.c,v <- hello.c
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single `.' or end of file:
> Added -advice option, and made regular case use exit.
> .
done
$ ls -l hello.c
-rw-r--r-- 1 arnold        208 Nov 5 22:54 hello.c
```

Here we see where the log message is entered. Log messages should be relatively brief, describing what was changed and why. In a commercial environment, you might enter the bug number associated with a particular fix into the log, as well. We also see that the file is still available for further editing (permissions **-rw-r--r--**).

Comparing Versions of a File

You can compare any two versions of a file using the **rcsdiff** command. This command accepts all the options that the regular **diff** command does. However, it usurps the **-r** option for providing revision numbers. By default, rcsdiff compares the current version of the working file with the most recently checked-in version. With one **-r** option, it compares the current file against the specified previous version. You can supply two instances of the **-r** option to make it compare two different revisions, neither of which is the current version. Here's an example of the default (and most common) case:

```
$ rcsdiff -c hello.c
===================================================
RCS file: RCS/hello.c,v
retrieving revision 1.1
diff -c -r1.1 hello.c
*** 1.1 1994/11/06 03:36:45
--- hello.c     1994/11/06 03:54:49
***************
*** 1,6 ****
   #include <stdio.h>
!  int main(void)
   {
        printf("hello, world\n");
--- 1,12 ----
   #include <stdio.h>
+  #include <string.h>
!  int main(int argc, char **argv)
   {
+       if (argc > 1 && strcmp(argv[1], "-advice") == 0) {
+           printf("Don't Panic!\n");
+           exit(42);
+       }
        printf("hello, world\n");
+       exit(0);
   }
```

This generates a “context diff”, showing the surrounding context of what was changed, not just the changes themselves. The lines marked with ! indicate a changed line from the old to the new version, and the lines marked with + indicate lines that were added.

The **rcsdiff** program makes it easy to generate updates that can be applied with **patch**. When a program is finished, simply check in all the files that make it up with a new, higher level revision number, such as 3.0. Then, for the next release, run rcsdiff against revision 3.0 for all files.

```
$ rcsdiff -c -r3.0 RCS/* > myprog-3.0-4.0.patch 2>&1
```

(This doesn't catch the case of brand new files added in 4.0, or deleted files that were in 3.0, but you get the idea.)

As a side note, in order to build and install the RCS software, you need to have the GNU version of **diff**. Linux systems have this already. If you don't have GNU diff, you should get it anyway, since it is very full- featured and noticeably faster than the standard Unix version of **diff**.

Automatically Tracking Interesting Information

It is often useful to be able to look at the contents of a source file and tell what version of the file it is. RCS allows you to do this by performing “keyword substitutions” on the contents of your file when it is checked out. There are a large number of these keywords; the co man page documents them in full. The most common ones are **Id** and **Log**.

The **Id** keyword is replaced with text describing the filename, revision, date and time of checkout, the author, and the state (e.g., **Exp**, for experimental). Usually, this is embedded in a C string constant so that a binary, generated from the file, can be identified with the **ident** command.

The **Log** keyword is replaced with the text of the most recent log message. This is usually placed inside a comment, so that the source file is self-documenting, showing what was changed and when. This is useful, but should be used with caution: if a file is changed frequently, this log can grow quite a lot.

We'll now add the keywords and show the state of the file, both before and after checking in the changed version.

```
$ sam hello.c
$ cat hello.c
#include <stdio.h>
#include <string.h>
static const char rcsid[] = "$Id$";
/*
 * $Log$
 */
int main(int argc, char **argv)
{
   if (argc > 1 && strcmp(argv[1], "-advice") == 0) {
        printf("Don't Panic!\n");
        exit(42);
}
```

```
    printf("hello, world\n");
    exit(0);
}
$ ci -l hello.c
RCS/hello.c,v  <-  hello.c
new revision: 1.3; previous revision: 1.2
enter log message, terminated with single `.' or end of file:
>> add id and log keywords.
>> .
done
$ cat hello.c
#include <stdio.h>
#include <string.h>
static const char rcsid[] = "$Id: hello.c,v 1.3 1994/11/07 03:41:32
arnold Exp arnold $";
/*
 * $Log: hello.c,v $
 * Revision 1.3  1994/11/07  03:41:32  arnold
 * add id and log keywords.
 *
 */
int main(int argc, char **argv)
{
    if (argc > 1 && strcmp(argv[1], "-advice") == 0) {
         printf("Don't Panic!\n");
         exit(42);
}
    printf("hello, world\n");
    exit(0);
}
```

We see that RCS has filled in the information for both keywords. When the program is compiled, the ident command will give us information about all the files used to compile the program that have RCS ids in them.

```
$ gcc -O hello.c -o hello
$ ident hello
hello:
    $Id: hello.c,v 1.3 1994/11/07 03:41:32 arnold Exp arnold $
```

Miscellaneous RCS Commands

You can do just about everything you need to with **ci**, **co**, and **rcsdiff**. There are a few other commands that come with RCS that are also of interest.

The **rcs** command is used for changing the state of RCS files. In particular, it can be used to lock a file that is not locked or to break someone else's lock on an RCS file. This latter operation is perilous and should only be done in an emergency. There are a number of other operations that rcs can perform; see the man page for details.

It is possible to have “branches” off the main line (or “trunk”) of development. For instance, assume that the released version of hello.c is 2.6 and that version 2.7 will be the next released version. Programmer Mary is writing version 2.7, while programmer Joe has to maintain version 2.6. Normally, Joe would start a separate branch off the main development trunk, generating versions 2.6.1.1, 2.6.1.2, and so on. RCS can maintain an arbitrary number of branches off the main trunk, as well as branches off the branches. However, as you might imagine, keeping track of many levels of branching can become confusing.

At some point, Mary will want to make sure that all of Joe's fixes are incorporated into her version of hello.c; she would do this using **rcsmerge**. (**rcsmerge** uses a separate program that also comes with RCS, named merge, which does the actual work of merging the files.)

Finally, the **rlog** command will print out all the log messages for a particular source file. This allows you to see the complete change history of a file.

```
$ rlog hello.c
RCS file: RCS/hello.c,v
Working file: hello.c
head: 1.3
branch:
locks: strict
        arnold: 1.3
access list:
symbolic names:
comment leader: " * "
keyword substitution: kv
total revisions: 3;     selected revisions: 3
description:
world famous C program that prints a friendly message.
----------------------------
revision 1.3    locked by: arnold;
date: 1994/11/07 03:41:32;  author: arnold;  state: Exp;  lines: +6 -0
add id and log keywords.
----------------------------
 revision 1.2
date: 1994/11/07 03:40:21;  author: arnold;  state: Exp;  lines: +7 -1
Added -advice option, and made regular case use exit.
----------------------------
revision 1.1
date: 1994/11/07 03:38:50;  author: arnold;  state: Exp;
Initial revision
 =============================================================
```

Most of the initial stuff that rlog prints out is explained in the RCS man pages. Of interest to us are the description and log message parts of the output, which tell us what the program is, what changes were made, by whom, and when. Interestingly, the timestamps are in UTC, not local time. This is so that developers in different time zones can collaborate without getting discrepancies in their Id strings.

Problems RCS Does Not Solve

The main problem that RCS does not solve is having multiple people working on a file at the same time and the larger issues of release management, i.e., making sure that the release is complete and up to date.

A separate software suite is available for this purpose: **cvs**, the Concurrent Version System. From the README file in the **cvs** distribution:

> **cvs** is a front end to the **rcs**(1) revision control system which extends the notion of revision control from a collection of files in a single directory to a hierarchical collection of directories consisting of revision-controlled files. These directories and files can be combined together to form a software release. cvs

> provides the functions necessary to manage these software releases and to control the concurrent editing of source files among multiple software developers.

You can get **cvs** from ftp.gnu.ai.mit.edu in /pub/gnu. At the time of this writing, the current version is cvs-1.3.tar.gz. By the time you read this, CVS 1.4 may be out, so look for cvs-1.4.tar.gz, and retrieve that version if it is there.

Summary

RCS provides complete, flexible revision control in an easy-to-use package. Like **make**, RCS is a software suite that any serious programmer needs to learn and use.

More Info

Acknowledgements

Thanks to Paul Eggert for reviewing this article. His comments were very useful; several of them were incorporated almost verbatim. Thanks also to Miriam Robbins for forcing me to run **spell**.

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.

Archive Index Issue Table of Contents

Advanced search

Letters to the Editor

Various

Issue #10, February 1995

Readers sound off.

Modem woes

I thoroughly enjoy *Linux Journal*, but (better yet) I thoroughly like Linux! I have come across a problem, however, that I am having difficulty correcting.

My system at home is a Packard Bell Legend 1170. Yes, I'm aware of all the jokes/stories/ etc.,...we'll skip that part! I like my computer and it serves my purposes fine for now.

I have upgraded it to 8mb of RAM, added a PowerGraph video card and most recently installed a new Zoom modem to replace the built-in 2400 that PB uses. During the installation, I discovered that the original built-in modem could not be physically removed, but COM PORT 1 could be disabled. Therefore I disabled COM PORT 1 and set my new modem up for COM PORT 3. The comm program in the DOS partition of my computer works fine with the modem (14.4k beats 2400bd all to pieces!).

The question I have been struggling with is how do I tell my Linux partition to go to COM PORT 3? I've checked with other Linux users and the Linux manual and the common response (other than “replace the computer” or “sue PB”) is that I should create a cua3 in dev. But the problem is there is already a cua0, 1, 2, and 3; so, obviously, the problem is not creating it, but making the connection somehow.

I am sure this is a relatively easy problem that someone more familiar with Linux and more technically proficient than I, can solve easily. I certainly would appreciate any information you may be able to provide me, even if it's a page number and a note to RTM! Great—I'll find it and read it!

Thank you and keep up the good work with the magazine. It is one of the very few that seems to consider those of us who are not computer wizards by publishing articles that are easily read and understood even by a novice such as myself. The series on Andrew was great and I find it does indeed live up to all the claims!

Thanks again.Donald R. Barnhart (barney) don.barnhart@bbs.amaranth.com

LJ Responds:

COM1=cua0, COM2=cua1, COM3=cua2, COM4=cua3, unless Linux is told otherwise. There is a Serial HOWTO at sunsite.unc.edu in /pub/Linux/docs/HOWTO/ that will tell you what you need to know.

I'm not sure what you mean by "tell my Linux partition to go to COM PORT 3". What software are you using the modem with under Linux? Kermit? Seyon? Something else? You probably need to configure the program you are using to connect to the correct port. You may have a file /dev/modem which is linked to the previously correct port, and you may need to remake the link:

```
rm /dev/modem
ln -s /dev/cua2 /dev/modem
```

assuming your modem is correctly installed on cua2.

The other very important thing to consider is the IRQ setting. The modem should not share an IRQ with any other serial line on the computer. The setserial program, which comes with most Linux distributions and can also be retrieved with ftp from tsx-11.mit.edu, is the program to use to make sure that Linux knows what IRQ your modem is on. Again, the Serial HOWTO can help you with this.

Tcl Thanks

I want to thank Matt Welsh for writing the article "X Window System Programming with Tcl and TK" (*LJ* #8). I see a way now to add an X windows interface to my slow port of a DOS style BBS to Linux. I have started the task of moving the simplex BBS software to Linux. Simplex is interesting because it is a complete fidonet BBS package. I mean it can import fido echomail and netmail. I don't know of a BBS for Linux that can do this.

I think that the article was very well written and I enjoyed reading it and fooling around with Tcl/Tk. I can hardly wait for the one explaining how to do this from C.

P.S. A list of Linux user groups would be a nice addition. Thank you for all you have done for Linux and the Linux community!Geoffrey Robert Deasey Geoffrey.Deasey@lambada.oit.unc.edu

LJ Responds:

Having another BBS for Linux will be nice. We have begun publishing a BBS list in this issue (see page 44); BBS sysops are encouraged to send us information about their Linux-specific (or even better, Linux-hosted) BBSs.

As far as Linux User Groups, see page 39 of the November issue and page 46 of the December issue. Our biggest problem in creating a list of Linux User Groups has been that most of the groups have not contacted us to volunteer information.

See page 26 for Matt's latest article: Using Tcl and Tk from your C programs.

Not Again!

Hi! First off, thanks for *Linux Journal*. I have a comment about a particular command in the article “Linux Tips: How to move /home to a new hard drive on your Linux system” in December's issue.

The command:

```
cp -r /home/* /mnt
```

as suggested to copy the home directory does not copy symbolic or hard links correctly. (I am not sure about the other command suggested for I don't have cpio on my system.) If the user has many links, she will find that the “copy” in /mnt takes up more disk space. The following command will copy links correctly:

```
(cd /home ; tar cf - .) | (cd /mnt ; tar xvf -)
```

Cheers,Delman Lee delman@mipg.upenn.edu

LJ Responds:

We had the same problem in an earlier article, and the same (correct) response. I never use **cp -r** myself, and for some reason or other did not catch this mistake. Thanks for the note.

Arithmetic

The control port which corresponds with printer data port 0x378 is 0x37a, and not 0x380, as described in Kernel Korner, December issue (#8). Why is it

required that the example user space printer driver be compiled with optimization turned on? And what level opt?David Morris dwm@shell.portal.com

Michael Responds:

You are absolutely right; I must have been writing too late at night. Fortunately, this mistake was not propagated into the userlp.c code that was included in the article; there, I let the computer do the arithmetic, so that I couldn't accidentally replace hex arithmetic with decimal arithmetic. The reason I was working in decimal was that I had just been working on the shell script version which I mentioned but didn't include in the article, where I had to do all the arithmetic in decimal. Perhaps that is another good reason not to try to write device drivers as shell scripts.

The reason that the userlp.c program included has to be compiled with optimization (of any level at all) turned on is that all the port I/O functions are defined in the header files as “static inline” functions, a GCC extension is enabled only by turning on optimization. I'm sorry that was not adequately addressed in the article.

Archive Index Issue Table of Contents

Advanced search

Documentation?

Phil Hughes

Issue #10, February 1995

When you go shopping you just have to say Linux every time you see Unix.

As we zero in on the second “production” release of the Linux kernel, 1.2, the most common complaint I hear about Linux is its lack of documentation. But, is that really a problem? I think not. Here's why.

Linux Is Very Unix-like

What this means is that if you are looking for a book on basic Linux commands or how to use the VI editor you have a lot of choices. When you go shopping you just have to say Linux every time you see Unix. Sure, there are some differences, but not many from the user's point of view. As all “Unix-like” systems converge on the various standards, such as POSIX and Spec 1170, the differences disappear. Today there are more differences between a BSD-based version of Unix and a System V-based version of Unix than there are between either of these and Linux.

Linux Is Documented by the LDP

There is real Linux-specific documentation. The Linux Documentation Project (LDP), headed by Matt Welsh, is producing high-quality documentation. Right now, Matt's own *Linux Installation and Getting Started* is either included with most CD-ROM distributions or available from the distributor or reseller.

Olaf Kirch's *Linux Network Administrator's Guide* has been published by SSC and an O'Reilly version is also on the way. The excellent coverage of networking in general will probably make this book the first Linux-specific book to become commonly used by non-Linux (read that as Unix) users as well.

Michael K. Johnson, our editor, is now in the process of updating his *Linux Kernel Hacker's Guide (KHG)*. You will see some of this information in his new

monthly *Kernel Korner* column. And, once it is up to date, SSC plans to publish it.

There are more LDP books on the way. Expect to see a System Administrator's Guide, Programmer's Guide, User's Guide and man pages. And, of course, all of these books are available on the archive CD-ROMs and ftp sites.

New Linux Documentation Appears Every Month

Besides the LDP efforts, publishers are getting on the Linux bandwagon. Springer-Verlag has published a book titled Linux: Unleashing the *Workstation in Your PC*, available in both German and English. O'Reilly is working on a Linux book, and rumors indicate that Benjamin-Cummings and SAMS are working on Linux books. 1995 will probably be the year of Linux documentation.

Why All the Complaints?

Many people who are complaining just downloaded Linux off an ftp site or bought an archive CD. In either case, there is lots of documentation available but it doesn't print itself. You need to remember that if you bought SCO Unix you probably dug into your pocket for $1,000 or more as opposed to less than $100 for Linux. Either take some money and buy the books you need or take some time and print out what is on the CD.

The other problem, of course, is getting you to read the documentation. Many CD vendors are now including a small book with the distribution to get you started. People seem less intimidated by a 20-page insert in the CD case than by a 1,000-page manual. Just remember, you may still need that 1,000 pages of documentation to find all the answers.

Is There Still a Problem?

Yes, there is. Linux is changing so rapidly that it is hard for the documentation to keep up. For example, if you buy a printed copy of the Linux HOWTOs that has been sitting on a bookstore shelf for a few months, most of it is already out of date. Documentation that will be outdated by the time it is printed is not what publishers (or booksellers) like.

But this is one of the reasons the LDP separated the HOWTOs from the LDP books. These books evolved with much feedback from the early users and, as each one becomes stable, they are becoming available in printed form.

Many of the distributors include HOWTOs on their distribution CDs along with browser tools to search, read and print them. The initial focus has been browsers under MS-Windows. (As much as I dislike this dependence on

Microsoft, some people are going to need to be able to read the documentation in order to get Linux up and running.)

Matt Welsh, coordinator of the LDP, has recently made the HOWTOs available on sunsite.unc.edu in HTML format (as well as other formats). This means that if you have the HTML format documents in an uncompressed form on your system (some archive CDs have this) you can use a WWW browser such as Lynx or Mosaic to access these documents in hypertext form. And, as the lead time on pressing and distributing a CD is shorter than that for a printed book, the consumer can get the best of both worlds—stable documentation for their bookshelf and up-to-the-minute documentation on the CD.

In conclusion, the Linux community and the publishing community have heard you and are taking action. There is documentation and there will be more. But remember your obligation: support these publishing efforts including the LDP, get the documentation and, last but not least, RTFM.

Phil Hughes is the publisher of *Linux Journal*.

Archive Index Issue Table of Contents

Advanced search

New Products

LJ Staff

Issue #10, February 1995

HaL Announces Ishmail for Linux, MetaCard, Version 1.4 and more.

HaL Announces Ishmail for Linux

HaL Software Systems has announced the availability of Ishmail (Information SuperHighway Mail), a multi-media electronic mail tool for Unix systems. Ishmail features a Motif graphical user interface and extensive support for MIME (the Internet standard for multi-media mail). Extensive on-line help, including a user's guide which can be viewed on-line on the World Wide Web (www.hal.com/products/sw), combined with an industry-standard user interface, make this product very easy to learn and to use. Ishmail is available on a variety of Unix platforms, including Linux, SunOS, Solaris, AIX, HP-UX, DEC OSF/1 and Novell UnixWare. Compatibility with the most common Unix mail-folder formats ensures easy transition for new users and coexistence with other e-mail programs.

In addition to on-line documentation, Ishmail utilizes the Internet for distribution and technical support. The product can be down-loaded from HaL's ftp server (ftp.halsoft.com) with a 30-day free, no-obligation evaluation license. If the product is purchased, a permanent license is delivered by e-mail. A single-user license is $99. Multi-user discounts, educational discounts and site licenses are also available. For more information, contact Tom Lang, Hal Software Systems, 3006A Longhorn Blvd., Austin, TX 78758; phone (512) 834-9962 or (800) 762-0253; e-mail to info@halsoft.com.

MetaCard 1.4 Released

MetaCard Corporation has released MetaCard, Version 1.4, a multimedia development environment capable of playing QuickTime, AVI, FLI and FLC format movies and importing HyperCard 2.2 stacks. It is available in an embedded version as well as a normal development environment. The Linux

version is available at a special price of $195; the normal price is $495 for a single-user.

For more information, contact Scott Raney, MetaCard Corporation, 4710 Shoup Pl., Boulder, CO 80303; phone (303) 447-3936; e-mail to raney@metacard.com.

Archive Index Issue Table of Contents

Advanced search

Block Device Drivers: Interrupts

Michael K. Johnson

Issue #10, February 1995

Last month, we gave an introduction to block device drivers. This month, we look at some tricks that are useful when writing block device drivers, starting with the most basic “trick” of using hardware interrupts where available and describing some neat infrastructure that block device drivers can take advantage of by adding five lines of code and one function.

Block devices, which are usually intended to hold file-systems, may or may not be interrupt-driven. Interrupt-driven block device drivers have the potential to be faster and more efficient than non- interrupt-driven block device drivers.

Last month, I gave an example of a very simplistic block device driver that reads its request queue one item at a time, satisfying each request in turn, until the request queue is emptied, and then returning. Some block device drivers in the standard kernel are like this. The ramdisk driver is the obvious example; it does very little more than the simplistic block device driver I presented. Less obvious to the casual observer, few of the CD-ROM drivers (actually none of them, as I write this) are interrupt-driven. It is easy to determine which drivers are interrupt-driven by reading drivers/block/blk.h, searching for the string **DEVICE_INTR**, and noting which devices use it.

I'm tired of typing “block device driver”, and you are probably tired of reading it. For the rest of this article, I will use “driver” to mean “block device driver”, except where stated otherwise.

Efficiency Is Speed

Interrupt-driven drivers have the potential to be more efficient than non-interrupt-driven ones because the drivers have to spend less time busy-waiting —sitting in a tight loop, waiting for the device to become ready or finish executing a command. They also have the potential to be faster, because it may

be possible to arrange for multiple requests to be satisfied at once, or to take advantage of peculiarities of the hardware.

In particular, the SCSI disk driver tries to send the SCSI disk one command to read multiple sectors and satisfy each of the requests as the data for each block arrives from the disk. This is a big win considering the way the SCSI interface is designed; because initiating a SCSI transfer takes some complex negotiation, it takes a significant amount of time to negotiate a SCSI transfer, and when the SCSI driver can ask for multiple blocks at the same time, it only has to negotiate the transfer once, instead of once for each block.

This complex negotiation makes SCSI a robust bus that is useful for many things besides disk drives. It also makes it necessary to pay attention to timing when writing the driver, in order to take advantage of the possibilities without being extremely slow. Before certain optimizations were added to the generic, high-level SCSI driver in Linux, SCSI performance did not at all approach its theoretical peak. Those optimizations made for throughput 3 to 10 times greater on most devices.

As another example, the original floppy driver in Linux was very slow. Each time it wanted a block, it read it in from the media. The floppy hardware is very slow and has high latency (it rotates slowly and if you wanted to read the block that just started going past the head, you had to wait until the disk made a full revolution), which kept it very slow.

Around version .12, Lawrence Foard added a track buffer. Since it only takes approximately 30% to 50% more time to read an entire track off the floppy as it does to wait for the block you want to read to come around and be read (depending on the type of disk and the position of the disk at the start of the request), it makes sense, when reading a block, to read the entire track the block is in.

As soon as the requested block has been read into the track buffer, it is copied into the request buffer, the process waiting for it to be read can continue, and the rest of the track is read into a private buffer area belonging to the floppy driver. The next request for a block from that floppy is often for the very next block, and that block is now in the track buffer and ready immediately to be used to fulfill the request. This is true approximately 8 times out of 9 (assuming 9 blocks, or 18 sectors, per track). This single change turned the floppy driver from a very slow driver into a very fast driver.

Alright! Enough Already!

So, you are convinced that interrupt-driven drivers have a lot more potential, and you want to know how to turn the non-interrupt-driven driver you wrote

last month into an interrupt-driven one. I can't give you all the information you need in a single article, but I can get you started, and after reading the rest of this article, you will be better prepared to read the source code for real drivers, which is the best preparation for writing your own driver.

The basic control flow of a request for a block from a non-interrupt-driven driver usually runs something like this **simplification alert**:

user program calls **read() read()** (in the kernel) asks the buffer cache to get and fill in the block buffer cache notices that it doesn't have the data in the cache buffer cache asks driver to fill in a block with correct data driver satisfies request and returns buffer cache passes newly-filled-in block back to **read() read()** copies the data into the user program and returns user program continues An interrupt-driven driver runs more like this **simplification alert**: user program calls **read() read()** (in the kernel) asks the buffer cache to get and fill in the block buffer cache notices that it doesn't have the data in the cache buffer cache asks driver to fill in a block with correct data driver starts the process of satisfying the request and returns buffer cache waits for block to be read by sleeping on an event Some other processes run for a while, perhaps causing other I/O on the device. the physical device has the data available and interrupts the driver driver reads the data from the device and wakes up the buffer cache buffer cache passes the newly-filled-in block back to **read()**. **read()** copies the data into the user program and returns user program continues

Note that **read()** is not the only way to initiate I/O.

One thing to note about this is that just about anything can be done before waking up the process(es) waiting for the request to complete. In fact, other requests might be added to the queue. This seems, at first, like a troublesome complication, but really is one of the important things that makes it possible to do some worthwhile optimizations. This will become obvious as we start to optimize the driver. We will start, though, by taking our non-interrupt-driven driver and making it use interrupts.

Interrupts

I am going to take the *foo* driver I started developing last month, and add interrupt service to it. It is hard to write good, detailed code for a hypothetical and vaguely defined device, so (as usual) if you want to understand better after reading this, take a look at some real devices. I suggest the hd and floppy devices; start from the **do_hd_request()** and **do_fd_request()** routines and follow the logic through.

```
static void do_foo_request(void) {
  if (foo_busy)
    /* another request is being processed;
```

```
        this one will automatically follow */
    return;
    foo_busy = 1;
    foo_initialize_io();
}
static void foo_initialize_io(void) {
  if (CURRENT->cmd == READ) {
    SET_INTR(foo_read_intr);
  } else {
    SET_INTR(foo_write_intr);
  }
    /* send hardware command to start io
       based on request; just a request to
       read if read and preparing data for
       entire write; write takes more code */
}
static void foo_read_intr(void) {
  int error=0;
  CLEAR_INTR;
    /* read data from device and put in
       CURRENT->buffer; set error=1 if error
       This is actually most of the function... */
    /* successful if no error */
    end_request(error?0:1);
    if (!CURRENT)
      /* allow new requests to be processed */
      foo_busy = 0;
    /* INIT_REQUEST will return if no requests */
    INIT_REQUEST;
    /* Now prepare to do I/O on next request */
    foo_initialize_io();
}
static void foo_write_intr(void) {
        int error=0;
  CLEAR_INTR;
  /* data has been written. error=1 if error */
  /* successful if no error */
  end_request(error?0:1);
  if (!CURRENT)
    /* allow new requests to be processed */ foo_busy = 0;
/* INIT_REQUEST will return if no requests */
  INIT_REQUEST;
  /* Now prepare to do I/O on next request */
  foo_initialize_io();
}
```

In blk.h, we need to add a few lines to the FOO_MAJOR section:

```
#elif (MAJOR_NR == FOO_MAJOR)
#define DEVICE_NAME "foobar"
#define DEVICE_REQUEST do_foo_request
#define DEVICE_INTR do_foo
#define DEVICE_NR(device) (MINOR(device) > 6)
#define DEVICE_ON(device)
#define DEVICE_OFF(device)
#endif
```

Note that many functions are missing from this; this is the important part to understanding interrupt-driven device drivers; the “heart”, if you will. Also note that, obviously, I haven't tried to compile or run this hypothetical driver. I may have made some mistakes—you are bound to make mistakes of your own while writing your driver, and finding bugs in this skeleton will be good practice for finding bugs in your own driver, if you are so inclined. I do suggest that when you write your own driver, you start with code from some other working driver rather than starting from this skeleton code.

Structure

An explanation of some of the new ideas here is in order. The first new idea is (obviously, I hope) the use of interrupt routines to do part of servicing the hardware and walking down the request list. I used separate routines for reading and writing; this isn't fundamentally necessary, but it does generally help allow cleaner code and smaller, easier-to-read interrupt service routines. Most (all?) of the interrupt-driven device drivers of any kind in the real kernel use separate routines for reading and writing.

We also have a separate routine to do most of the I/O setup instead of doing it in the request() procedure. This is so that the interrupt routines can call the separate routings to set up the next request, if necessary, upon completion of a request. Again, this is a design feature that makes most real-world drivers smaller and easier to write and debug.

Context

It must be noted that any routine that is called from an interrupt is different than all the other routines I have described so far. Routines called from an interrupt do not execute in the context of any calling user-level program and cannot write to user-level memory. They can only write to kernel memory. If they absolutely need to allocate memory, they must do so with the **GFP_ATOMIC** priority. In general, it is best for them to write into buffers allocated from user-process-context routines with priority **GFP_KERNEL** before the interrupt routines are set up. They also can wake up processes sleeping on an event, as **end_request()** does, but they cannot sleep themselves.

Macros

The header file blk.h provides some nice macros which are used here. I won't document them all (most are documented in *The Linux Kernel Hackers' Guide*, the *KHG*), but I will mention the ones I use, which are used to manage interrupts.

Instead of setting up interrupts manually, it is easier and better to use the **SET_INTR()** macro. (If you want to know how to set them up manually, read the definitions of **SET_INTR** in blk.h.) Easier because you just do **SET_INTR(interrupt_handling_function)**, and better because if you set up automatic timeouts (which we will cover later), **SET_INTR()** automatically sets them up.

Then, when the interrupt has been serviced, the interrupt service routine (**foo_read_intr()** or **foo_write_intr()** above) clears the interrupt, so that spurious interrupts don't get delivered to a procedure that thinks that it is supposed to

read or write to the current request. It is possible—it only takes a little more work—to provide an interrupt routing to handle spurious interrupts. If you are interested, read the **hd** driver.

Automatic Timeouts

In blk.h, a mechanism for timing out when hardware doesn't respond is provided. If the *foo* device has not responded to a request after 5 seconds have passed, there is very clearly something wrong. We will update blk.h again:

```
#elif (MAJOR_NR == FOO_MAJOR)
#define DEVICE_NAME "foobar"
#define DEVICE_REQUEST do_foo_request
#define DEVICE_INTR do_foo
#define DEVICE_TIMEOUT FOO_TIMER
#define TIMEOUT_VALUE 500
/* 500 == 5 seconds */
#define DEVICE_NR(device) (MINOR(device) > 6)
#define DEVICE_ON(device)
#define DEVICE_OFF(device)
#endif
```

This is where using **SET_INTR()** and **CLEAR_INTR** becomes helpful. Simply by defining **DEVICE_TIMEOUT**, **SET_INTR** is changed to automatically set a "watchdog timer" that goes off if the foo device has not responded after 5 seconds, **SET_TIMER** is provided to set the watchdog timer manually, and a **CLEAR_TIMER** macro is provided to turn off the watchdog timer. The only three other things that need to be done are to:

1. Add a timer, **FOO_TIMER**, to **linux/timer.h**. This must be a **#define**'d value that is not already used and must be less than 32 (there are only 32 static timers).
2. In the **foo_init()** function called at boot time to detect and initialize the hardware, a line must be added:
   ```
   timer_table[FOO_TIMER].fn = foo_times_out;
   ```
3. And (as you may have guessed from step 2) a function **foo_times_out()** must be written to try restarting requests, or otherwise handling the time out condition.

The **foo_times_out()** function should probably reset the device, try to restart the request if appropriate, and should use the **CURRENT->errors** variable to keep track of how many errors have occurred on that request. It should also check to see if too many errors have occurred, and if so, call **end_request(0)** and go on to the next request.

Exactly what steps are required depend on how the hardware device behaves, but both the hd and the floppy drivers provide this functionality, and by comparing and contrasting them, you should be able to determine how to write

such a function for your device. Here is a sample, loosely based on the **hd_times_out()** function in hd.c:

```
static void hd_times_out(void)
{
   unsigned int dev;
   SET_INTR(NULL);
   if (!CURRENT)
      /* completely spurious interrupt-
         pretend it didn't happen. */
      return;
   dev = DEVICE_NR(CURRENT->dev);
#ifdef DEBUG
   printk("foo%c: timeout\n", dev+'a');
#endif
   if (++CURRENT->errors >= FOO_MAX_ERRORS) {
#ifdef DEBUG
      printk("foo%c: too many errors\n", dev+'a');
#endif
      /* Tell buffer cache: couldn't fulfill request */
      end_request(0);
      INIT_REQUEST;
   }
   /* Now try the request again */
   foo_initialize_io();
}
```

SET_INTR(NULL) keeps this function from being called recursively. The next two lines ignore interrupts that occur when no requests have been issued. Then we check for excessive errors, and if there have been too many errors on this request, we abort it and go on to the next request, if any; if there are no requests, we return. (Remember that the **INIT_REQUEST** macro causes a return if there are no requests left.)

At the end, we are either retrying the current request or have given up and gone on to the next request, and in either case, we need to re-start the request.

We could reset the foo device right before calling **foo_initialize_io()**, if the device maintains some state and needs a reset. Again, this depends on the details of the device for which you are writing the driver.

Stay Tuned...

Next month, we will discuss optimizing block device drivers.

Other Resources

Michael K. Johnson is the editor of *Linux Journal*, and is also the author of the Linux Kernel Hackers' Guide (the KHG). He is using this column to develop and expand on the KHG.

Archive Index Issue Table of Contents

Advanced search