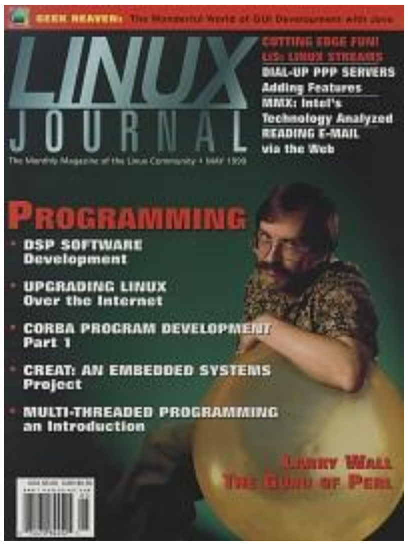
Advanced search

Linux Journal Issue #61/May 1999



Focus: Programming

Marjorie Richardson

Issue #61, May 1999

This month, in order to support the quest for applications, applications and more applications, we feature programming tutorials and tools.

Last year, Linux use in the business community jumped by 10%. For this sort of growth to continue, more programs need to be written which directly target this community and the consumer in general. Computer stores are stocking Linux distributions. To promote the sale of these distributions to their customers, applications of interest to the consumer must be available. Accounting programs for the small as well as the large business, financial programs for the individual, educational tools, games and more games. These are the types of programs people want and are therefore the types that must be supplied.

Many projects to bring this type of application to Linux exist. Pick your favorite (http://www.linuxresources.com/apps/projects.html) and help out, or start one that is missing. Get active.

The place to start for programming information is, of course, *Linux Journal*. This month, in order to support the quest for applications, applications and more applications, we feature programming tutorials and tools. Programming has become an annual focus for *Linux Journal* because of its popularity with our readers. Our writers like it too, sending us more articles dealing with programming issues than any other topic.

Last November, we interviewed Guido Van Rossum, creator of the Python scripting language; last month, we interviewed John Ousterhout, the wizard of Tcl/Tk; this month, we talk to Larry Wall, the guru of Perl. To hone your programming skills, you can study the complete programming cycle, learn about POSIX threads, write your own GUI using Java and learn all about that architecture called CORBA. To learn a bit more about memory management, take a look on-line at a review of three memory checkers and a description of the buffer-overflow hack and how to avoid it (see “Strictly On-line” in the Table of Contents and our web site at http://www.linuxjournal.com/issue61/).

Marjorie Richardson, Editor in Chief

Archive Index Issue Table of Contents

Advanced search

Larry Wall, the Guru of Perl

Marjorie Richardson

Issue #61, May 1999

Discover a bit about Perl's creator and what's happening with Perl.

I “talked” to Larry Wall, the creator of the Perl scripting language, by e-mail on March 1. Larry proved to be quite voluble, and I think you'll find this interview fun as well as informative. I certainly did.

Marjorie: Back in the beginning, what inspired you to write Perl?

Larry: That depends on what you mean by “beginning”. Like Moses said: “In the beginning, God created the heavens and the earth.” I'm not being entirely facetious about that. Whichever way you care (or don't care) to interpret scripture, I think the universe is a pretty hefty inspiration for anyone who aspires to be a creator. I've certainly tried to put a universe of ideas into Perl, with some amount of success.

In terms of biographical beginnings, my father was a pastor, as were both my grandfathers, and many of my ancestors before that. My wife likes to say that preachers are bred for intelligence (though I suppose she might be saying that just to flatter me). Be that as it may, I did receive a fairly decent set of brain construction genes. Beyond that, I also received a rich heritage of ideas and skills, some of which found their way into Perl culture. For instance, the notion that you can change the world. The idea that other people are important. The love of communication and an understanding of rhetoric, not to mention linguistics. The appreciation of the importance of text. The desire to relate everything to everything else. The passion to build up rather than tear down. And, of course, the dead certainty that true wealth is measured not by what you accumulate, but by what you pass on to others.

The beginnings of Perl were directly inspired by running into a problem I couldn't solve with the tools I had. Or rather, that I couldn't *easily* solve. As the Apostle Paul so succinctly put it, “All things are possible, but not all things are expedient.” I could have solved my problem with awk and shell eventually, but I possess a fortuitous surplus of the three chief virtues of a programmer: Laziness, Impatience and Hubris. I was too lazy to do it in awk because it would have been hard to get awk to jump through the hoops I was wanting it to jump through. I was too impatient to wait for awk to finish because it was so slow. And finally, I had the hubris to think I could do better.

Of course, actually writing something like Perl takes a great deal of hard work, patience and even humility. Had I just been doing it for myself, I probably wouldn't have made the effort. However, I was aware from the beginning that other people were going to be using Perl, so I've always integrated the “laziness curve” over the whole community, not just over myself. I was being vicariously lazy. So here we are talking about vicars again.

Marjorie: Well, that certainly answered the question fully. I must admit I didn't expect you to go back as far as the beginning of the Universe. :-) How'd you come up with that name?

Larry: I wanted a short name with positive connotations. (I would never name a language “Scheme” or “Python”, for instance.) I actually looked at every three- and four-letter word in the dictionary and rejected them all. I briefly toyed with the idea of naming it after my wife, Gloria, but that promised to be confusing on the domestic front. Eventually I came up with the name “pearl”, with the gloss Practical Extraction *and* Report Language. The “a” was still in the name when I made that one up. But I heard rumors of some obscure graphics language named “pearl”, so I shortened it to “perl”. (The “a” had already disappeared by the time I gave Perl its alternate gloss, Pathologically Eclectic Rubbish Lister.)

Another interesting tidbit is that the name “perl” wasn't capitalized at first. UNIX was still very much a lower-case-only OS at the time. In fact, I think you could call it an anti-upper-case OS. It's a bit like the folks who start posting on the Net and affect not to capitalize anything. Eventually, most of them come back to the point where they realize occasional capitalization is useful for efficient communication. In Perl's case, we realized about the time of Perl 4 that it was useful to distinguish between “perl” the program and “Perl” the language. If you find a first edition of the Camel Book, you'll see that the title was *Programming perl*, with a small “p”. Nowadays, the title is *Programming Perl*.

Marjorie: Okay, is Perl perfect now or do you continue to do further development?

Larry: Hmm, the two are not mutually exclusive. Look at Linux. :-)

Actually, Perl was never designed to be perfect. It was designed to evolve, to become more adaptive, as they say. There is no such thing as a perfect organism, biologically speaking. About the most you can say is an organism is more or less suited for the environment in which it finds itself. In fact, biologists are just now realizing that any organism which seems to be “perfect” for one environment is likely to be in danger of extinction as soon as the environment changes. Over-specialization is only as good as your ecological niche. We're not just talking about dinosaurs here, but also snail darters and cheetahs and a bazillion beetles in Brazil—not to mention Visual Basic.

We've already seen the deaths of many over-specialized organisms in computing: Lisp machines, Ada chips and many so-called fourth generation languages. Any program ever written in assembly language for an obsolete architecture is now obsolete. Likewise, any program that ties its fortunes to a single operating system is likely to go down with the ship. I don't know how many more torpedoes Windows can take before it sinks, but if and when it does, a whole batch of specialized programs are going down with it. Obviously, for reasons relating to the open source movement, Linux doesn't have this particular problem.

Anyway, back to Perl. Right from the start, Perl was designed for change. This involves certain tradeoffs, some of which appear to be suboptimal to people who don't think the way I think. For instance, I wanted to be able to add new keywords to Perl without breaking old programs, so I put them into a separate namespace from variable names. This meant either variable names or keywords had to be marked somehow as special. I chose to mark variables, since it also made it easy to interpolate variables into strings, and since there's a history of marking variables in computer languages such as BASIC. Note this was actually non-adaptive in certain environments, namely in the minds of certain purists who think the added punctuation makes Perl ugly, and too much like BASIC. Well, maybe it does. So what? That was a conscious tradeoff so that

Perl would be more useful in the long run. In that respect, Perl is less adaptive in the specific ecological niche comprising the minds of computer scientists, but more adaptive in the world as a whole. I've never regretted that particular tradeoff.

Of course, once you get past first impressions, there are many things in Perl that computer scientists do like, such as lexically scoped variables and closures. So by and large, those computer scientists who can hold their nose long enough to get the cheese into their mouths find the taste bearable.

More importantly, Perl 5 introduced an extension system that, much like Linux's module system, allows continued development of the language without actually changing the core language. That is, you can pull in a Perl module that warps the language to your purposes in a controlled fashion. If a module becomes popular enough, we can consider making it part of the core of Perl—maybe.

That's not to say we never change the core anymore. We recently added support for multi-threading and for Unicode. Interestingly, even when we do make changes to the core these days, we make it look as though the programmer is pulling in an extension module. Essentially, if you use a fancy new core feature that warps the semantics in some way, you have to declare it. This is how we maintain almost complete compatibility with older Perl scripts. Most Perl 1 scripts still run unchanged under Perl 5. As a side benefit, feature declarations are right up front where the dependency is visible at compile time, so we rarely die in the middle of execution for lack of a feature. Compare this with shell programming, where you don't even know whether all the programs you're intending to invoke actually exist until you try to run them, and then, kablooey!

Marjorie: What are your future development plans for Perl?

Larry: If I could predict that, I'd be a smarter person than I am. I'm just smart enough to know I'm no smarter than that, which is why I designed Perl to evolve in the first place.

That being said, I can tell you some of the characteristics I look for in a project.

First, if it has anything to do with text processing, Perl is a natural. Perl has never stopped being a text-processing language, though it long ago escaped the straitjacket of being *just* a text processing language. That's one reason Perl was a natural for CGI programming, because Perl excels at ripping text apart and putting it back together.

Second, I look for projects that involve gluing things together. We don't use glue on Legos—we glue together things that weren't designed to go together. As a glue language, Perl has thin characteristics so that it can flow into tiny gaps, and thick characteristics so that it can fill in larger voids. Perl is always at home in the interstices. The typical CGI script or mod_perl servlet glues a database together with the Web. When that particular interstice disappears, there will be other interstices.

Third, I look for projects that franchise the disenfranchised. We joke about sending our leftovers to the starving people in Africa, but there are, in fact, billions of potential programmers outside of America who can't afford to lay out hundreds of bucks for an operating system or an application. China recently put in a single order for 200,000 Internet books from a publisher I know (and work for). That's just the beginning. This is why I hacked Unicode support into Perl last year. Of course, text processing has something to do with Unicode too.

Having said all that, it almost doesn't matter what I look for in my next development project, because I don't do most of the Perl development these days. The Perl community outweighs me by many orders of magnitude, and they're really the ones who are making Perl the be-all and end-all of scripting languages. I just sit on the sidelines and cheer occasionally. I'm cheering now. Rah, rah, rah! :-)

Marjorie: In what way is Perl better than other scripting languages such as Python and Eiffel?

Larry: Perl is unique, not just among scripting languages, but among computer languages in general. It's the only computer language consciously and explicitly designed to be postmodern. All other computer languages are still stuck in the modern era to some degree. Now, as it happens, I don't normally use the term “postmodern” to describe Perl, because most people don't really understand postmodernism, even as they embrace it. But the fact is that American culture has become thoroughly postmodern, not just in music and literature, but also in fashion, architecture and in overall multicultural awareness.

Modernism was based on a kind of arrogance, a set of monocultural blinders that elevated originality above all else, and led designers to believe that if they thought of something cool, it must be considered universally cool. That is, if something's worth doing, it's worth driving into the ground to the exclusion of all other approaches. Look at the use of parentheses in Lisp or the use of white space as syntax in Python. Or the mandatory use of objects in many languages, including Java. All of these are ways of taking freedom away from the end user “for their own good”. They're just versions of Orwell's Newspeak, in which it's impossible to think bad thoughts. We escaped from the fashion police in the 1970s, but many programmers are still slaves of the cyber police.

In contrast, postmodernism allows for cultural and personal context in the interpretation of any work of art. How you dress is your business. It's the origin of the Perl slogan: “There's More Than One Way To Do It!” The reason Perl gives you more than one way to do anything is this: I truly believe computer programmers want to be creative, and they may have many different reasons for wanting to write code a particular way. What you choose to optimize for is your concern, not mine. I just supply the paint—you paint the picture.

Marjorie: Who is using Perl and how are they using it?

Larry: A couple of years ago, I ran into someone at a trade show who was representing the NSA (National Security Agency). He mentioned to someone else in passing that he'd written a filter program in Perl, so without telling him who I was, I asked him if I could tell people that the NSA uses Perl. His response was, “Doesn't everyone?” So now I don't tell people the NSA uses Perl. I merely tell people the NSA *thinks* everyone uses Perl. They should know, after all.

As an interesting side note, it turned out this fellow was the very administrator who shut down the NSA project Perl was (indirectly) written to support. He was vaguely amused when I pointed out Perl might well be the most enduring legacy of the project.

As to what everyone uses Perl for, it's really all over the map. I was astounded several years ago to be told how heavily Perl is used on Wall Street. “A Perl book on every other desk” is how I heard it. But it makes sense when you realize that market analysts need to revise their models continually, and they need to scan news services for information that might be related to their positions in the market. Rapid prototyping and text processing are what they need.

Many people associate Perl with CGI scripts, though of course most of the heavy lifting is done with mod_perl servlets under Apache. Perl is used just as much on the client side in the robots and spiders that navigate the Web and build much of the linkage implicit in various on-line databases. And that's not all. If you've ever been spammed (and who hasn't?), your e-mail address was almost certainly gleaned from the Net using a Perl script. The spam itself was likely sent via a Perl script. One could say that Perl is the language of choice for Net abuse. And one could almost be proud of it.

That's only scratching the surface of what Perl is used for. Without getting Mr. Gallup or the U.S. Census Bureau involved, the best way to figure out what Perl is used for is to look at the 800 or so reusable extension modules in the Comprehensive Perl Archive Network (the CPAN, for short). If you glance through those modules, you'll get the impression that Perl has interfaces to almost everything in the world. With a little thought, you may figure out the reason Perl has interfaces to everything is not so much so Perl itself can talk to everything, but so Perl can get everything in the world talking to everything else in the world. The combinatorics are staggering. The very first issue of *The Perl Journal* (not to be confused with *Linux Journal*) contained an article entitled "How Perl Saved the Human Genome Project". It explains how all the different genome sequencing laboratories used different databases with different formats, and how Perl was used to massage the data into a cohesive whole.

Marjorie: We received a product announcement for PerlDirect from ActiveState Tool Corporation that says:

> "PerlDirect provides reliability, stability, support and accountability for Perl through the following features: validated, quality-assured releases of Perl and its popular extensions; advice and support; Y2K test suite; and a Perl Alert weekly bulletin. PerlDirect offers an opportunity to provide direct input to a leading organization involved in open-source development. Basic annual subscription rates start at $12,000 US."

Are you affiliated with this company? I think it's interesting they are offering to let a subscriber have direct input to open-source development for $12,000 a year. Does this make sense?

Larry: Sounds like a pretty ordinary support contract to me. I don't think even Richard Stallman would disagree with the notion that support is a valid way to make money off free software.

I'm not directly affiliated with ActiveState, but I've worked with them, and I think the problems they've solved far outweigh any problems they've created. You've got to understand their market has always been the Windows space, where you're actually doing people a favor by charging them money for things, because that's the only way to keep from confusing them. Linux users are smarter than this, of course, but some Linux users aren't quite smart enough to realize Windows is a different culture, and Perl, being a postmodern language that is sensitive to context, will look different in a different culture.

Marjorie: Oops, didn't mean to sound as if I thought they weren't on the up-and-up—just curious if you knew them. What are your views of the Open Source movement? Do you think it will become a true phenomenon, or is it just a passing fancy?

Larry: I must have a conjunctive rather than a disjunctive brain, because I think both of those notions are true. And I also think they're both false. :-)

How can we claim open source is becoming a true phenomenon when it has already been a true phenomenon for a couple of decades now? We're merely pointing out to everyone a practice that has a proven track record of producing excellent code. On the other hand, we're certainly trying to make it a *truer* phenomenon, in the sense that we hope more people will feel it's a valid development model for many kinds of software that were formerly developed under a closed model.

And, of course, it's a passing fancy—just as we've had other passing fancies for free-form syntax, structured programming, and more recently, object orientation.

What you have to understand is that, from the viewpoint of the passing fancy, people represent a kind of fork in the road. It's like separating the sheep from the goats in the book of Revelation. Some of these fancies pass on the one side and go into oblivion, while others pass on the other side and go into common practice, usually after a period of excessive enthusiasm. Free-form syntax, structured programming and object orientation are all good (in moderation). But note that all of these passing fancies had a history of being useful before they became popular. The passing fancies that go into oblivion are the ones rooted not in history, but rather in someone's wishful thinking (usually someone from marketing). By this criterion, open source will probably go into common practice because it's already in common practice.

The way I see it, the open source movement is just another manifestation of the growing postmodernism of our culture. By contrast, the notion of trade secrecy is just a rehash of the modernistic idea of originality at any cost. We've had a lot of lip service given to code reuse over the years, but it really only works with open source. A postmodern computer programmer truly believes in reusing good code whether it's original or not. It's not a point of pride anymore. A good postmodern is *supposed* to plagiarize the things he or she thinks are cool.

Marjorie: If everything becomes open source, how will programmers make a living?

Larry: Contrary to many open source advocates, I don't see everything becoming open source. What I do see is a growing recognition that anything resembling large-scale infrastructure ought to be open source, much as the United States has recognized that interstate highways should not be toll roads. On the other hand, we don't expect city parking lots to be free except in certain enlightened municipalities. So I'd expect to see Windows become open source before Word does.

That being said, there are many ways to make a living off open source, just as there are many ways to make a living off open *science*. But here's precisely where I think the open source movement has some growing to do. Open science basically started out as a hobby of the rich, but it didn't truly blossom into the form we recognize today until its patronage was taken over by educational institutions. This has not quite happened yet with open source. Or rather, it started to happen, but then many educational institutions got caught up in the drive for the almighty dollar. I wish more places would follow the example of UC Berkeley.

Marjorie: On that note, how do you make a living?

Larry: To start off, I worked programming and sys admin jobs like anyone else and did my free software on the side. Later, I wrote a book and started collecting royalties. The book became a best-seller, so it made my publisher, O'Reilly & Associates, even more money than it made me. Of course, they have to pay more people with that money, so it evens out.

Anyway, three years ago it occurred to Tim O'Reilly and me that anything good for Perl was also good for O'Reilly & Associates, so now they pay me to do whatever I like, as long as it helps Perl. It's been a good symbiosis.

Marjorie: Any interesting projects you'd like to tell us about?

Larry: I'm supposed to be working on the third edition of the Camel Book, so I don't officially have any other interesting projects at the moment. Of course, I *have* been playing around with that Palm Pilot Tim O'Reilly gave me for Christmas, but I won't tell if you won't.

Marjorie: Agreed. Give us some personal info—where you went to school, interests, etc.

Larry: I spent the first half my childhood in south Los Angeles about two miles from where the Watts riots broke out, and the second half of my childhood in Bremerton, Washington, where no riots broke out, but I did graduate from high school. For the third half of my childhood, I went to Seattle Pacific University, where I started off majoring in chemistry and music, later switched to premed, and eventually (after taking several years off to work in the SPU computer center) ended up majoring in Natural and Artificial Languages (a self-designed major). After that, my wife and I attended grad school in linguistics at Berkeley and UCLA. At the time, we were actually planning to be missionaries (more specifically, Bible translators), but we had to drop that idea for health reasons. Funny thing is, now the missionaries probably get more good out of Perl than they'd have gotten out of me as a missionary. Go figure.

As for my interests, that's hard, because I tend to be interested in anything that's interesting. Which comes out to pretty much anything except opera and soap opera—space opera's okay, though.

Marjorie: What do you do for fun?

Larry: Read and listen to my wife read to me (especially space opera). Discuss anything and everything with anyone and everyone in my family. Work *NY Times* crossword puzzles. Play mah jong. Practice aikido. Watch anime. (Maybe Japanese soap opera is okay.) Play with my fish. Rescue my fish from the equipment I bought to keep them alive.

Marjorie: Looks like you stay busy and have fun—a good combination. What do you eat for breakfast?

Larry: I eat all kinds of things for breakfast—but then, I generally eat breakfast at lunchtime.

Marjorie: Seems as good a time as any. Thanks for giving us so much of your time. It's been interesting.

LINUX JOURNAL

CORBA Program Development, Part 1

J. Mark Shacklette

Jeff Illian

Issue #61, May 1999

The authors provide some basics to get the new CORBA programmer started.

CORBA (Common Object Request Broker Architecture) is one of those acronyms for which most people have some “feel”, others have some interest, but very few have any real experience. This is the first article in a series of three in which we will attempt to increase the first, augment the second and remedy the third. We will be quick to point out CORBA's strengths, but will not shrink from disclosing some of CORBA's current shortcomings which, while not terribly numerous, nevertheless can present a stumbling block to the uninitiated.

Our goal is to help the Linux programmer new to CORBA get his feet wet through examination of a very simple CORBA application. Our simple distributed application is developed using OmniORB, a free ORB from Oracle-Olivetti Research in Cambridge, England. OmniORB is a fast, clean implementation of the basic CORBA 2.0 architecture. We have calculated its performance to be anywhere from 2 to 15 times faster measuring the same code as its commercial competitors—plus, it runs on Linux. It implements the CORBA standard's naming service, although it does not currently support some of the more popular CORBA services, such as the event service, the trader service or the life cycle service. (It has a life cycle service of sorts, but is nonstandard.) It also does not yet offer a dynamic invocation interface, a dynamic skeleton interface or an Interface Repository. It does, however, provide a fine thread abstraction for POSIX pThreads, one that is highly portable (one of the authors has ported it to HP's tenuous DCE threads implementation). The OMNIThread abstraction supports Linuxthreads 0.5 and above (POSIX 1003.1c-draft 10 and comes with Red Hat's glibc implementation) and MIT pThreads and is well worth the download. The simple example shown here does not require two separate networked machines, so even if you have only one machine, you can still get your feet wet. There is no requirement in CORBA that the objects actually be physically distributed. That design feature is completely optional.

We know that CORBA is an approach to distributed programming, but what exactly is it? CORBA defines a coordinated specification for the implementation of distributed object communication. CORBA is a specification managed by the Object Management Group (http://www.omg.org/), a consortium of over 800 different hardware and software vendors, whose diverse interests intersect around the concept of distributed computing. The OMG's goal with CORBA was to define a communication standard that would be platform-and-language independent, with a focus on the object-oriented paradigm.

The primary motivation for distributed programming is parallel processing, or actually distributing the activity of an application across several computers so that they are simultaneously working to solve a problem or a complex of problems. Resources are generally limited in any environment, and for applications that are stranded on one machine, the resource pool (memory, disk, I/O, etc.) can easily become strained. When that happens, overall throughput and performance of the application suffers. Distributed programming allows a developer to distribute the workload of an application across several different machines, each with its own set of resources. By doing so, the developer can greatly influence the overall throughput of an application in a very positive way.

CORBA provides developers with a framework in which to develop objects which can communicate with one another on a single machine or over a network, regardless of the hardware platform or the programming language. Using CORBA, it is possible for an object written in C++ on UNIX to communicate with another object written in Java on Windows or an object written in COBOL on a mainframe. Several technologies attempt to provide similar functionality: TCP/IP, Berkeley Sockets, DCOM, Remote Procedure Calls (ONC and DCE), Java's Remote Method Invocation, et al. System V IPC offers several inter-process communication capabilities such as shared memory and message queues, but they are single-machine solutions by default, whereas CORBA is centered around exposing objects on a network. CORBA, the implementations of which are often built on top of some of these components such as TCP/IP, sockets and DCE, offers a unique package of benefits over and above these alternatives.

CORBA allows distribution to take place almost entirely within the object model, abstracting the details of the object communication so that the developer has to worry about only the higher-level interfaces as opposed to the nuts and bolts of the communication layers. There is a cost in terms of network latency that any CORBA developer must be conscious of early in the game. In a sense, the resource limitations of a single computer are traded for the network limitations in bandwidth and performance. Remember, each call to a remote object is a network call. If your design calls for 100 set methods to be called on a remote object just to initialize it (a very bad CORBA design), you will soon see performance degradation in a whole new light.

CORBA objects can be implemented in many different programming languages running on many different platforms. In order for a specification to define a framework for such an environment, it is important to have a platform- and language-independent method of describing objects. In order to meet this need, CORBA defines a mapping language called IDL (Interface Definition Language) that is actually very similar to C++ in syntax. IDL is used to describe the way a remote object appears to the outside world, along with properties or methods that exist in the object. An IDL compiler is then used to translate the IDL into the source code of a particular implementation language, e.g., C++, Java, C, Ada, Smalltalk and COBOL. For the server, the IDL compiler creates the source code needed by the ORB to expose the object to the outside world and creates a skeleton that is then "filled in" with the actual implementation of the object by the developer. For the client, the IDL compiler creates stubs which allow the remote object to appear to be local to the client. In order to remain platform-and-language independent, IDL has its own variable types. The IDL compiler maps each of these variable types to a representative language construct in the native language for the client or server.

Now we can address the question of how CORBA actually works. The first and most important component to look at is the ORB. In the CORBA specification, the ORB (Object Request Broker) is the communication layer that resides between a CORBA object and the user of the CORBA object. Through the ORB, a client application can access properties, pass parameters, invoke methods and return results from a remote object. It is a common misconception that the ORB is a daemon or a service that implements CORBA—some ethereal middleman that floats around out on the network. Actually, the ORB is a communication layer that resides partially in the client and partially in the server at the same time. The ORB is responsible for intercepting a call to a remote object, locating the remote implementation of the object and facilitating communication with the remote object. Thus, when we talk about "the ORB", we are talking about the communication capabilities provided to a client and a server through their respective stubs and skeletons, as well as through calls those stubs and skeletons make to the ORB implementation's runtime libraries, which provide low-level communication and marshaling capabilities.

Given that an ORB is a communication layer and is responsible for locating an implementation, it needs some method by which to find the remote object.

CORBA achieves this by assigning all remote objects a unique IOR (Interoperable Object Reference). The IOR is like a telephone number by which the client application can call upon a specific remote object. In order for the client application to access the remote object server, it must first be able to obtain an IOR. There are several different methods by which a client can obtain an IOR, the easiest and least practical being to pass it on the command line. Many CORBA implementations (such as VisiBroker and Orbix) have simple proprietary IOR lookup mechanisms that allow the IOR to be passed to the client using a “bind” call. The OMG defines the Naming Service as the preferred way for a client to obtain an IOR of a remote object. In part three of this series, we will go into the Naming Service in detail. For our first example, however, we will pass the IOR of the server's object to the client in a common file that will be read by the client during initialization.

We deliberately created a simple example so that the code can be easily followed, and we provided a Makefile and Make.rules because omniORB uses a rather complicated **make** scenario that is difficult to follow. This sample code was developed and tested using omniORB 2.5.0 running on Red Hat Linux 5.1 (kernel 2.0.35). The code was compiled using g++ version egcs-2.90.27 980315 (the egcs-1.0.2 default C++ compiler with Red Hat). (We also compiled and ran the code using the latest omniORB 2.7.0 and egcs 1.1.1.)

To build and run this example, download omniORB 2.5.0 from http://www.orl.co.uk/omniORB/omniORB.html. Fill out the form, and download the correct version of omniORB for your version of Linux—it is free. You might want to get down the binary version that comes with complete source code as well. The binary version expects you to be using the g++ that comes with gcc 2.7.2. If you're running egcs (for example, Red Hat 5.x), you will need to go ahead and recompile omniORB, so it will work with the egcs g++ compiler (choose **i586_linux_2.0_egcs** in **config.mk** in the config directory). You can find out what g++ compiler you are running by typing **g++ -v**. Follow the instructions in the README* files for instructions on building and setting up the omniORB environment.

Once you have omniORB installed and built, you can download the sample code from ftp://ftp.linuxjournal.com/pub/lj/listings/issue61/3201.tgz. Then unpack the tar file. In order to build the sample, you must edit both the Make.rules and Makefile files. See the file README.build for information on what to change for your location and compiler. Once you've edited the appropriate files for your location, simply type **make** to build the samples.

Once you have built the example, run it by launching the server in one window (or virtual terminal) by typing **server**. Notice that the IOR for the server's object is printed to STDOUT. It is also written to a file called ior.out. Next, run the client in another window by typing **client**. This opens the IOR file, obtains the IOR for the server's object, then resolves that IOR to an object reference and makes a call on the remote object. For a remote connection, you will need to get the ior.out file from the server's directory to the directory the client is going to run in. You can do this by using FTP to transfer the file, after the server is up and running, to the client's directory on the other machine.

Like most CORBA applications, our simple CORBA example is made up of three items. First, a server application that instantiates a CORBA object, then basically blocks forever, thus exposing the CORBA object to potential clients. Second, the implementation of the CORBA object itself, which is run when a client obtains a reference to the server's object. Third, a client that binds to the CORBA object and proceeds to make calls against its interface as defined in the IDL for the CORBA object.

Let's begin with the IDL. Our example has a very simple interface defined, PushString. **Listing 1** shows that interface PushString has a single function, called pushStr, which takes a single input parameter of IDL type string and returns an IDL boolean value. When this is compiled by the IDL compiler (**omniidl2**), the following files are created:

- PushString.hh: stub/skeleton header
- PushStringSK.cc: stub/skeleton code

In the omniORB implementation (the creation of stubs and skeletons is ORB-specific), both the stubs and the skeletons are defined in the same file for each IDL file processed. The skeleton for the implementation is called _sk_PushString and it is inherited by the implementation of PushString as follows (in PushString_i.h):

```
class PushString_i : public _sk_PushString
```

Every function defined in an interface is declared as a pure virtual in the skeleton class (in PushString.hh):

```
virtual CORBA::Boolean pushStr ( const char * inStr ) = 0;
```

When the implementation of PushString (PushString_i) states that it is inheriting from the skeleton (public _sk_PushString), it pledges to implement each single pure virtual function in the skeleton class. In this way, the pledge of an implementation fully supporting an interface is enforced. It is a compiler error to fail to implement one of the pure virtual functions declared in the skeleton.

In **Listing 2** (Srv_Main.C), we see the server application that creates the CORBA object and presents it to potential clients via the ORB. The first thing the program does is open an output file called ior.out. This is where it is going to write the IOR for the object it is about to create. Then, it initializes the ORB:

```
CORBA::ORB_ptr orb =
CORBA::ORB_init(argc,argv,"omniORB2");
```

This call takes in parameters to the orb which were passed in as command-line arguments, such as flags to turn on tracing, set the name of the server, etc., and uniquely identifies this initialization as expecting the omniORB2 ORB.

After the ORB has been initialized, it is the BOA's (Basic Object Adapter) turn. The BOA for the server is initialized with the following call:

```
CORBA::BOA_ptr BOA =
orb->BOA_init(argc,argv,"omniORB2_BOA");
```

The BOA initialization is ORB-dependent. In omniORB, the name of the BOA is set to “omniORB2_BOA” and the user can specify certain flags to the BOA. A communication layer must be able to communicate with something, and one of the alternatives developed in the CORBA specification is the BOA. The BOA resides in a CORBA server and is responsible for initializing the remote object when a client requests access. The BOA then provides a translation layer between the remote representation of the object to which the ORB communicates and the actual physical implementation of the object.

After the BOA has been initialized, the implementation of a CORBA interface is created, in our case, the PushString interface. Once the implementation has been created, we register the newly created implementation with the BOA object by calling the object's **_obj_is_ready** function with the object reference of the BOA itself, which was returned by the **BOA_init** call. The main purpose for registering object implementations with the BOA is to let it know the implementation is running so it can dispatch calls to the object made by clients.

Finally, we call **impl_is_ready** on the BOA. We do this to let the BOA know it should *now* begin to listen for client requests on its designated port. Prior to this call, although the implementation is ready and waiting, no traffic will arrive because the BOA is not listening for it. It is the impl_is_ready call that tells the BOA to start listening for client connections on behalf of this object's implementation. Depending on the ORB implementation, the client may block on a function call to the remote object, or an exception may be thrown if impl_is_ready has not been called.

In **Listing 3** (PushString_i.h), we see the implementation's header file which declares that this implementation will be implementing the **pushStr** function, but the class must also define its own constructor and destructor. The IDL compiler does not create the implementation header for you (some compilers, such as Orbix's **idl2cpp** compiler, will create a “shell” implementation header and cpp file if you request it); generally you have to do that on your own. Notice the class declaration line:

```
   class PushString_i : public _sk_PushString
```

We are declaring that PushString_i will be implementing all the pure virtual functions defined in _sk_PushString (we have defined only one), by inheriting from the skeleton.

In **Listing 4** (PushString_i.C), we actually go about the task of defining the implementation of the interface defined in the PushString.idl file. We will, of course, implement our constructor and destructor, but we will also give a real body to our virtual pushStr function. We do this with the definition of the function:

```
   CORBA::Boolean PushString_i::pushStr(
         const char * inStr)
      throw(CORBA::SystemException)
   {
      int retval;
      cerr << "in PushStr\n";
      char * m_str = new char[strlen(inStr)+1];
      strcpy(m_str,inStr);
      // just for fun, mess with the boolean return
      if( strlen(m_str) > 5 )
         retval = 1;
      else
         retval = 0;
      cout << "The string pushed was "
           << m_str << endl;
      delete [] m_str;
      cerr << "Implementation leaving PushStr..."
           << endl;
      return(retval);
   }
```

Here, when the client calls the pushStr function, passing it a string (notice the IDL string type has been mapped in C++ to a **const char ***), the function prints out a message letting us know we're in the implementation. It then copies the incoming string into a local buffer, checks to see if the length of the incoming string is greater than 5, and if so, sets the function's Boolean return value to 1; otherwise, it sets the return value to 0. Then, pushStr prints out the string that was copied, deletes it, and notifies us we are now leaving the implementation. At that point, we return to the client the Boolean **retval** created earlier.

In **Listing 5** (Client.C), we see the client code. The first thing we do when we enter the Client.C code is open the ior.out file the server created, which contains the IOR of the server's object implementation. The client expects this file to be in its current working directory. Once that file has been opened and the IOR stored in the variable “IOR”, the client code begins to look reminiscent of the code in the Srv_Main.C file. Namely, the same calls to initialize the ORB and the BOA take place, with the same parameters.

Then, we create a variable of type PushString_var. The type PushString_var is essentially a helper type that provides the stub capabilities for marshaling and unmarshaling parameters. It also provides other essential functions such as releasing and duplicating object references, along with functions for determining whether a particular object reference is empty (null) or not. Essentially, the PushString_var type stands as a proxy within the Client's process space for the real implementation of the PushString object, which may be miles away across the network.

After pushStringVar is created, we enter a try/catch block that calls the ORB and asks it to translate the string IOR (which we obtained from the ior.out file created by the server) into an actual object reference. The object reference created here, however, is of type **CORBA::Object_var**, a generic type. In order to actually make a call against that object's interface, we must “downcast” it into the actual type of object it represents and for which we have an implementation. This is done through a call to a function named **narrow**, defined in the Abstract Base Class for the stub PushString (defined in PushStringSK.cc). Once the generic type has been resolved to an actual interface implementation, we can then make calls on that interface. This is done with the call:

```
   pushStringVar->pushStr(src);
```

This makes a call to the remote object's implementation, passing in a string that contains simply the phrase “Hello World”. At that point, given no exceptions were thrown during the try block, the client notifies us it has completed the call without an exception and terminates.

Finally, you might be wondering whether it is always necessary to pass IORs around via files. The answer is certainly not, but because omniORB does not have its own proprietary bind mechanism (which is how a simple example like this would be implemented in VisiBroker or Orbix), the only way to get the client talking to the server object without using the Naming Service is through an IOR passed in to the client by the server. In a future article on CORBA services, we will show how the CORBA naming service can be used to obtain an object reference through nothing more than a name (which looks much like an absolute path name in UNIX). With the naming service in place, we will no longer need to pass IORs from the server to the client.

In our next article, we will be talking about CORBA on Linux using VisiBroker for Java, implementing in Java. We will also have a much more complicated example in our article on CORBA services, where we will offer an example that utilizes the Naming Service, as well as a factory which creates objects on behalf of the client.

Resources

Mark Shacklette is a principal with Pierce, Leverett & McIntyre in Chicago, specializing in distributed object technologies. He holds a degree from Harvard University and is currently finishing a Ph.D. in the Committee on Social Thought at the University of Chicago. He is an adjunct professor teaching UNIX at Oakton Community College. He lives in Des Plaines, Illinois with his wife, two sons and one cat. He can be reached at jmshackl@plm-inc.com.

Jeff Illian is a principal with Energy Mark, Inc. in Chicago, specializing in electric utility deregulation and distributed trading technologies. He holds a degree from Carnegie-Mellon University in Operations Research (Applied Mathematics). He lives in Cary, Illinois with his wife, son and daughter. He can be reached at jeff.illian@energymark.com.

Archive Index Issue Table of Contents

Advanced search

GUI Development with Java

Ian Darwin

Issue #61, May 1999

Mr. Darwin takes a look at Java and describes the steps for writing a user interface in Java.

If you looked at the earliest versions of Java and concluded that its GUI development toolkit wasn't quite ready for prime time, it's time to look again.

The Java Foundation Classes (JFC) introduced with Java Version 1.2 bring Java forward to the point where it can easily compete head-on with Motif and MFC for professional GUI development. If you already know the Java language, JFC can beat both Motif and MFC hands down for ease of programming. In this article, I will show code that was developed “by hand” using just *vi* and the Java Development Kit (JDK). Many higher-level development tools and GUI builders are available to make this job even easier.

What are Java and AWT?

Portability is one of the holy grails of system designers. The UCSD Pascal System of 1980 compiled into portable P-Code that could be interpreted on most of the microcomputer systems common in its day. The C language and the UNIX operating system Linux is based upon both became popular because they could run on a variety of platforms. The latest newcomer is Java. Java programs compile from source code into “byte code”, a portable and compact machine representation of the executable statements the programmer wrote.

Java Continues from C and C++

C and C++ are well-known languages in the developer community. To help developers come up to speed quickly and easily, Java borrows most of the syntax of C and quite a bit of the syntax of C++. All the basic syntax operators such as +, -, *=, (), {} and others work. For C programmers who wish to understand Java's OO syntax, think of objects as `struct`s with functions associated with them. Note that all methods (functions) are part of objects, so the syntax

```
objectName.function(args);
```

is normally used. One of the most notable differences from C is the lack of pointers, **malloc** and **free**.

Pointers were necessary in the days when we needed to access words in particular locations in memory, but have led to a lot of unreadable and hard-to-maintain code. The functions *malloc* and *free* provide C with a low-level paradigm for allocating and freeing memory. Java does away with both; since Java programs are compiled for the Java Virtual Machine, in which the addresses are unknown at compile time, there are no pointers. This has the beneficial side effect of ruling out viruses based on jumping into the BIOS or system disk-formatting routines; no syntax is present in the language or in the underlying virtual machine for referring to a particular location in real memory.

As in C++, allocation of memory is handled by the operator **new**, which is similar to *malloc* but can be used only to allocate objects and arrays. Freeing of memory, however, is automatic; no free or delete is available, and memory is reclaimed by a “garbage collector” routine at runtime. Sounds like it gives you less control, but if you write C, you probably don't bemoan the fact that local variables are allocated and freed when you enter and leave a function. Java simply extends this notion to arrays and objects, which makes for more reliable programs.

```
int foo() {
int i;   i is allocated "someplace" in
         memory (or register)
         do something with i
}        i is automatically reclaimed
```

Java is an object-oriented language in the traditions of C++ and Smalltalk. Java eliminates a few C++ operators, but experienced C++ programmers have little trouble upgrading their skills to the new language.

Instead of C++'s multiple inheritance, Java provides “interfaces” with multiple inheritance of specification but not implementation. This may be more powerful than C++, since it allows multiple views on an object; that is, a class can implement several interfaces and can be passed by any of those type names, exposing only the methods known to objects implementing that interface. This can provide greater type safety than traditional C++ multiple inheritance.

A Complete Environment

If it's a complete programming environment you want, Java has it. Instead of using the native C library, for example, Java programs use classes and methods in **java.lang**, the package of classes for Java language features. One example is String—normal quoted strings like “Hello, world” compile to String objects, and the String class has methods such as **compareTo**, **equals**, **substring**, **startsWith**/**endsWith**, etc. No more worry about **bcmp** versus **memcmp**; Java provides a single set of methods that works everywhere. AWT, which is the subject of most of this article, provides a portable Graphical User Interface layer. Java.util is a package of utility routines such as random numbers, collection classes and others.

In today's global village, it's important that software be able to function in any locale. Internationalization is basic to “Java, the programming language for the Internet”. Strings and characters are therefore 16-bit Unicode rather than 8-bit ASCII, which is not too surprising (see http://www.unicode.org/). What may surprise you is that Java identifiers can be written in Unicode characters, so that programmers in any language can write identifiers which make sense to them (assuming they have a way of typing the characters).

If it is database access you want, Java provides the Java Database Connectivity (JDBC) to access relational databases. It's patterned loosely on Microsoft ODBC, but operates at a somewhat higher level. There is even a bridge to ODBC, so you can access an ODBC database even if a Java driver is not yet available for it. (See “Database Connectivity Using Java” by Manu Konchady, *LJ* November 1998.)

To meet the needs of rapid application development, JavaBeans supports composition of programs out of reusable components in GUI builders. Not just the GUI's arrangement but the entire application can, in many cases, be “written” by visually indicating the relationship between events such as a button press and software components such as spreadsheets, graphing Beans or HTML viewers. Since the ActiveX market didn't grow as expected, many ActiveX developers are converting their components into portable JavaBeans. And since Beans and applications based on them can run on any UNIX, Linux or *BSD system, this can be only good news for Linux developers.

Open and Free Technology?

But isn't Java proprietary? Well, although Sun invented Java and many of the pieces of technology that accompany it, it can be called an “open” technology. Old-timers will remember how Sun dominated the UNIX distributed file system by making its NFS a public specification, even giving away the source for the RPC and XDR layers that underlie NFS. From the beginning, the specification of the Java language and the specification and format of the compiled class files have been publicly available and the source code of the public API has been included with the freely-downloadable JDK. The source for everything—compiler, runtime interpreter and the internal parts of the API—has been available for free under a non-disclosure agreement that permits free redistribution of binaries. Without this, there would most likely not be a Java for Linux, SunOS4.1 or *BSD. In fact, there are several: JDK ports, Kaffe ports and others. Further, the licensing is designed to encourage the "open API" concept. Read this extract from Clause 2 of the JDK license, which every Java developer who uses Sun's JDK or any derivative must agree to:

> In the event that Licensee creates any Java-related API and distributes such API to others for applet or application development, Licensee must promptly publish an accurate specification for such API for free use by all developers of Java-based software.

Because of this, members of the free software community have responded as enthusiastically to Java as they did to Linux. Several free compilers, at least one free interpreter and many free libraries are available. Even commercial companies are making some libraries free. A good place to explore Java freeware (and payware) is Gamelan (pronounced Gamma-LOHN), at http://www.developer.com/. My own contributions include Jabadex, a Rolodex-like application, a set of X Color names (Java's AWT has only 13 named colors) and others (see http://www.darwinsys.com/freeware/).

Most recently, Sun announced easier licensing—the Sun Community Source License—presumably patterned after the Mozilla license. It's not the BSD Copyright or the GPL, but it's a step closer. See http:java.sun.com/ and look for licensing.

AWT—A Windowing Toolkit

Java's developers wanted everything about Java to be portable, including how to deal with the X Window System, MS Windows, Macintosh and other window systems. The Abstract Window Toolkit is the solution they provide. Instead of starting by writing components that work everywhere, they wrote a library of GUI components that is a least common denominator to what the big three systems offer. It used the underlying native components on each platform, so that it would "look and feel" like a native application. This is an important aspect of user acceptance—if users have to learn a whole new GUI just to use your software, they probably won't bother. This approach limited what you could do with the early versions of Java's AWT, but with more recent versions, this is no longer true. The JFC components bring Java to the forefront of fully functional GUI development environments.

JFC = AWT + 2-D + Swing + Accessibility

Java has been increasing in popularity since its first public release in 1995. Version 1.0 incorporated the Applet API, a basic window toolkit (AWT) and numerous other APIs. Version 1.1, released in the fall of 1996, added a tremendous amount of new functionality including internationalization, better coupling between GUI controls and their action-handling code, text formatting and hundreds of new classes. JDK1.2 has just been released at the time of this writing (January 1999). Version 1.2 of the JDK, which is also being called “Java 2”, includes the Java Foundation Classes (JFC). JFC includes the Swing GUI classes which are the focus of this article, some accessibility features to make computing usage easier for persons with various disadvantages, the 2-D graphics package and the original AWT.

The 2-D graphics can be thought of as PostScript for Java. If you've done any PostScript, you'll know it is really two things: a scripting language and a marking engine. Since Java already provides a powerful programming language, the 2-D developers needed to provide only the “marking engine” (putting marks on paper), transforms, composites and other fancy graphics and a much-expanded set of fonts. However, it is implemented in a backward-compatible way: a Graphics2D object is subclassed from a Graphics object. This provides Java developers and users with all the fancy graphics capabilities invented over a decade of desktop publishing, all in a platform-independent way.

We've Got Swing!

The Swing Set portion of JFC is named after the musical style which revolutionized popular music in the 1940's with such greats as Duke Ellington (Hint: a penguin-like creature named Duke is Java's mascot and logo). Swing has many features, including:

- Customizable look and feel
- More choice items, ComboBoxes, etc.
- More layout managers, panels with borders, etc.
- Tabbed folders
- Table View widget
- Tree View widget
- Tooltips
- Easy to make all standard types of Dialogs with one call
- Color, Font, and other chooser dialogs

Figures 1 and 2 are UNIX screen shots of the color chooser (with tool tips) and the TreeView program; the source for these and all other examples shown here is on the FTP site.

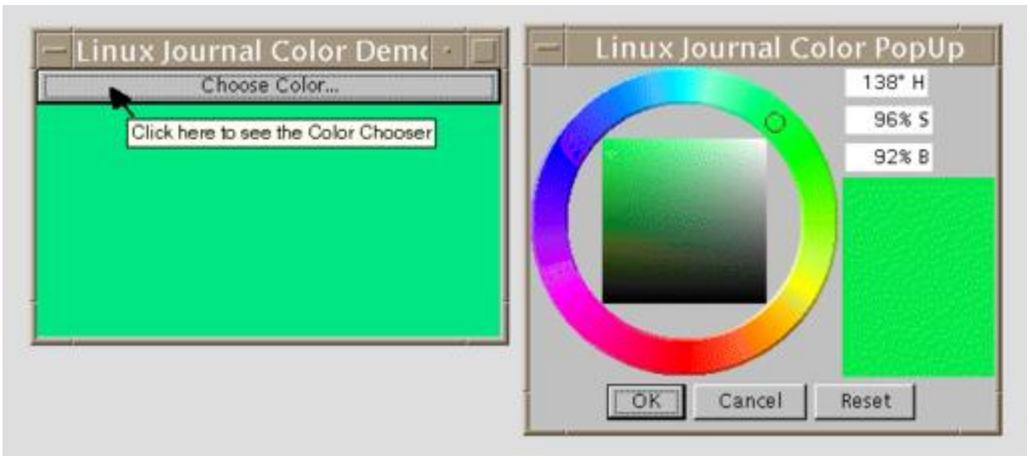


Figure 1. Color Chooser

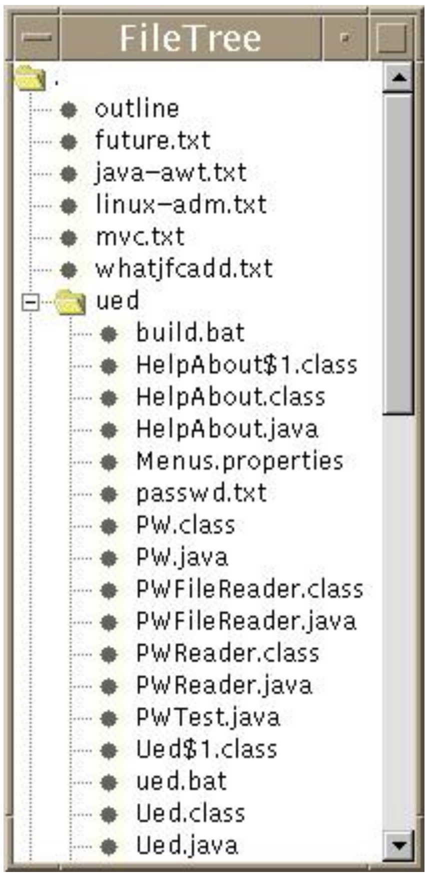


Figure 2. TreeView Program

Java 1.1's look and feel was that of the underlying operating system. On Motif, its menus and buttons look like Motif widgets; on MS Windows, they look like Microsoft widgets; on the Macintosh, like Macintosh widgets. They truly are the native platform's widgets. Using a Java interface called “Peers”, a 1.1 program constructs and uses native toolkit components, but the developer never has to think about it. You simply write in terms of AWT components.

After 1.1 had been in use for a while, somebody at JavaSoft decided to poll the developers. Apparently 50% were happy with the status quo, another 30% favored a platform-independent look and feel, while the remaining 20% wanted to be able to provide their own corporate look and feel, to be the same across all platforms. The Swing set UIFactory was the result. All Swing components have a settable look and feel. The look and feel is provided by a combined View-Controller object generated by a class called UIFactory. UIFactory is powerful—it can make up and apply new UI objects on the fly, resulting in programs that can change their look and feel on demand. Our demo program (see below) has a button that lets you choose between several "look-and-feel" styles. The Motif emulation is included in the JDK. The "Metal" style is a crisper look developed by JavaSoft. Implementations of the proprietary look-and-feel of MS Windows 95 and the Macintosh UI are available, but only on those platforms (for licensing, not technical, reasons).

When you switch, the entire UI is repainted in the new look without losing any of the choices you have made so far. It is quite an impressive operation. Other look-and-feel classes such as an OPEN LOOK or NextStep are also possible, even probable. The only downside I have seen to Swing is its performance. Don't expect Swing applications to be quite as snappy as native C/C++ Motif/ MS-Windows/Macintosh applications, particularly in startup time. Once the application is up, though, Swing applications run acceptably fast on modern computers with adequate memory.

Simple Applications

Java can be used to create many kinds of programs. One of the first to garner widespread attention was Web Applets, which dynamically extend the behaviour of the web browser by being embedded in a web page. Java can also be used to make Web Servlets, background TCP or UDP servers, or ordinary GUI-based desktop applications. The latter are easiest to demonstrate, so we'll use them for our example. Most of what we say here applies to the GUI part of an Applet as well.

The simplest application is probably a window with a quit button that exits when you run it. The simplest form of the program is shown in **Listing 1.**

In this listing, the class ButtonDemo1 is both the "model" (data handling code) and the "action listener", the code that responds to user events such as pushed buttons. The class "extends JFrame" so that it can be a top-level window. Also, it "implements ActionListener" so that it can provide the **actionPerformed** method called by the button when it is pressed.

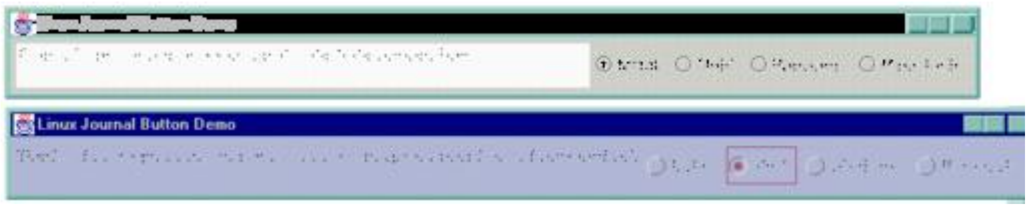


Figure 3. Button

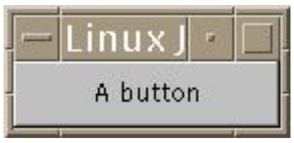


Figure 4. Demo Button

Having a GUI layout be its own action listener works adequately for toy applications. The actionPerformed method has to figure out somehow which button was pressed. It's not hard here, but doesn't scale well: as the code gets larger and larger, it becomes difficult to manage all the interactions among the GUI components. Thus, it is preferable for each component to have its own action listener. One way of doing this is to use Java's “inner classes”: write one class inside another. The inner class is syntactically analogous to nested procedures in languages like Pascal. In Listing 2, I have simply added a second button and recast the code so that each button has its own listener.

Listing 2. Demo Button with Inner Class ActionListener

Now we can write a third version (not shown, but available in the archive file on the FTP site) that uses CheckButtons instead of JButtons. This version will not quit, but will change the GUI among those listed.

The action listener for each button just calls the UIManager class' **setLookAndFeel** method with the correct full class name and a utility class' **updateComponentTreeUI** passing in the top-level window (the JFrame subclass). This changes all components in the tree to display in the newly selected look and feel. Since some components may need a different size, we again call the JFrame **pack** routine, which computes the sizes of all components and makes the main window large enough to hold them all.

This, along with a working knowledge of the other “action” components, is enough to begin writing portable Java-based GUI applications. However, before we can approach large-scale applications, we must consider the organization and partitioning of the code, and the ideal way to handle this is with a paradigm known as MVC, or Model-View-Controller.

MVC In Java

Model-View-Controller provides a powerful model for separating the functionality of a GUI-based application into three constituent parts. Putting it simply, the model is the code that keeps track of the data. The view is the code that displays the model on screen. The controller is the code that responds to user actions such as mouse clicks, button presses and the like. This separation, first formalized in 1988 for use with Smalltalk-80, has become the dominant model for object-oriented developers building GUI-based applications. And with good reason: it partitions the code into three (or more) reasonably small modules. It provides maximum flexibility—there can be more than one view and more than one controller. In a slide show program, for example, you might have a Slide Show view and a Slide Sorter view. Either or both would be visible, each in its own window. With MVC, a change to either would immediately be reflected in all the views. So a results-oriented way of looking at MVC is a way of making all the views on your data be dynamically self-updating as the data changes.

Let's take that slide show program as a simple example, which I've called JabberPoint (no relation at all to PowerPoint). The main program (see **Listing 3**) simply creates the model, the view and the controllers, then connects them together.

The data or model is maintained by a class called JPModel. It is little more than an array of Strings, except that each line has a Style associated with it. The model also has certain other data, such as the current slide number. Plus, it has methods for updating the data. This version of the program doesn't have any slide-editing capabilities (I still use *vi* to edit the show's text), but it does have methods—in the model—to change the current slide number.

Note that this is not the full source code, but only the fragments needed to show the MVC architecture. If you want the full source code to compile or use, go to the course author's web site (see sidebar) and follow the link to Free Software.

- The model contains the data and functionality, and can be displayed by many views. It commonly includes a main program and may subclass java.util.Observable.
- The view is the GUI or display of the model's data. It commonly creates a frame, or is an applet, and adds listeners. It may implement java.util.Observer.
- The controller handles events for the model and the view. It commonly implements listener interfaces and responds to events by calling methods in the model.

The Model

Part of the model, Model.java, is shown in **Listing 4.**

The View

The simplest view is a SlideShow view, which simply paints the current page in large letters. This view is a Component that can be embedded in a Frame or an Applet.

How does it know when its data has changed? Note the method **update**. This is not the update method of AWT, but is part of the Observable interface. This update simply saves the data that was passed in as a Slide and calls AWT's **repaint**, which will call the **paint** method a few lines below it in the listing.

There can be more than one view. A slide-show program usually has three: the slide show (which we implement), the Outline and the Sorter (which we do not yet provide). Each of these would be a different view and would be registered as an Observer for the model as above. You would switch between them with a CardLayout or some kind of Tab Layout manager, or they could each be in a Frame. Since they use Observable/Observer, when you update the data in one window it would immediately be updated in all of them.

The Controllers

The controllers are called when the user does something. The KeyController.java is a simple controller that responds to PageUp and PageDown (or Enter) and moves the current page up or down as appropriate. It is “connected” with

```
   frame.addKeyListener(new KeyController(model));
```

A Controller does not have to be an explicit listener. We might, for example, use a MenuBar as a listener and connect it with the statement

```
   frame.setMenuBar(new MenuController(view,model));
```

after the instantiation of KeyController in our main program. It then calls methods on the Model, such as **nextPage**.

We can add additional functionality such as **loadFile**. When we get around to writing the editing part of this program, we can add methods such as **saveFile**, **newFile**, etc., to the model and call them from here.

One complication is that the MenuController may need access to the top-level frame (just for purposes of Dialog creation), but the view is a component inside the frame, and we don't wish View to know too much about its environment. One way around this is to pass the frame into the MenuController's constructor; another is for the view to have a **getFrame** method.

Where is Main?

The model, view and controller are usually tied together with a **main** program; the part of JabberPoint.java that sets this up is shown in the method JPMain, a “Constructor” in Listing 3.

Beyond the Basics

MVC can be more complex than this, although we've covered the basics here. For an extremely powerful (and wonderful) example, see the JFC/Swing components JTable and TableDataModel. In fact, we'll use these in our simple UNIX Administration Tool.

Java for Linux Administration

Here we present a simple example of a Linux/UNIX administration tool, a program for viewing password and group file information. It may seem strange to write system-specific administration tools in a portable language—this tool will work on most UNIX variants. And anyway, Java is a nice language to write in, and the JFC GUI components bring the creation of powerful tools to a wider audience. In particular, this tool will let us showcase the JTable widget, which provides most of the screen functionality of a spreadsheet, including dynamically-arrangeable columns and other nice options.

Since Java is portable, it doesn't provide an API for reading the system password file. We designed and wrote a class PW that has the same public members as the C-language structure returned by the system password resolvers. We also provide a “PWReader” class to read them and provide a sample implementation that just reads from a traditional format password file. This is not suitable for production use on most systems, but serves as a simple demonstration. Since these readers don't affect the GUI, we won't discuss them in detail here, but the code for both is on-line.

Displaying and Searching the Password Information

Since we want this to be a “good” application and maybe the basis for a general UNIX user database editor (read, write, validate) later on, we'll design it according to the model-view-controller paradigm from the start. I called this program **Ued**, originally in tribute to a much older program written at the University of Toronto around 1982 and maintained for a time by my colleague Geoffrey Collyer. My program has no code in common with that older ued. The class **UedModel** (see UedModel.java) is the user data portion of the program. **UedView** displays a list of users or groups on the screen. **UedControl** responds to user requests to modify the data. The main thing to note is the look-and-feel it presents (see Figure 5).

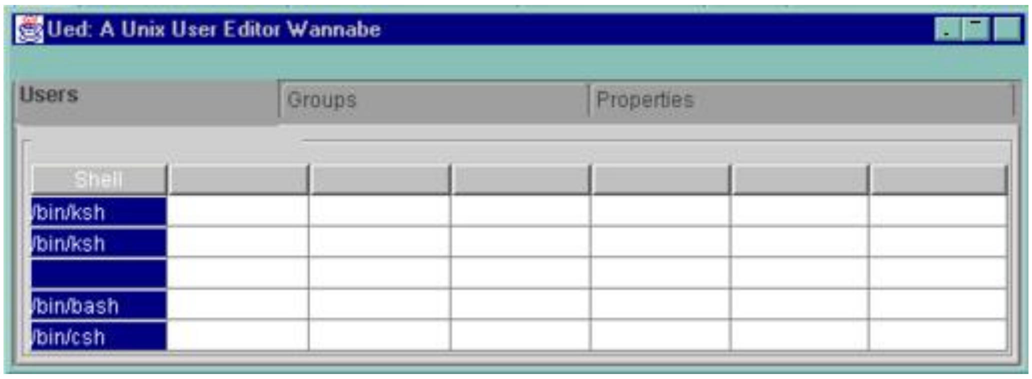


Figure 5. Ued Screenshot

Note that you can drag columns around. If we wanted the user to be able to sort by user ID, for example, we'd have our sort routine interrogate the “table model” to see the current column order and use that for sorting. You can select a column by clicking on its title. (This feature isn't used here, but would be in a spreadsheet.) Or you can select all the fields in a row (one user) by clicking anywhere. This would be used in a menu-based “Delete” operator, for example.

How does the data get into the table? The nice thing about JTable is that it specifies a helper class called a JTableModel, which is the interface between your data model and the JTable. Once we have a data model based on PW objects as described above, the JTableModel need only obtain the individual fields for the table and return them to the JTable upon request. See source file UedTableModel.java, which is only about 40 lines long, most of it is just a switch statement. Again, JFC's object model makes code development easy.

Note also that the main program is in a tabbed layout. Group and Properties tabs are also present and not yet implemented, but they do show how easy it is to use the JTabLayout. We just write:

```
JTabbedPane mainPane = new JTabbedPane();
 add tabbed pane to Frame
cp.add(BorderLayout.CENTER, mainPane);
 add user view to tab
mainPane.addTab("Users", uv);
mainPane.addTab("Groups",
 new JLabel("Not Written Yet", JLabel.CENTER));
mainPane.addTab("Properties",
 new JLabel("Not Written Yet", JLabel.CENTER));
```

From then on, management of the tab view is automatic—when the user clicks on a tab, its content is brought to the fore and displayed.

The neat thing about JTable/JTableModel is that you can easily make any table editable just by following these three steps:

1. Write a routine **isEditable** that returns **true**.
2. Provide a **CellEditor**, which can be a wrapper on a **TextField** (then you need only double-click in a cell to start editing it).

3. Write a **setValueAt** routine for the TableModel to call to set the values in your program when the user changes them on-screen.

That's all you need, although in a real application you would also do some error checking and set a “save needed” flag. The password editor in the Ued program does this. In effect, the JTable widget gives you almost all of the user interface portion of a spreadsheet, and it is just one of the many great widgets included in the Swing Set of JFC. And that's just one piece of the new functionality included in Java 1.2.

Java in the Crystal Ball

The near future for Java shows no letup in the rapid rate of innovation. JFC has just been released, with the 1.2 version of Java. Many promising technologies are just on the horizon, including 3-D, JTAPI, Java Sound, Java Speech and many others. Since there is far too much alphabet soup to remember, please check out the JavaSoft API page at http://java.sun.com/products/api-overview.html. The 3-D API tries to provide a comprehensive imaging model for three-dimensional graphics with some of the best features of PEX, GL and friends. JTAPI lets Java programs control telephony equipment at all scales, from a single voice-mail modem up to a large Private Branch Exchange (PBX). Java Media Framework gives access to all kinds of image, audio and video recording/ playback, including Java Sound. Java Sound will provide several sound formats from simply playing sound files (available in 1.2), to recording, to full control over synthesizers such as MIDI. Java Speech will include both speech synthesis and speech recognition.

Many contact tracker systems are available from the simple (my own freeware JabaDex) to the fancy ones limited to MS-Windows, such as Symantec ACT. When Java Sound and JTAPI are released, developers of contact tracker systems can write code to dial the phone, answer it and incorporate voice mail, maybe even add bidirectional FAX support. We will no longer have to write it once for Linux, again for MS-Windows, again for Macintosh and again for Solaris. We will be able, as JavaSoft's slogan promises, to “write once, run anywhere”.

Java Networking API

Resources

Ian Darwin has used UNIX systems since 1980 (mostly Solaris and OpenBSD in the last few years) and used Java heavily since 1995. He is the author of JabaDex (a 5,000-line Rolodex application entirely in Java), two textbooks (Checking C Programs with Lint*, published by O'Reilly, and* X User's Guide Volume 3: OPEN LOOK Edition*, available on CD-ROM) and more recently, two four-day Java*

Programming courses through Learning Tree International. E-mail him at ian@darwinsys.com.

Archive Index Issue Table of Contents

Advanced search

DSP Software Development

Ian V. McLoughlin

Issue #61, May 1999

Follow the development of speech algorithms for digital radios through the complete project life cycle.

In this article, I describe a Linux success story based on researching and developing DSP (Digital Signal Processing) speech coding algorithms. I chose Linux over Windows for good reasons—reasons that may provide you with ammunition to persuade the bosses that Linux really does mean business. To emphasize this, I developed the software for the next generation of digital radio products in the headquarters of the world's largest private mobile radio manufacturer.

Luckily, I had an open-minded boss, but there were still difficulties. These included interoperability issues with existing systems, resource sharing, accessibility, documentation and the non-availability of some crucial software for Linux.

Project Life Cycle

A typical project life cycle begins with university research and proceeds through initial investigation and prototyping, a complex coding route and various testing stages to a fully documented software package for passing on to system integrators.

My project was advanced speech processing software for fixed-point DSP. Bearing this in mind, audio capabilities topped the list of requirements for any development machine. Also needed were good mathematical processing and visualization software and a whole set of code-development tools. Finally, some DSP-specific software was required.

Research

Given recorded speech files, research often involves processing and evaluating the changes by listening to them. For this, a sound card is useful, and with the availability of the OSS drivers, sound output is no problem for Linux. The easiest way to generate sound is to copy a sound data file to /dev/audio. Tradition specifies this data file should be in Sun's 8-bit logarithmic format, sampled at 8KHz. The command

```
cp audiofile.au /dev/audio
```

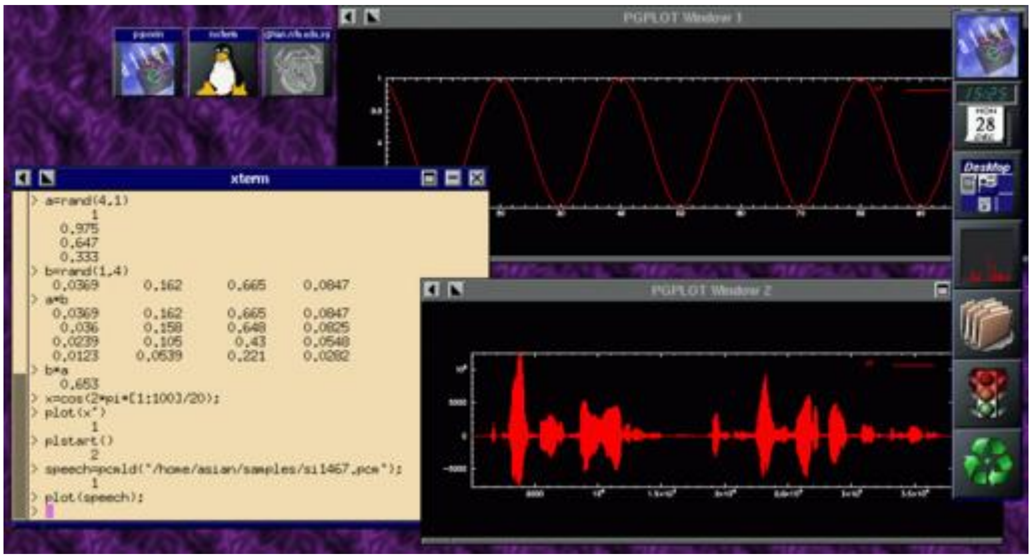
outputs sound, assuming everything is set up properly (see Resources for good audio information).

How do you get Sun format audio? The answer is to use **sox** (SOund eXchange). Its command-line options seem a bit unfriendly at first, but the following command converts a .wav file into a Sun format .au file:

```
sox audiofile.wav -t ul -r 8000 audiofile.au
```

Traditional processing is done by writing a C or similar program to read in the speech file, perform some processing and write the output either directly to /dev/audio (if the program can output the data in real time) or to a temporary file first (if it cannot). This works okay, but the *compile-link-test-modify* cycle can be too lengthy to permit efficient trial-and-error testing (sometimes called research).

One alternative is **MATLAB**, the excellent commercial mathematical manipulation package; however, I found an alternative with a GPL—**Rlab**. Although not promoted as a MATLAB clone, this high-quality suite of software is at least as usable, truly multi-platform and free. The range of built-in functions in Rlab is impressive and allows the seamless addition of user functions. Data can be imported/exported, processed and displayed graphically, as shown in Figure 1. See Resources for some useful additional Rlab functions, including an audio playback routine.



```
> a=rand(4,1)
        1
   0,975
   0,647
   0,333
> b=rand(1,4)
  0,0369      0,162      0,665     0,0847
> a*b
  0,0369      0,162      0,665     0,0847
   0,036      0,158      0,648     0,0825
  0,0239      0,105       0,43     0,0548
  0,0123     0,0539      0,221     0,0282
> b*a
   0,653
> x=cos(2*pi*[1:100]/20);
> plot(x')
        1
> plstart()
        2
> speech=pcmld("/home/asian/samples/si1467.pcm");
        1
> plot(speech);
> 
```

Figure 1. Using Rlab to Modify and Plot Speech Waveforms

All this gives us an ideal platform for speech algorithm research. We can listen to audio, make modifications and build up a library of speech processing routines to use in current and future investigations. The modifications can be tried and evaluated with little effort.

Prototyping

Now the algorithms found after playing with Rlab must be converted manually to DSP code. This isn't actually easy for a number of reasons: the Rlab code makes use of built-in library routines and is floating-point. The humble DSP is only fixed-point, so normally this conversion is done in three steps.

The first is to go directly from the Rlab script to C—replicating all the Rlab functions with C functions of the same name, which you write and test, and rewriting the glue code. This produces an executable that is bit-exact with the Rlab code, so halting or single-stepping the code with **gdb** can allow direct comparisons between the C and the Rlab script.

The second step is to replace all the floating-point variables and functions with fixed-point alternatives. For each data variable, we need to know maximum and minimum values and the effect of truncation, then truncate and scale appropriately.

For trigonometric functions, a number of established techniques such as approximation and table lookup may be used, but these can be difficult to code. It doesn't help that DSP memory is extremely limited and the law of code size applies (i.e., code size will expand until it's just larger than the available space).

However, fiddling with numbers can be good fun, and trying to write a fixed-point **log** function by hand may require a few more visits to Rlab in order to work out exactly what a logarithm does.

Eventually, a C program emerges that has no floating-point variables (use **grep** to make sure); other than scaling and truncation errors, it performs the same function as the original Rlab code. Again, gdb can be used to investigate execution. We can import array data to Rlab for plotting by selecting the data and dropping it into an Rlab script.

One trick is to write a C function that, when passed an array, prints the array formatted so that it can be selected and pasted into Rlab—code such as:

```
void rprint(int length, int *array) {
   printf("\narray=[");
   for (int i=0;i<length;i++)
      printf("%d,",array[i]);
   printf("\b];\nplot(array)\n"); }
```

When used frequently in the debugging cycle, this can be very effective.

The last point to mention under prototyping is the benefit of using some form of version control, or perhaps I should say the foolishness of not using it. Effective version control is one of the major reasons UNIX/Linux is a stable and capable development platform. We used RCS throughout the development process. In fact, the main RCS directory was on a Sun accessed via an NFS mount and shared by a number of developers working under Solaris.

Development

Now comes the DSP involvement—for this a DSP starter kit is needed. For simple and easy development, two main contenders are available. Both have patchy support for Linux, cost in the region of £80 and are aimed at the hobbyist, small business or university user.

The original was from Texas Instruments, the TMS320C50 DSK (there was an earlier, less powerful C26 board), with the newer contender being the Analog Devices ADSP2181 EZ-KIT Lite. Both have audio I/O—the latter has 16-bit CD-quality stereo audio, while the former can manage only 14-bit voice quality. On the software side, both provide a nice set of DOS executables—assembler, linker and (for the Analog Devices kit) a simulator. The ADSP has an edge with its assembly language syntax being much more user-friendly than the TI chip. I won't stick my neck out too far and comment on which DSP is more powerful—both are fairly competent.

Linux versions of most DSP development tools are floating around on the Internet, but some are still missing, notably for the ADSP2181. These omissions are the assembler, linker and simulator, which is a pity since I had to use the ADSP.

The freely available cross-assembler **as** will soon include ADSP21*xx* compatibility. It already handles TMS320C*xx* code along with a staggeringly wide array of other processors, with more added whenever the author, Alfred Arnold, has free time. Analog Devices have been approached about providing Linux versions of assembler and linker, but stated they do not currently have plans to support Linux.

For DSP code development, we need an assembler, linker and a code downloader that sends an executable through the PC serial port to the DSP development board. For the ADSP21*xx*, few Linux tools are available just now, only the downloader.

The solution is to use DOSEMU, the Linux DOS emulator, which has an impressive feature called the **dexe** (directly executable DOS application). This is basically a single DOS file or application in a tiny DOS disc image that can be executed within Linux without the user being aware that it is actually a DOS program.

To use this method, the entire ADSP21*xx* tool set can be incorporated into a single .dexe file. With a little ingenuity, a few simple shell scripts and batch files, the user will never know the assembler and linker he is using are actually DOS programs (see Resources for a HOWTO).

With the newly created dexe, we now have an assembler and a linker for our DSP code. Hidden in the depths of the Analog Devices web site is the source code for a UNIX (Linux/Sun) download monitor to load the DSP executable into the EZ-KIT Lite through the PC serial port. This means the assembler source can be compiled and downloaded all (more or less) under Linux.

The one irritation is the simulator. Analog Devices supply a DOS version of their simulator which will not run under the emulator, but this is no reason to throw Linux out, as we shall see later.

Analog Devices does have a 21*xx* C compiler based on good old **gcc** and even released the source. The C code integrates neatly with the assembly language and speeds up development time, but it is quite inefficient both in terms of code size and instruction cycles.

Completion

We now have an algorithm that runs on a DSP system. The complete software package generated by this effort includes:

- Rlab research and investigation scripts
- Test vectors and speech files from Rlab
- Floating-point C implementation
- Fixed-point C implementation
- Assembly language version of the code
- A working DSP executable

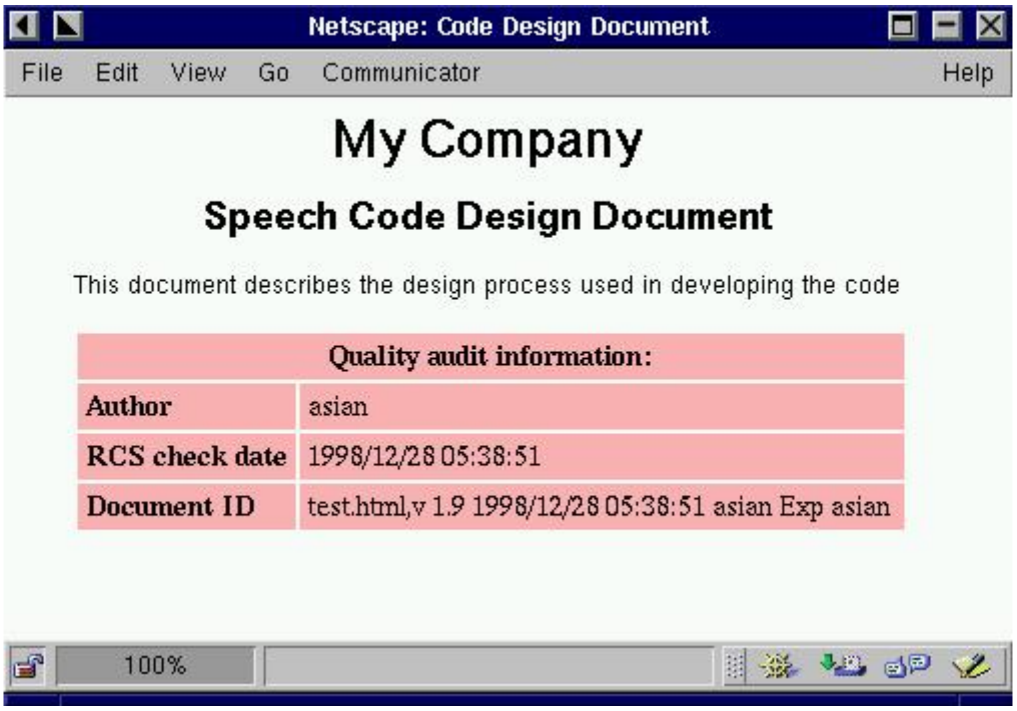
Does this list look complete to you? If so, you must be a born programmer like me. Anyone else would realize that documentation is missing.

Documentation

Has this happened to you? When your management says documentation must be in a standard format, you think **LaTeX** and they think Microsoft Word. ASCII is insufficient because of the lack of text formatting and graphics support.

However, one irrefutable standard that even your boss can agree to is HTML. Once a common standard has been agreed upon, it is time to produce a set of documentation templates. After that, any editor can be used to add content, including Netscape **composer**, Emacs or even Word. Graphics are more of a problem, but a combination of **xfig** and **GIMP** can handle most situations. The resulting web documentation can be read under Linux, Windows, RISC OS, etc. and is even accessible on palmtop computers.

We used RCS to manage our documentation versions too, in order to comply with company quality control standards. This allows a construct such as **<li>RCS id: Id</li>** to be embedded in the HTML. When the HTML document is checked into RCS, the RCS identifier will be inserted between the “$” symbols and will therefore be displayed on the HTML page.


Netscape: Code Design Document
File Edit View Go Communicator Help
100%

My Company

Speech Code Design Document

This document describes the design process used in developing the code

Quality audit information:	
Author	asian
RCS check date	1998/12/28 05:38:51
Document ID	test.html,v 1.9 1998/12/28 05:38:51 asian Exp asian

Figure 2. RCS Information Used in Inter/Intranet-Based Documentation

A prettier method is to use JavaScript for display in Netscape to format the page and remove the unwanted $ symbols. The HTML page in Listing 1 forms the front cover to some code documentation, as shown in Figure 2.

Listing 1.

We all know HTML isn't perfect, but at least it is a compromise that can be agreed upon in striving toward a paperless office. Some other features we incorporated were placing the RCS log entries into a scrollable text area on the HTML pages and judicious use of hyperlinks to commented source code, data flow diagrams and flow charts.

To enhance our documentation, the C prototype code was compiled using **GCC -pg** which inserts extra code to write a profiling information file during program execution. Then **gprof** was used to interpret this profiling information. **xfig** was used to manually convert this into a function-call, graph GIF, and a sensitive image map was created for it. A set of HTML templates was created and edited to document each function; these pages can be accessed by clicking on this top-level GIF.

The result was a single HTML page showing the entire code in a pyramidal layer structure starting from **main** and the calling links between each function, with passed variable names written next to each calling link. The functions were named inside clickable boxes, which pointed to an explanation of that function.

This HTML documentation process is now being automated; see Resources for more information.

As an added bonus, my colleagues used the new documentation standard to justify buying more Linux machines. One was used to serve the documents on the company intranet using the Apache web server. This system can control access to the documents on a need-to-know basis, and keep a log of user accesses versus date and document version. It is even possible to automatically notify affected parties by e-mail when a document they accessed recently has changed.

Alternatives

Finally, let's consider the alternatives to Linux. The Analog Devices tools supplied with the EZ-KIT all run under DOS and are command-line programs. Of course, they could be run from a Windows DOS prompt, but this provides no advantage over Linux. Furthermore, an xterm is more flexible than a Windows DOS prompt, especially when you want to refer back to a page of error messages that flashed past. Also, the ADSP21*xx* simulator will not run under Windows, which would have to be rebooted cleanly into DOS, just as a Linux machine that needed to run the simulator would.

UNIX versions of the tools are supplied by Analog Devices at extra cost and are functionally identical to the DOS versions. However, they run only under SunOS; they do not run under newer versions of Solaris.

MATLAB is available for Linux, other UNIX systems and Windows, as is Rlab, but I would argue that only the flexibility of a UNIX operating system can allow the full use of these applications to interact with other command-line-based code development and debugging tools. Of course, debugging tools are available for all platforms. They may sometimes be more user friendly, but are probably less capable than gdb and are seldom freely available.

Revision control systems are also available for many platforms, but not all can cope with code development and integrate with a hyperlinked HTML-based documentation system being served via Apache. The revision control system you choose must also have the capability to interface with your favourite editor and be utilized within the **make** hierarchy.

Summary

Obviously, Linux makes a good DSP development system. All you need to buy is a DSP starter kit—everything else is on your installation CD or freely downloadable. This system has been used in the real world—it takes a little setting up, but it works. It is reliable and a lot more fun than Windows.

In the future, it will only get better: more DSP development tools will be available under Linux. I encourage you all to advocate the use of Linux-based development systems for both university and corporate research and development.

Resources

Ian V, McLoughlin (asian@ntu.edu.sg) has been programming since he got his first home computer, a BBC Micro in 1983. As well as continuing with Acorns, he enjoys using Linux. He is now passing his experiences on to the younger generation in Singapore (human programming). During those brief moments when not in front of a computer, he and his wife enjoy traveling, eating and anything Chinese.

Introduction to Multi-Threaded Programming

Brian Masney

Issue #61, May 1999

A description of POSIX thread basics for C programmers.

The purpose of this article is to provide a good foundation of the basics of threaded programming using POSIX threads and is not meant to be a complete source for thread programming. It assumes the reader has a good strong foundation in C programming.

A thread is sometimes referred to as a lightweight process. A thread will share all global variables and file descriptors of the parent process which allows the programmer to separate multiple tasks easily within a process. For example, you could write a multi-threaded web server, and you could spawn a thread for each incoming connection request. This would make the network code inside the thread relatively simple. Using multiple threads will also use fewer system resources compared to forking a child process to handle the connection request. Another advantage of using threads is that they will automatically take advantage of machines with multiple processors.

As I mentioned earlier, a thread shares most of its resources with the parent process, so a thread will use fewer resources than a process would. It shares everything, except each thread will have its own program counter, stack and registers. Since each thread has its own stack, local variables will not be shared between threads. This is true because static variables are stored in the process' heap. However, static variables inside the threads will be shared between threads. Functions like **strtok** will not work properly inside threads without modification. They have re-entrant versions available to use for threads which have the format *oldfunction*_r. Thus, strtok's re-entrant version would be **strtok_r**.

Since all threads of a process share the same global variables, a problem arises with synchronization of access to global variables. For example, let's assume you have a global variable X and two threads A and B. Let's say threads A and B will merely increment the value of X. When thread A begins execution, it copies the value of X into the registers and increments it. Before it gets a chance to write the value back to memory, this thread is suspended. The next thread starts, reads the same value of X that the first thread read, increments it and writes it back to memory. Then, the first thread finishes execution and writes its value from the register back to memory. After these two threads finish, the value of X is incremented by 1 instead of 2 as you would expect.

Errors like this will probably not occur all of the time and so can be very hard to track down. This becomes even more of a problem on a machine equipped with multiple processors, since multiple threads can be running at the same time on different processors, each of them modifying the same variables. The workaround for this problem is to use a **mutex** (mutual exclusion) to make sure only one thread is accessing a particular section of your code. When one thread *locks* the mutex, it has exclusive access to that section of code until it *unlocks* the mutex. If a second thread tries to lock the mutex while another thread has it locked, the second thread will *block* until the mutex is unlocked and is once more available.

In the last example, you could lock a mutex before you increment the variable X, then unlock X after you increment it. So let's go back to that last example. Thread A will lock the mutex, load the value of X into the registers, then increment it. Again, before it gets a chance to write it back to memory, thread B gets control of the CPU. It will try to lock the mutex, but thread A already has control of it, so thread B will have to wait. Thread A gets the CPU again and writes the value of X to memory from the registers, then frees the mutex. The next time thread B runs and tries to lock the mutex, it will be able to, since it is now free. Thread B will increment X and write its value back to X from the registers. Now, after both threads have completed, the value of X is incremented by 2, as you would expect.

Now let's look at how to actually write threaded applications. The first function you will need is **pthread_create**. It has the following prototype:

```
   int pthread_create(pthread_t *tid,
      const pthread_attr_t *attr,
      void *(*func)(void *), void *arg)
```

The first argument is the variable where its thread ID will be stored. Each thread will have its own unique thread ID. The second argument contains attributes describing the thread. You can usually just pass a NULL pointer. The third argument is a pointer to the function you want to run as a thread. The final argument is a pointer to data you want to pass to the function. If you want to exit from a thread, you can use the **pthread_exit** function. It has the following syntax:

```
void pthread_exit(void *status)
```

This will return a pointer that can be retrieved later (see below). You cannot return a pointer local to that thread, since this data will be destroyed when the thread exits.

The thread function prototype shows that the thread function returns a void * pointer. Your application can use the **pthread_join** function to see the value a thread returned. The pthread_join function has the following syntax:

```
int pthread_join(pthread_t tid, void **status)
```

The first argument is the thread ID. The second argument is a pointer to the data your thread function returned. The system keeps track of return values from your threads until you retrieve them using pthread_join. If you do not care about the return value, you can call the **pthread_detach** function with its thread ID as the only parameter to tell the system to discard the return value. Your thread function can use the **pthread_self** function to return its thread ID. If you don't want the return value, you can call **pthread_detach(pthread_self())** inside your thread function.

Going back to mutexes, the following two functions are available to us: **pthread_mutex_lock** and **pthread_mutex_unlock**. They have the following prototype:

```
int pthread_mutex_lock(pthread_mutex_t *mptr)
int pthread_mutex_unlock(pthread_mutex_t *mtr)
```

For statically allocated variables, you must first initialize the mutex variable to the constant **PTHREAD_MUTEX_INITIALIZER**. For dynamically allocated variables, you can use the **pthread_mutex_init** function to initialize a mutex variable. It has the following prototype:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
   const pthread_mutexattr_t *mutexattr)
```

Now we can look at actual code as shown in Listing 1. I have commented the code to help the reader follow what is being done. I have also kept the program very basic. It does nothing truly useful, but should help illustrate the idea of threads. All this program does is initiate 10 threads, each of which increments X until X reaches 4,000. You can remove the pthread_mutex_lock and unlock calls to further illustrate the uses of mutexes.

Listing 1. Example Program

A few more items need to be explained about this program. The threads on your system *may* run in the order they were created, and they *may* run to completion before the next thread runs. There is no guarantee as to what order the threads run, or that the threads will run to completion uninterrupted. If you put “real work” inside the thread function, you will see the scheduler swapping between threads. You may also notice, if you take out the mutex lock and unlock, that the value of X may be what was expected. It all depends on when threads are suspended and resumed. A threaded application may appear to run fine at first, but when it is run on a machine with many other things running at the same time, the program may crash. Finding these kinds of problems can be very cumbersome to the application programmer; this is why the programmer must make sure that shared variables are protected with mutexes.

What about the value of the global variable *errno*? Let's suppose we have two threads, A and B. They are already running and are at different points inside the thread. Thread A calls a function that will set the value of *errno*. Then, inside thread B, it will wake up and check the value of *errno*. This is not the value it was expecting, as it just retrieved the value of *errno* from thread A. To get around this, we must define **_REENTRANT**. This will change the behavior of *errno* to have it point to a per-thread *errno* location. This will be transparent to the application programmer. The **_REENTRANT** macro will also change the behavior of some of the standard C functions.

To obtain more information about threads, visit the LinuxThreads home page at http://pauillac.inria.fr/~xleroy/linuxthreads/. This page contains links to many examples and tutorials. It also has a link where you can download the thread libraries if you do not already have them. Downloading is necessary only if you have a libc5-based machine; if your distribution is glibc6-based, LinuxThreads should already be installed on your computer. The source code for threaded application that I wrote, gFTP, can be downloaded from my web site at http://www.newwave.net/~masneyb/. This code makes use of all concepts mentioned in this article.

Resources

Brian Masney is currently a student at Concord College in Athens, WV. He also works as a computer technician at a local computer store. In his spare time, he enjoys the outdoors and programming. He can be reached at masneyb@newwave.net.

Red Hat Motif 2.1 for Linux

John Kacur

Issue #61, May 1999

Motif has become a standard in the UNIX world and is the basis for the common desktop environment (CDE).

Red Hat Motif 2.1 for Linux

- Manufacturer: Red Hat Software
- E-mail: info@redhat.com
- URL: http://www.redhat.com/
- Price: $149 US
- Reviewer: John Kacur

Motif is a windowing system and environment developed by the Open Software Foundation (OSF). The Motif Xm library is a software layer used with Intrinsics' Xt library and the Xlib library of the X Window System. According to the Motif user-interface specification, Motif is independent of how it is implemented, so it is theoretically possible to implement the Motif GUI on a different windowing system. PC users will immediately recognize the similarity of the Motif GUI to the Microsoft Windows 3.*x* and OS/2 GUI.

Motif has become a standard in the UNIX world and is the basis for the common desktop environment (CDE). The Motif 2.1 release placed great emphasis on compatibility with CDE. Since CDE is based on Motif 1.2, some features and components available in Motif 2.0 are no longer supported under Motif 2.1. This provides the programmer with a minimum of portability problems in designing Motif-based programs for a variety of UNIX systems.

Motif is an unusual choice in the Linux and free software world, as it is commercially licensed software. The license cost about $149 US at the time of this writing (early 1999), and you must purchase a license for each copy of Motif you run. You may not redistribute the Motif library with your software, but you may freely distribute statically linked binaries created for people who don't have a copy of Motif on their system. The Netscape browser is an example of a statically linked Motif application. People who have Motif libraries can compile their software dynamically, creating faster and smaller binaries.

The most compelling reason for Linux users to use Motif is the ability to create programs for commercial UNIX systems on their home computer. However, many free software programmers prefer to work with totally free toolkits such as gtk+. Another alternative is Lesstif, which is a free Motif work-alike. Lesstif is still considered alpha-release software, but is a good place to start if you want to teach yourself some of the fundamentals of Motif programming.

I tested Motif 2.1 from Red Hat Software. Red Hat gets its Motif license from Metro Link, which in turn licenses it from the OSF. Although I have been very happy with my Red Hat product, I've noticed a new Motif 2.1.10 release is available. This is not a major upgrade but mostly bug fixes. I sent an e-mail message to support, asking about the possibility of an upgrade. They promptly and kindly told me I could get an upgrade from Metro Link for a reduced price. Because this was only a minor upgrade, I didn't feel that even the reduced price was warranted. Red Hat did tell me they would understand if I wanted to return the product, but they didn't have any arrangement for an upgrade because of the licensing agreement.

In the Package

The Red Hat Motif includes 30 days of installation support, shared and static libraries, man pages, the UIL compiler, the MWM window manager, Motif demo programs with source code, a printed user manual and a lot of other documentation in PostScript form on CD. As a bonus, you get the KL Group's XRT Professional Developer's suite of widgets. This version is fully licensed, but if you require support, you purchase it separately from the KL Group. These widgets include 2-D and 3-D graph and chart widgets and XRT gear which includes tabs, tree widgets and various icons.

If you purchase Motif directly from Metro Link, you can get a product called Motif Complete. Motif Complete provides you with Motif 1.2, 2.0 and 2.1 on one CD, so that you can create a custom installation.

Getting Started

The installation process is quite straightforward and adequately documented. Just be sure to mount your CD with the `exec` option turned on, as explained in the README. This allows the programs to be executed directly from the CD. The process is slightly different for different systems, depending on whether your system is a.out or ELF, and also whether your system can use RPM. If you have Slackware, for example, you need to run the **instelf.sh** program. With Red Hat, you can use **glint** or any other RPM-based tool.

Next, you must edit (or create) an .Xclients or .xsession file to use MWM as your window manager. I chose the .Xclients method, which allows you to leave your .xinitrc file in place. Here is a sample .Xclients file:

```
nxterm -geometry 80x44+0+2 +ut &
nxterm -geometry 80x50+509+2 &
# Color for the display or root window
# doesn't need to be put in the background
xsetroot -solid CadetBlue
xscreensaver &
exec mwm
```

This starts two nxterm windows, sets the background color to CadetBlue, starts the **xscreensaver** program and the Motif Window Manager. Notice that you use the normal X Window System programs to do things such as set the background color. You can get a list of color names for **xsetroot** in the /usr/lib/X11/rgb.txt file. If you are creating the .Xclients file from scratch, don't forget to make it executable with **chmod +x**.

Next, the two places to customize resources are the .Xdefaults file and the .mwmrc file in your home directory. The .Xdefaults file most likely already exists, and only needs some lines appended to it. The .mwmrc file should be copied to your home directory as follows:

```
cp /etc/X11/system.mwmrc ~/.mwmrc
```

Note the User Guide is in error and says the system.mwmrc file is found in /usr/lib/X11/system.mwmrc.

Now you can have some fun customizing your environment and creating menus. For example, I have these lines appended to my .Xdefaults file:

```
Mwm*activeBackground: CadetBlue
Mwm*UseIconBox: true
Mwm*keyboardFocusPolicy: pointer
```

The general format for these lines is **Mwm*resource:** ***`value`***. In my file, I've defined the **activeBackground**, i.e., the window which has the focus, as CadetBlue, which is the same as the background color I set in my .xinitrc file. Setting the **keyboardFocusPolicy** to **pointer** means moving the mouse pointer to another window gives that window the focus automatically. Setting the keyboardFocusPolicy to **explicit** would require you to explicitly click your mouse button in the window to give it focus.

The **UseIconBox** default is **false** which means when you minimize a window, its icon appears in the root window. Setting the UseIconBox to **true** creates an MWM window which holds the icons. See Figure 1 for an example of an icon box. The dark black line around the xterm icon indicates that the xterm window has the focus. The frame around the Netscape icon indicates that it is iconized, and the lack of frames around the nxterms, Mail and nedit icons indicates that the windows are not iconized.

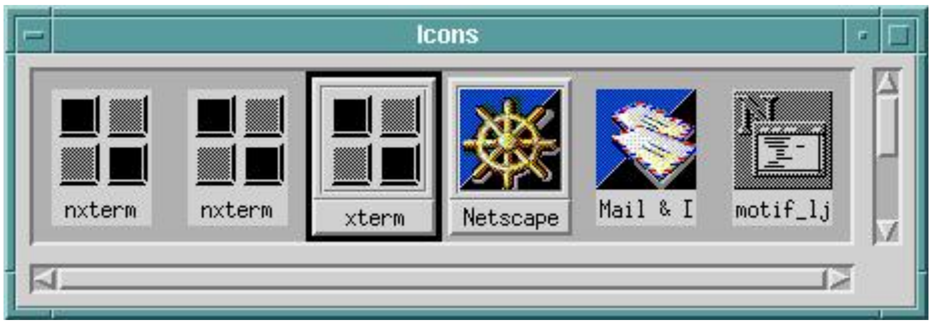


Figure 1. Icon Box

Listing 1.

Let's take a look at the .mwmrc file. Listing 1 is a portion of my modified .mwmrc file copied from system.mwmrc. The complete file can be found in the archive file ftp://ftp.linuxjournal.com/pub/lj/listings/issue61/3218.tgz. Make sure to enclose your items in exclamation points if they contain spaces. Mnemonics and accelerators are optional. Mnemonics are one of the letters in the item which should appear underlined. Once the menu is chosen, typing the mnemonic is the same as clicking on the item. An accelerator is a series of keystrokes for accomplishing the function without using the menu at all. Functions include titles, exec, separator and menu. Titles are the titles in the menu (see Figure 2), and separators draw a line in the menu. The program you want to start is lauched by exec. You can specify the full path name of the program or just its name, if your path variable is set correctly. Menus are the sublevel menus. For example, I've created a menu item “games” and the name of the menu is Games Menu. Then, later in the listing, you have the definition for the Games Menu. Customizing your environment in Motif is easy and fun.

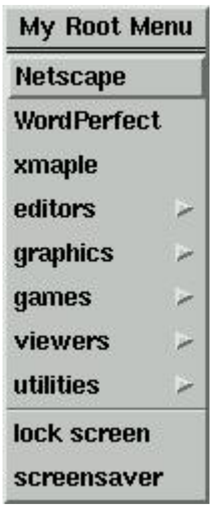


Figure 2. My Root Menu

Compiling Programs

Last but not least, you can use your Motif libraries to compile programs which are dynamically linked, which should make the binaries smaller and quicker.

An example I suggest trying is NEdit. This is a nice WYSIWYG (what you see is what you get) editor available from ftp://ftp.fnal.gov/. If you don't have Motif, you can still use the statically linked version of this editor, or try to compile it with Lesstif. Compiling this program on my system gave me errors of this type:

```
/usr/X11R6/lib/libXm.so: undefined reference to 'XpEndJob'
/usr/X11R6/lib/libXm.so: undefined reference to
XpSelectInput
/usr/X11R6/lib/libXm.so: undefined reference to
XpGetPdmStartParams
```

The libXp library comes in XFree86-devel, so the Makefiles which come with NEdit must be modified to include **-lXp**. You can examine the Makefiles which come with the demo programs (see Figure 3) to give you clues to other libraries which are not properly linked.

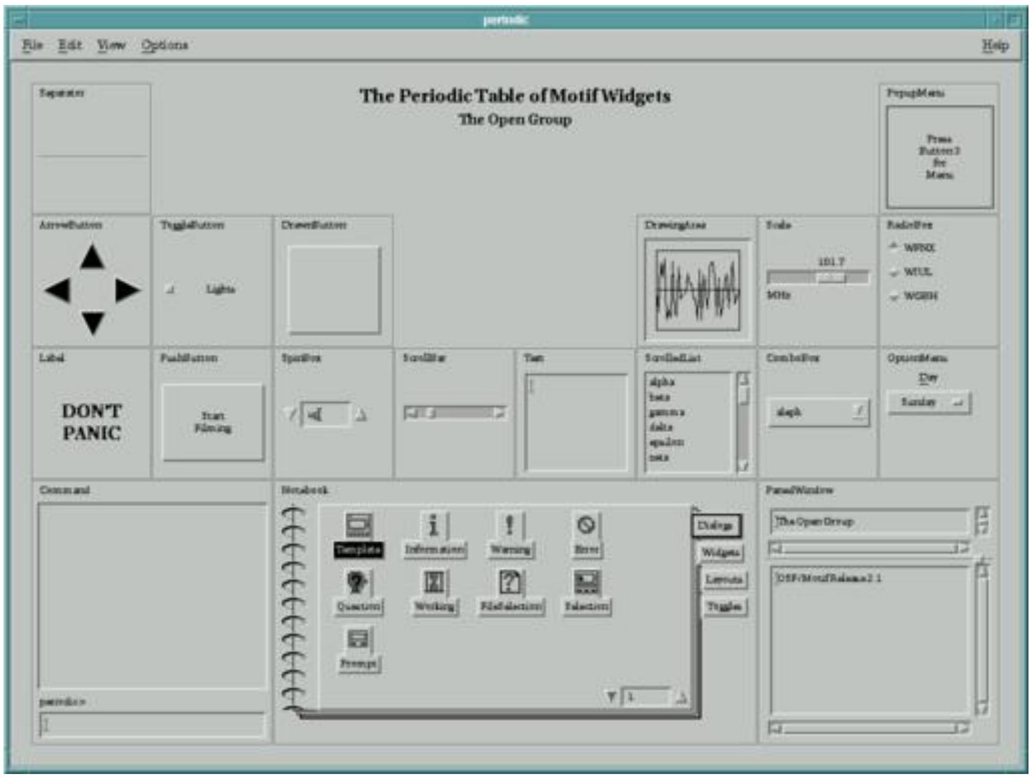


Figure 3. Motif Widgets Provided in the Demo Programs

Conclusion

Motif isn't a necessity for the average Linux user, but it does provide you with a clean interface and a standard GUI model for the UNIX environment. It is especially nice for programmers who want to use their Linux boxes to write software that will easily port to proprietary UNIX systems.

Listings

Resources

John Kacur (jkacur@acm.org) has a B.A. in Fine Arts. After two years studying Russian in the Ukraine and two more years teaching English in Germany, John returned to Canada to pursue a second degree in Computer Science.

Linux Programmer's Reference

Andrew G. Feinberg

Issue #61, May 1999

This book touches on almost every aspect of writing an application for Linux.

- Author: Richard Petersen
- Publisher: Osborne/McGraw-Hill
- URL: http://www.osborne.com/
- Price: $16.99 US, $24.95 CAN
- Reviewer: Andrew G. Feinberg

Last summer, I picked up *Linux Programmer's Reference* looking for a good volume on kernel internals or on writing modules. Instead, I found a major shell scripting tutorial and introductory lessons in C, Tcl/Tk, TeX/LaTeX, the use of **make**, RPM and writing man pages. The table of contents lists the following chapters:

1. BASH Shell Programming
2. TCSH Shell Programming
3. Z Shell Programming
4. Compilers and Libraries: G++, GCC, and GDB
5. Development Tools
6. Perl: Quick Reference

7. Tcl and Tk
8. TeX and LaTeX

The titles of Chapters 1 through 3 explain their contents. Chapter 4 is not as much a C tutorial as a compiler reference, including the basics of the GNU Debugger. Chapter 5 has a wonderful section on basic development tools. It teaches the fundamentals of managing a package with make and RCS and writing documentation. The use of **autoconf** and RPM, touched on in Chapter 4, should probably have been placed in Chapter 5. Chapter 6 is just what it states, although I recommend O'Reilly's *Programming Perl* (the Camel Book) for those who want to learn that wonderful language. Chapter 7 is a good start for beginners wishing to get comfortable with Tcl and Tk programming. Chapter 8 provides instruction on those formatting languages with which you can typeset books about your applications, if you feel the need.

This book touches on almost every aspect of writing an application for Linux. The shell scripting sections are the best I have seen. I was attracted to the Z Shell section in particular, since I have never seen much documentation for that shell, which is my personal favorite. I am already a fan of Perl, so Chapter 3 didn't add much for me; however, Chapter 5 blew me away. Covered here is material I have found before only in separate books.

Linux Programmer's Reference is a small book that seldom goes into much detail. However, I can say that this little text is a perfect companion for anyone —from the “hacks-binary-code-for-fun” type to the “I-want-to-give-this-cool-program-I-wrote-to-my-friends” type. As someone decidedly in between these two, I would definitely say this book has something for everyone.

Andrew G. Feinberg is a student at Walt Whitman High School in Bethesda, Maryland. In his spare time, he is a developer for Debian GNU/Linux and runs the High School Linux User Group (http://hs-lug.tux.org/). He can be reached at andrew@ultraviolet.org.

Archive Index Issue Table of Contents

Advanced search

An Overview of Intel's MMX Technology

Ariel Ortiz Ramirez

Issue #61, May 1999

An introduction to MMX and how to take advantage of its capabilities in your program.

Commercially introduced in January 1997, the MMX technology is an extension of the Intel architecture that uses a single-instruction, multiple-data execution model that allows several data elements to be processed simultaneously. Applications that benefit from the MMX technology are those that do many parallelizable computations using small integer numbers. Examples of these kinds of applications are 2-D/3-D graphics, image processing, virtual reality, audio synthesis and data compression.

If your Linux system has a Pentium II or a Pentium with MMX technology, you can build programs that take advantage of the MMX instruction set using **gcc** and a bit of assembly language. In this article, I will briefly introduce the main features of the MMX technology, explain how to detect whether an x86 microprocessor has built-in MMX capabilities and show how to program a simple image processing application.

The assembly language code presented here uses NASM, the Netwide Assembler. NASM employs the standard Intel syntax instead of the AT&T syntax used on many popular UNIX assemblers, such as GAS.

MMX Programming Environment

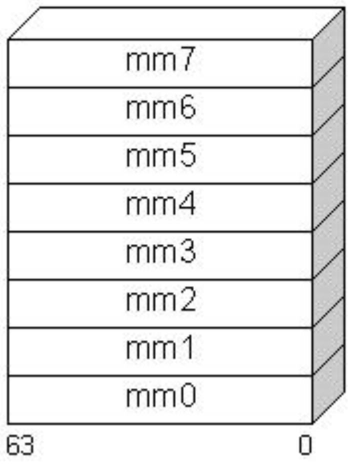


Figure 1. MMX Register Set

The MMX technology extends the Intel architecture by adding eight 64-bit registers and 57 instructions. The new registers are named MM0 to MM7 (see Figure 1). Depending on which instructions we use, each register may be interpreted as one 64-bit quadword, two packed 32-bit double words, four packed 16-bit words, or eight packed 8-bit bytes (see Figure 2).

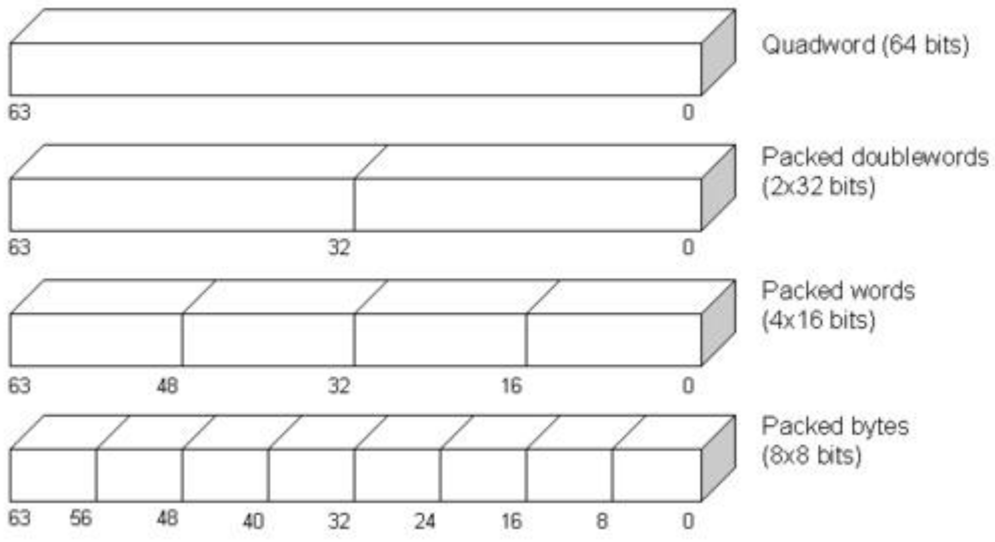


Figure 2. MMX Data Types

The MMX instruction set comprises several categories of instructions, including those for arithmetic, logical, comparison, conversion and data transfer operations.

The syntax for MMX instructions is similar to other x86 instructions:

```
OP Destination, Source
```

This line is interpreted as:

```
  Destination = Destination OP Source
```

Except for the data transfer instructions, the destination operand must always be any MMX register. The source operand can be a datum stored in a memory location or in an MMX register. A few specific MMX instructions will be discussed further on.

Detecting MMX Processors

Before running a program that uses MMX instructions, it is important to make sure your microprocessor actually has MMX support. Your Linux system should be an Intel x86 or compatible microprocessor (386, 486, Pentium, Pentium Pro, Pentium II, or any of the Cyrix or AMD clones). This is easily checked by executing the **uname -m** command. This command should return i386, i486, i586 or i686. If it does not, your Linux system runs on a non-x86 architecture.

In order to determine if your CPU supports MMX technology, use the assembly language CPUID instruction. This instruction reveals important processor information, such as its vendor, family, model and cache information. Unfortunately, the CPUID instruction is present only on some late 80486 processors and above. So, how do you know if CPUID is available on your system? Intel documents the following trick: if your program can modify bit 21 of the EFLAGS register, then the CPUID instruction is available; otherwise, you are working with an aged CPU. See Listing 1 (lines 12-29) to learn how this can be done.

Listing 1.

Next, request CPU feature information by putting a value of 1 in the EAX register and executing the instruction. The resulting value is returned in bit 23 of the EDX register. If this bit is on, the processor supports the MMX instruction set; otherwise, it does not. Listing 1 (lines 43-50) shows how to do this.

Programs should contain two versions of the same routine: one using MMX technology and one using regular scalar code. At runtime, the program can decide which routine it should actually call.

If MMX instructions are executed in a system that does not support them, the CPU will raise an “invalid opcode exception” (interrupt vector number 6) which is trapped by the Linux kernel. The Linux kernel in turn sends an “illegal instruction signal” (code number 4) to the offending process. By default, this action terminates the program and generates a core file.

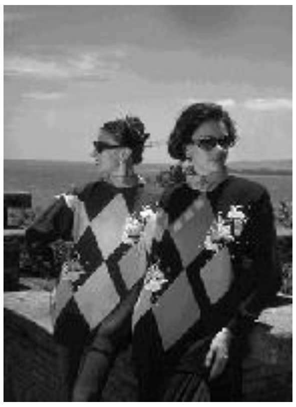


Figure 3. Original Gray-Scale Image

An Image Brightening Program

Let's suppose we have a gray-scale bitmap image, like the one in Figure 3. Each pixel is stored in one unsigned 8-bit byte contained in an array. Smaller numbers represent darker tones of gray, while larger numbers represent brighter tones. Numbers 0 and 255 represent the pure black and white colors, respectively. For the sake of code simplicity, the images employed in this program (see Listing 2 in the archive file) use Microsoft Windows' gray-scale BMP file format. John Bradley's **xv** utility can easily be used under Linux to create and display this kind of bitmap image.

To make the image brighter, we just need to add a positive integer (let's say 64 hexadecimal) to each of its pixels. In C, we would have something like this:

```
#define BRIGHTENING_CONSTANT 0x64
unsigned char bitmap[BITMAP_SIZE];
size_t i;
/* Load image somehow ... */
for(i = 0; i < BITMAP_SIZE; i ++)
  bitmap[i] += BRIGHTENING_CONSTANT;
```

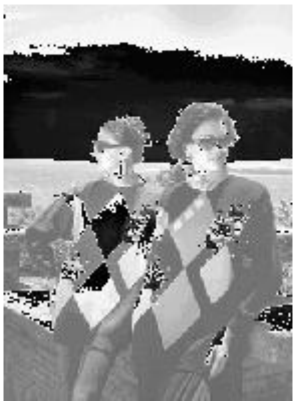


Figure 4. Brightened Image Using Wraparound Arithmetic

Unfortunately, we end up with the undesired image found in Figure 4. This happens because of wraparound; if the result of the addition overflows (i.e., exceeds 255, which is the upper unsigned 8-bit byte limit), the result is truncated so that only the lower (least significant) bits are considered. For example, adding 100 (64 hexadecimal) to a pixel value of 250 (almost pure white) gives the result shown below.

```
      250 decimal       11111010 binary
    + 100 decimal    +  01100100 binary
    -------------    -----------------
    = 350 decimal    = 101011110 binary   Overflow
                                          produced
    =  94 decimal    =  01011110 binary   Take the 8
                                 least significant bits
```

The result is 94 which produces a darker gray instead of a brighter one, causing the observable inversion effect.

What we require is that whenever an addition exceeds the maximum limit, the result should *saturate* (clipped to a predefined data-range limit). In this case, the saturation value is 255, which represents pure white. The following C fragment takes care of saturation:

```
   int sum;
   for(i = 0; i < BITMAP_SIZE; i ++)
     {
     sum = bitmap[i] + BRIGHTENING_CONSTANT;
     /* UCHAR_MAX is defined in <limits.h>
      * and is equal to 255u */
     if(sum > UCHAR_MAX)
       bitmap[i] = UCHAR_MAX;
     else
       bitmap[i] = (unsigned char) sum;
     }
```

Now we obtain the image shown in Figure 5, which is brightened as we wanted.

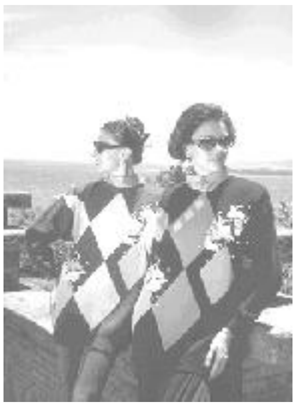


Figure 5. Brightened Image Using Saturation Arithmetic

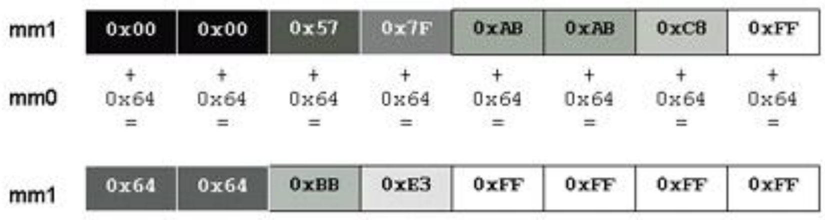


Figure 6. Unsigned Byte-Packed Addition with Saturation

MMX technology allows us to do this saturated arithmetic addition on eight unsigned bytes in parallel using just one instruction: **paddusb**. Figure 6 shows an example of how this instruction works. Our image-brightening algorithm (see Listing 1, starting at line 61) can be described as follows:

- Pack the same brightening constant byte eight times into the MM0 register (line 66).
- Repeat bitmap-size / 8 times:
  1. Copy the next eight bytes from the bitmap array into the MM1 register (line 74).
  2. Add the eight packed unsigned bytes contained in MM0 to the eight packed unsigned bytes in MM1. Use saturation (line 75).
  3. Copy the result of the MM1 register back to the bitmap array from where it was originally taken (line 76).
  4. Advance bitmap array index register (line 77).

The **movq** MMX instruction used in steps 1 and 3 copies 64 bits from the source operand to the destination operand.

Whenever we finish executing MMX instructions, the **emms** instruction (Listing 1, line 81) should be used to clear the MMX state. This is an important issue, especially if any floating-point instructions follow in our program. In order to make the MMX technology compatible with existing operating systems and applications, Intel engineers decided the MMX registers should share the same physical space with the floating-point registers. This was considered necessary because, for example, in a multi-tasking operating system such as Linux, whenever a task switch occurs, the running process must have its state preserved in order to be resumed some time in the future. This state preservation involves copying all of the CPU's registers into memory. If you add more registers to the CPU, you must also modify the operating system code that takes care of saving the registers. However, if your new registers are aliased to existing registers, no change is required in the code.

Unfortunately, this workaround in the case of MMX and floating-point registers has a major drawback: you cannot use both types of registers at the same time, simply because they represent two very different types of data. The general rule is you cannot mix MMX and floating-point instructions in the same portions of code. Therefore, the **emms** instruction is the mechanism of informing the CPU that future floating-point instructions are allowed in the program.

Conclusion

Is all this worth the trouble? The answer to this question depends on the importance you give to speed. Comparing the MMX example program to a pure C language version, the speed improvements speak for themselves. The MMX routine is roughly 14 times faster than the C version (Listing 2 in the archive file) when compiled with no optimizations and about five times faster when full **-O2** optimizations are enabled. Of course, you will lose portability and will probably have a harder time writing and debugging assembly language code. Life's full of tough choices, isn't it?

Resources

Ariel Ortiz Ramirez is a faculty member in the Computer Science Department of the Monterey Institute of Technology and Higher Education, Campus Estado de Mexico. He has been using Linux to teach x86 assembly language for two semesters now. Although he has taught several different programming languages for almost a decade, he personally has much more fun when programming in Scheme. He can be reached at aortiz@campus.cem.itesm.mx.

Archive Index Issue Table of Contents

Advanced search

Troll Tech's QPL

Craig Knudsen

Issue #61, May 1999

A look at the new Qt public license and the effects it may have on software development for KDE and GNOME.

Troll Tech announced in November that its upcoming Qt 2.0 Free Edition GUI toolkit will have a more open license. Qt is best known in the Linux community as the GUI toolkit used to develop KDE (K Desktop Environment), a UNIX desktop environment. Qt Free Edition 1.X for UNIX is currently free for non-commercial use—if you want to sell your software, you need to purchase Qt Professional Edition, which starts at over $1000 for a single-user license.

The new licensing terms apply to the upcoming version 2.0 of Qt, currently in beta release, and is considered “open source”. Troll Tech has dubbed its new license the “Q Public License” or “QPL”. How does this new license differ from the old one? The license for Qt Free Edition 1.X does not allow developers to redistribute modified versions of the Qt library. Some argue that the Qt Free Edition 1.X license can delay projects that require either fixes or enhancements to the Qt toolkit. By allowing developers to distribute modified versions of Qt, the new license overcomes this problem.

KDE

The KDE project was started at the end of 1996. The developers chose the Qt library over other toolkits such as Xforms and Motif because of its documentation, its look and feel and because they preferred using C++ (Qt) over C (Xforms and Motif). The new QPL will have a positive effect on KDE development and will most likely attract more developers to the project. When Qt 2.0 Free Edition is released, KDE will have the option of modifying Qt for use with KDE and will thus be able to produce more frequent releases.

Harmony

The Harmony project was started to create an open source replacement for Qt to be used with KDE, allowing KDE to become a part of completely free operating systems. For example, according to the “Debian Free Software Guidelines” (DFSG), Qt's existing license prevents it from being included in Debian Linux. KDE meets the DFSG requirements but requires Qt to run. Harmony's license meets the DFSG restrictions allowing it (and KDE) to be included in the Debian Linux distribution instead of Qt. The developers had also planned on making Harmony an improvement over Qt by adding new features such as multi-threading and themes which are included in Qt 2.0. Although the new QPL is not as open as Harmony's GNU Library General Public License (LGPL), it caused developers to lose interest, and the Harmony Project was shut down in late January.

GNOME

The GNU Network Object Model Environment (GNOME) project was announced in August 1997. GNOME is built with GTK+, a GUI toolkit originally developed as part of the popular GIMP image tool. There have been many heated debates over the licensing differences between GNOME (GTK+) and KDE (Qt). GTK+ uses the LGPL license, while Qt 1.X has a more restricted license. These issues were behind the initiation of the GNOME project. Now that licensing for KDE/Qt is becoming more open, GNOME's destiny as the desktop for free operating systems might be a little less secure. KDE clearly has a head start, having released KDE 1.1 in February while GNOME 1.0 was announced in March at the LinuxWorld Conference. GNOME, unlike the Harmony project, does have corporate support. Red Hat's Advanced Development Laboratories has a handful of people developing GTK+ and GNOME and has been very committed to the GNOME project.

Linux Distributions

Caldera became an early adopter of KDE by including it in OpenLinux 1.3 in September 1998, and plans to make it the default desktop for OpenLinux 2.0. Red Hat intends to use GNOME 1.0 for its default desktop and continues to use FVWM as its window manager in the meantime. Red Hat has made KDE available from its “Raw Hide” site which distributes developer releases, and will consider putting KDE into its main distribution when Qt 2.0 Free Edition and the corresponding version of KDE are available. Debian currently distributes KDE and Qt on their “non-free contrib” CD, but not the main distribution because it does not conform to the DFSG. It appears that the new QPL license will allow Debian to include KDE in their main distribution.

Summary

The full effect of Troll Tech's new QPL will not be known for quite some time. We'll need to wait for Troll Tech to release Qt 2.0 Free Edition and then for a new version of KDE based on Qt 2.0. Clearly, it will be a positive change for KDE, allowing it to be included in more Linux distributions. The effect on GNOME is less clear, but the QPL announcement does not appear to have affected GNOME development.

Resources

Craig Knudsen (cknudsen@radix.net) lives in Fairfax, VA and telecommutes full-time as a web engineer for ePresence, Inc. of Red Bank, NJ. Craig has been using Linux for both work and play for three years. When he's not working, he and his wife Kim relax with their two Yorkies, Buster and Baloo.

Archive Index Issue Table of Contents

Advanced search

Creat: An Embedded Systems Project

Nick Bailey

Issue #61, May 1999

Creat is a tool set for teaching embedded systems. In designing it, Mr. Bailey wanted it to be useful for real problems, cheap enough to build on the pittance which is an undergraduate's project budget, and totally open and accessible to the curious.

Creat stands for Combined Resource Embedded Application Toolkit. It is a collection of tools pulled together from the Internet to permit Linux users (and, in the future, any UNIX users) to construct simple projects based on the Motorola MC68HC811 8-bit microcontroller. For the hardware part of the project, the idea was to provide a general microcontroller for students who wanted to have a small lump of computing power in their final-year projects; for those who wanted to specialize in applied microcomputing, to study the innards of the project. I wanted it to be accessible enough from “both ends”--programming and hardware—so that an expert in one would benefit from experience with the other. The whole project turned out to be a positive experience and a lesson in the benefits of cooperation and open software.

In choosing hardware for the exercise, I would have liked to obtain an up-market 16-bit microcontroller with Linux ported to it. This would have given seamless integration between host machine and target platforms, but even now, the cost of such a project is prohibitive in both cash and development time. The monster thus created would probably have been a significant overkill for the target application areas. At the very low end, the Linux community already has a range of useful utilities aimed at the PIC microprocessor (see “PIC Programming with Linux” by Brian C. Lane, *Linux Journal*, October 1998), which is a useful chip for replacing large quantities of logic with a single package. More ambitious projects might make use of a microcontroller port of Linux itself (see the Linux/Microcontroller Home Page by D. Jeff Dionne, http://ryeham.ee.ryerson.ca/uCinux/). My target was those projects in between: more processing power than you need to count events and run a multiplexed LED display and less than you need to run X. Our typical projects had an LCD display, a keyboard and some custom electronics to handle the project-specific I/O. The handling of interrupts might be important, together with enough flexibility to store a reasonable amount of data. The projects need to be highly testable and modifiable, but in the interests of economy, special hardware adapters and programmers were to be avoided.

Putting Together a Solution

With limited time and money, the clear way forward was to trawl the Net. More than just the hardware must be considered to build a useful system. To compete with the large and expensive kits in the marketplace, I would need an in-circuit emulator, a compiler/assembler and some way of downloading the program and booting the target board. For the benefit of the hard-line computer scientist, in-circuit emulators are expensive devices which plug into the microcontroller socket at one end and the workstation at the other. They do all the things done by a decent IDE, but can also ensure the hardware is behaving by monitoring bus control signals and the like. For microcontrollers, students are encouraged to plug in a logic analyzer instead: rather like an oscilloscope with an enormous number of channels and triggering, which can be locked to a particular data value and hence to the execution of a particular instruction. You don't get to see a stack trace or register contents, but you can examine exactly what is going on in terms of logic levels.

At the University of Leeds, all Electronic Engineering graduates are familiar with C. Those who specialize in computer subjects will also probably have picked up some parsing, X Window System applications programming, Java or C++ and Occam. The major requirement is to provide them with “right-first-time” prototyping tools, so that they can debug programs on workstations and get them “shipped” with as little ado as possible. The environment within which they are performing their project actually makes quite a good analogy to a commercial one: too little money and too big a time pressure to craft the most beautiful and elegant system imaginable. One thing the School of Electronic and Electrical Engineering doesn't have is the facility to make plated-through PCBs (polyclorobenzine circuit boards). It is too expensive to run because of severe environmental problems associated with the technology. It is possible for students to produce single or double-sided copper boards by photo lithography and etching (and they have), but plating through and the production of microcontroller boards is truly out of the question. Thus, we need to make boards for general application. We have them manufactured externally on large panels and cut up, so they can be used as the brain of an electrically more simple project built on a circuit board manufactured in-house.

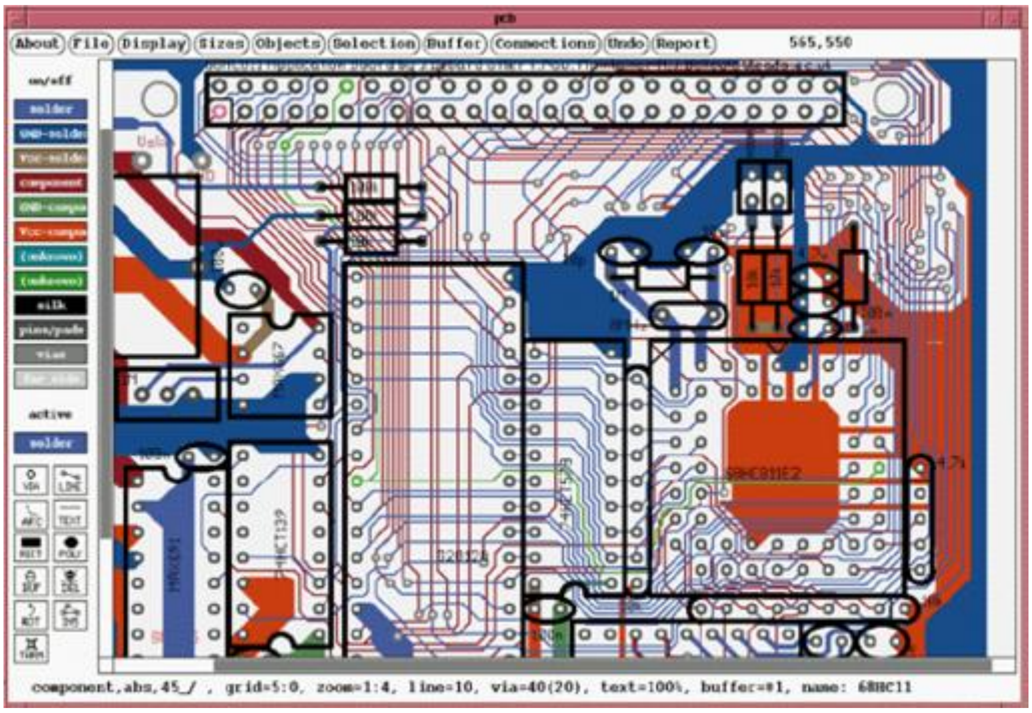
The Creat specification required the following:

- more powerful than a PIC with large data storage capacity

- a development environment which is open to C users
- very inexpensive to construct
- flexible in application area
- no special hardware for programming or servicing
- target system simulation to aid debugging

Cooperation vs. Competition

Having produced a requirements spec, it was quite obvious I would not have time to write all of the necessary code and design the hardware from scratch. The normal scenario would be to obtain a loan (or course development grant), hire staff and have them reinvent the wheel by building yet-another-microcontroller-kit, then persuade the University to try and market it. Of course, this would have produced all the usual arguments against releasing the software source, and the cost would not be much less than $100,000 US by the time everything had been shaken down, even for a simple system. The openness of the project would be compromised; distribution, maintenance and servicing issues would arise.



component,abs,45_/ , grid=5:0, zoom=1:4, line=10, via=40(20), text=100%, buffer=#1, name: 68HC11

Figure 1. PCB with Copper Trace Highlighted

Fortunately, as a long-time Linux user in an EE department, I had already come across Thomas Nau's PCB program. (Source code is available from ftp://ftp.uni-ulmde/pub/pcb/, also available as a Red Hat RPM or as a Debian package. See the README file for more details.)

PCB is a drawing package, with many useful debugging aids including net list import and export, but no auto route or schematic capture capability. It is ideal for students intricately hand-drafting a single-sided PCB—a skill not yet superseded by at least the majority of commercial packages. Maintenance of the package has been taken over by Harry Eaton, who looks after such features as Gerber output (the file format popular amongst PCB manufacturers). Figure 1 shows this program in action displaying part of the Creat CPU board. The really good thing about PCB is that *it comes with a microcontroller circuit layout* as a demonstration file. This is one of Thomas's projects, and I couldn't believe my luck, because it had all the attributes I needed for Creat. In fact, because Thomas's board was a full-blown stand-alone application, it was rather too complex for the tasks we needed, so I set about hacking it down to size.

The second stroke of luck came when I received an e-mail from Jerome Debard, a graduate of Toulouse. His degree was in Engineering, he wanted to gain some work experience and language skills by working in England, and was prepared to do so for free! He took the idea of reducing Thomas's board, and in a few weeks of frenzied activity, got it working, including writing a run-time library for it. (He also did the cooking at the Annual Communications Group Rooftop Barbecue, which is the first time that occasion has ever benefitted from having a French chef.)

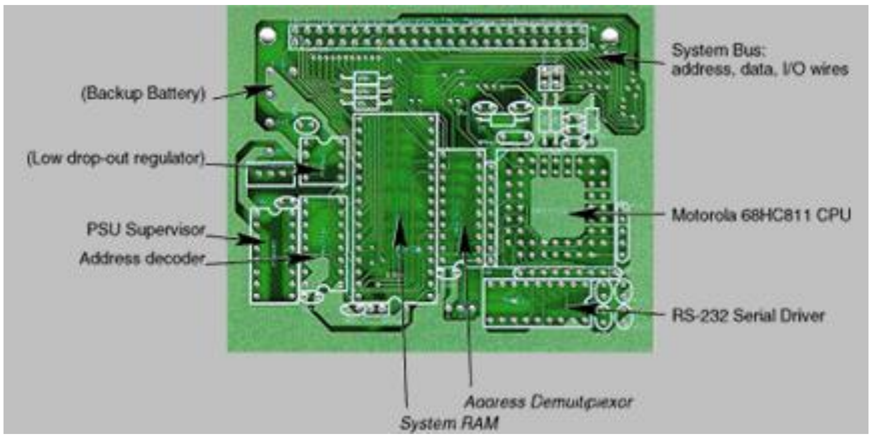


Figure 2. The Creat CPU board

Aside from getting the circuit *mis au point*, the hardware was basically wrapped up. Figure 2 shows the bare board after commercial manufacture: its actual size is about 3.5 inches square with all the components added. To make the board work, you have to solder up the components labeled in plain text. If you want it to work without a regulated bench supply, you will need the components labeled in parentheses too. This gives you a microcontroller module with 2KB of EEPROM, 256KB of RAM, 28 digital and 8 analogue I/O lines. This is useful for some projects, but not really enough for C. Adding the components labeled in italics gives you up to 128KB of RAM, but leaves only five I/O lines. Fortunately, Motorola makes a rather useful PRU (port replacement unit) which can be hung on the address and data bus, transparently giving you these lines back. Figure 3 shows the memory map of the system with 128KB fitted. (Motorola, Inc.

produces a comprehensive reference manual for this processor family, ref. no. MC68HC11RM/AD REV 3: Motorola Literature Distribution, P.O. Box 5405, Denver, Colorado 80217

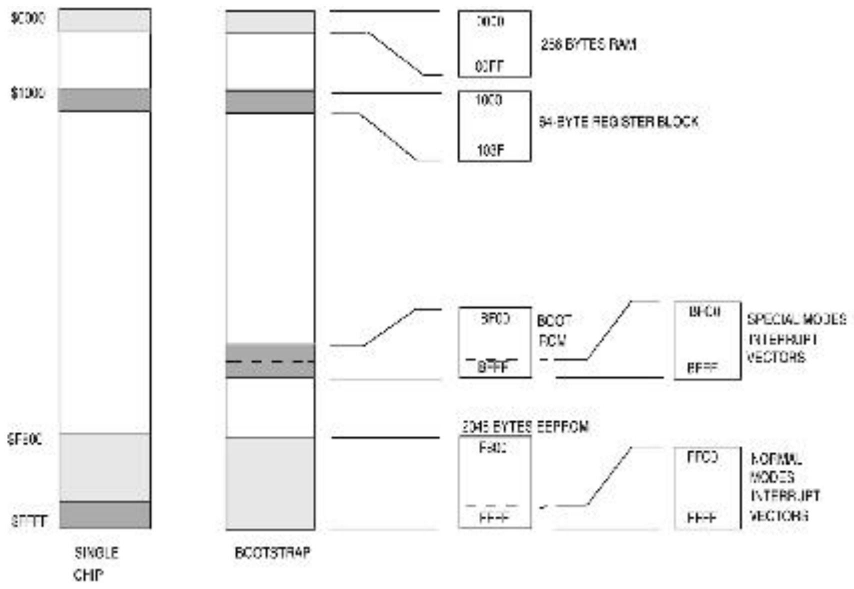


Figure 3. System Memory Map

A Low-Cost Software Development Philosophy

With essentially reliable hardware, the work now begins on a development system and first of all on the C compiler. Back to the Internet and to the aforementioned Donald Jeff Dionne, in whose public FTP space I found a binary distribution of exactly the compiler I needed. Based on the **gcc** port by Coactive (http://www.coactive.com/), it was ideal. A common source-code base could be built to run on the Linux boxes and then cross-compiled for the Creat board. The binary was in the old a.out format, and I e-mailed Jeff to find out whether it had been developed.

gcc-hc11 was the C compiler Jerome had looked at, then rushed out a run-time library including memory allocation and rudimentary string I/O routines. At the time, there were two European Community exchange students from Valencia in the department, so based on the availability of a working gcc, we started to cook up something which would be useful and free.

It is often efficient to divide a microcontroller-based project into two, testing the hardware and software independently before using them together. Of course, it is important for the two tasks to advise each other, but if the project is large, it is almost certain a hardware group will be working on the project concurrently with a software group. Creat seeks to make that process as easy as possible through a strict process of hardware abstraction. When somebody builds a piece of hardware which might be of general use, they make a package of it. A package consists of three things: a PCB layout in pcb format with the same form factor as the main CPU board, a set of subroutines (which might be written in C or assembler) with a specific C-language interface, and another set of C subroutines taking the same arguments which compile and run on the workstation.

The key idea is that any program which uses a particular piece of hardware, e.g., a dot-matrix LCD display, can be written using the Creat LCD device interface. This provides two main calls: an initialisation call invoked before the device can be used and another to write a character on the LCD. Creat's **make** system can be used to build the application for the Linux box by issuing the commands:

```
make depend; make wkstn
```

After testing, code can be compiled for the 6811 as well using:

```
make 6811; make boot
```

What's in a Package?

Daniel Roques Escolar and Alberto Ramos Fernandez worked long and hard on the coding for two packages: one to drive a dot-matrix LCD and the other to drive a row/column-scanned keyboard. We were inspired by a previous article (“Using Tcl and Tk from Your C Programs” by Matt Welsh, *Linux Journal*, February 1995) which showed how Tcl/TK scripts can be executed by forking off a new process and then executing a copy of **wish**. Connecting wish through a pipe to the parent process means commands can be sent to it and information received back. This makes it very easy to present a simulated LCD as a wish X window, and much can be achieved without having to write any X code at all. Even though Xlib-based code would execute much quicker than a wish script, it shouldn't be a problem since it is probably a case of a 586 of some sort versus an 8-bit microcontroller.

Thanks to Danny and Alberto, at least two additional modules are now available which plug into the Creat bus: the LCD and keyboard modules. These packages provide the hardware abstraction interface and use a forked wish script to emulate the LCD and the keyboard on-screen.

There is one problem area in which microcontrollers tend to outperform even Linux—interrupt handling. Microcontrollers come loaded with I/O facility, so a huge number of interrupt sources exist. There are even timers which don't have to connect with the asynchronous outside world, but just sit there generating interrupts. These and other troublemakers present a serious problem for simulation on a Linux platform. The programmer writing the hardware abstraction code might want to generate a handful of different interrupts, but the only ones present for the purpose of simulation are SIGUSR1 and SIGUSR2. A single timer can produce that many. We therefore devised a method for the simulation of multiple interrupt sources.

When there is a conflict of interests in hardware simulation, it is important to place the strain on the Linux side of things rather than on the microcontroller. The interrupt handling method is basically transparent as far as the 68HC811 is concerned. It amounts to the insertion of a vector in an interrupt table in response to a “register interrupt” subroutine call. At the Linux end, things are more complicated by the fact that all processes which generate interrupts are told to generate the same interrupt. In addition, each one shares a pipe with the parent process, down which they write a byte before raising SIGUSR1. When the handler gets called, it uses **select** against all pipes to establish which process needs servicing, then looks in its own table of service routines and calls the appropriate one. **Listing 1** shows the code which registers an interrupt process.

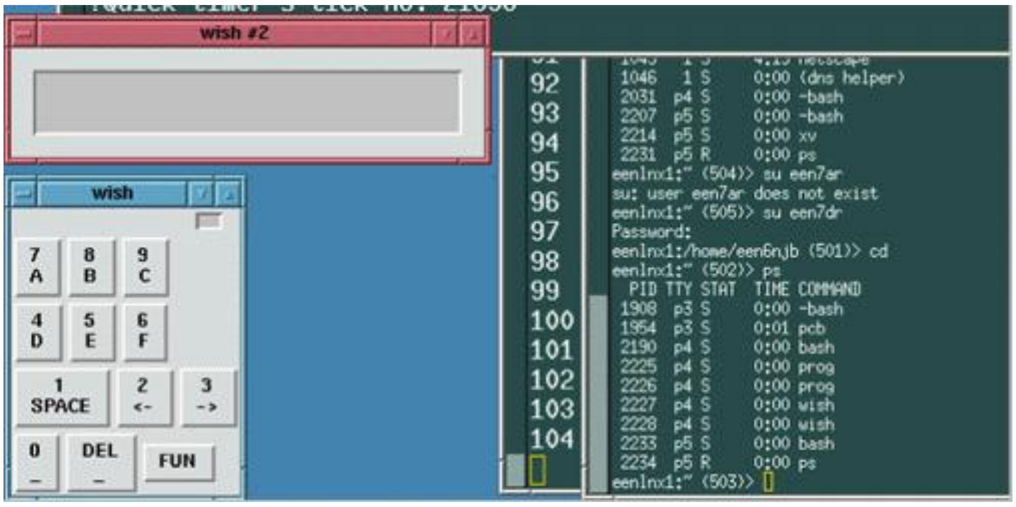
With the registration process available, it is possible to create packages that cause interrupts to take place. The simplest example is the timer device, which generates interrupts at a regular interval. The routine to create a timer device is **init_timer** and in the case of the microcontroller, it would only have to set the appropriate timer interrupt vector and start the hardware timer in free-running mode. When compiled on the workstation, the function registers an interrupt source, then forks off a function which spends most of its time sleeping, awakening now and then to raise a signal. The code appears in **Listing 2.**

Whether the interrupt source is a timer or some other simulated I/O device, the same interrupt service routine, **sigusr1_handler** (**Listing 4**), is called each time a signal is raised. The signal handler has access to the list of registered processes, so it can call **scanlist** (**Listing 3)** to catch the culprit and execute an associated interrupt service routine. The whole thing acts like a kind of interrupt multiplexor, so that with the help of a list of pipes, the appropriate source can be associated with the appropriate service routine, even though all possible sources raise the same signal.

What Have We Done; What Have We Learned?

In actuality, all I have done is manage a project. The project had some excellent resources and my input was to point them in what I feel is an educationally useful direction. I hope “real” applications (whatever they might be) will find this package useful too. The whole bundle is available in a package on my web site (http://www.ee.leeds.ac.uk/homes/NJB/Software/linux-stuff.html#creat) and you can browse the manual written by Danny and Alberto on-line. Thomas's PCB, as modified by Jerome and others, is also there, along with the port replacement unit PCB and all other software mentioned above. The manual contains over 130 pages.

I hope people will be encouraged to contribute packages to this project. The modules available so far are not entirely complete. I need to tidy up the interrupt registry to make it truly modular and make the vector insertion work on the microcontroller. Issues with the LCD module need resolving, such as the display size being fixed in the Tcl/Tk script; and a new version of gcc6811 needs building, using an up-to-date gcc as the base and glibc. However, on the positive side, among the things which have delayed my fixing these problems is that the whole system does actually seem to work together rather well. Figure 4 shows the simulated keyboard and LCD, with a simple counter ticking away in the background.



```
92
93
94
95
96
97
98
99
100
101
102
103
104
```

```
 1046   1 S     0:00 (dns helper)
 2031  p4 S     0:00 -bash
 2207  p5 S     0:00 -bash
 2214  p5 S     0:00 xv
 2231  p5 R     0:00 ps
eenlnx1:~ (504)> su een7ar
su: user een7ar does not exist
eenlnx1:~ (505)> su een7dr
Password:
eenlnx1:/home/een6njb (501)> cd
eenlnx1:~ (502)> ps
  PID TTY STAT  TIME COMMAND
 1908  p3 S     0:00 -bash
 1954  p3 S     0:01 pcb
 2190  p4 S     0:00 bash
 2225  p4 S     0:00 prog
 2226  p4 S     0:00 prog
 2227  p4 S     0:00 wish
 2228  p4 S     0:00 wish
 2233  p5 S     0:00 bash
 2234  p5 R     0:00 ps
eenlnx1:~ (503)>
```

Figure 4. The Workstation Simulating a Simple Configuration

Acknowledgments

Nick Bailey (n.j.bailey@leeds.ac.uk) obtained a B.S. in Computing and Electronics from the University of Durham. Having worked at British Telecom Applied Technology in West London, he returned to Durham to study for a Ph.D. in the application of parallel computing to audio signal synthesis. He is currently a lecturer at the University of Leeds in Applied Computer Systems at the Department of Electronic and Electrical Engineering, with additional responsibilities for Overseas and European Liaison. He enjoys old, unreliable fast cars and owns a cello, but demonstrates no discernible talent in those directions.

Archive Index Issue Table of Contents

Advanced search

Upgrading Linux Over the Internet

Daniel Dee

Dale Nielsen

Issue #61, May 1999

A real life experience in remote upgrading of a Linux PC across the Pacific Ocean.

As a business offering software internationalization services, we operate a small office in western Massachusetts as well as a small sister company in Taipei, Taiwan. We also need to support a distributed software development environment for engineers working remotely. While our bandwidth demands are not great, we do need reliable e-mail, web, news and FTP services. Primarily used to provide connectivity from the inside office to the Internet, the connection has to be available for external access from remote users on a 24 by 7 basis.

The network consists of a variety of UNIX workstations and PCs running Windows 95 and Windows NT, used for software development and support as well as the usual office applications. We use Linux running on Intel-based PCs as our network servers because it provides one of the most cost-effective small business network server solutions. The network uses a private class C Internet address from the 192.168.*.* block, since it is not directly connected to the Internet.

Our internal Linux server, with a Pentium 133MHz processor, an Adaptec 2940 SCSI card, a bunch of SCSI drives and a 4mm DAT tape drive, provides the backbone of our computing infrastructure. As a mail hub, it provides POP3 and SMTP support for mail-client applications running on the network. By running Samba, it acts a network file server for Windows-based PCs. Finally, it provides name resolution services using BIND.

The second Linux server in this “dynamic duo”, an old 486 PC with a 500MB hard disk and a monochrome monitor, is our external gateway machine. It connects us to the Internet through a persistent PPP connection with a static IP address over a 28.8K dial-up phone line to a local Internet service provider. This machine also acts as a dial-in server and as a firewall. It provides an e-mail relay and spam filter to and from the internal mail hub. HTTP, FTP and NNTP proxy services are also provided by this machine to allow internal users access to these Internet resources.

Both Linux machines were running Debian version 1.3. On an Internet firewall machine, you want to have precise control over what software is loaded on the machine. You want the minimum necessary to do the job, no more. Since the machine was to be remotely administered, it was even more important that it be easy to upgrade individual packages as necessary without having to do cold installs for new OS versions. Debian's **dselect/dpkg** system of handling software packages is ideal in this situation. We could easily select the software required to run the system, knowing that all prerequisite packages were included. Plus, Debian's large collection of software packages included almost everything we needed in its convenient dpkg format.

Debian Linux can be downloaded for free from http://www.debian.org/ or a host of mirror sites. In our case, we purchased a CD-ROM from Linux Software Labs, which also made it easy to add a contribution to the Debian project, whose work we greatly appreciate.

The Taipei office used a Linux gateway to connect to the Internet, but the configuration was quite different. We were issued a block of class C addresses from the Taiwanese ISP which advertised a route to them. The gateway machine was running a publicly accessible FTP server, HTTP server and mail hub, as well as being the primary public name server for our domain in Taiwan —all using a very old version of Caldera Linux.

When the operating systems on the server machines in the U.S. office were upgraded to take advantage of the new features in Linux 2.0, it seemed an ideal time to upgrade the systems in Taiwan, as well as reconfigure the network to more closely match the one in the U.S. office.

While the project seemed straightforward enough, the problem was that the work had to be done from ten thousand miles away across the Pacific Ocean using the Internet.

Planning

Upgrading Linux boxes remotely, especially across the ocean, requires some advance planning. Some of the issues we had to deal with were:

- Which Linux to use?

- What were the security concerns?
- Were we going to set up both a private and public side network?

Our choice for the first question was to stay with Debian Linux version 1.3. It was the same version we were running in Massachusetts, so we could essentially install a copy of what was on the U.S. system, reconfigure it for the different names and addresses in Taiwan, and be all set.

Since the upgrade was to be done across the Internet, security was a major concern. We needed a secure connection from the U.S. to Taiwan so that logins and passwords would not be revealed to Internet eavesdroppers and Ethernet sniffers; thus, we chose the Secure Shell (SSH) package. Due to U.S. export restrictions, we could not just upload the software from Massachusetts, so we downloaded the source for the SSH package from a Taiwanese FTP site to the Linux machine in the Taipei office. We then compiled and installed it, so the install/upgrade could proceed in a secure fashion.

While our U.S. setup is required to service only an internal network, our Taiwanese operation decided they needed to set up an area to allow public Web and FTP access. To do this without compromising security for the internal network, things had to be set up a bit differently.

Taiwan's block of Class C addresses, assigned by their ISP, were used by both the internal machines and the firewall. We designed a network setup including a publicly accessible network created using these addresses for use by the public HTTP and FTP servers. The rest of the machines were connected to a private network, once again using addresses from the 192.168.*.* block as in the U.S. office. The firewall machine was then configured with a second Ethernet interface: one to connect the outside PPP connection to the publicly available network and the other to connect the private network. We then used the IP firewalling capabilities of the Linux kernel to keep network traffic where it belonged.

Hardware Preparation

The Taiwan office already had an operational gateway PC named “dragon”. Rather than upgrading it while using it to provide our connection, a second machine, “dolphin” was identified as the new gateway machine. This way, we could be sure the upgrade was successful before putting it in place, and it gave us a fall-back position in case it was not. Since the name and address of dragon were in DNS maps outside of our control, and coordination with the local ISP had proved inconvenient in the past, we had to swap the identities of the machines before proceeding.

As the new dragon would be serving both public and private networks, two network cards were installed. Simple jumper-capable NE-2000 compatible cards were chosen so that their IRQs could be easily configured. In order for our system administrator to log in to dolphin through the Internet, a minimal Caldera Linux system was installed on it. Finally, dolphin was connected to the local network.

Since the new firewall machine was no longer going to act as a mail hub for the network, an existing server running Linux, “elephant”, was nominated. Sendmail and a POP3 server were installed on elephant. Dragon was reconfigured to relay e-mail in and out of the domain rather than acting as a hub. Elephant was also configured to act as the DNS server for the internal network, with dragon as a forwarder, since elephant would no longer be directly connected to the Internet. In turn, dragon was configured to continue acting as primary DNS server for the domain to the outside world while using elephant as its resolver. This way, only publically accessible machine names and addresses would be visible from the Internet, while dragon would continue to be able to resolve all internal addresses, both public and private.

Coordination

Two concerns arise when doing remote upgrades:

- Disruption of Internet access must be avoided as much as possible.
- A human being must be present to act as a remote pair of hands in the unlikely event that the new machine was hung or rendered inaccessible or unbootable as the upgrade proceeded.

To avoid disruption, we decided that the upgrade should be done during the weekend in Taiwan. Since a time zone difference of exactly 12 hours exists between Massachusetts and Taipei, it was agreed that the upgrade would start on Friday at 8 PM EST, or 8 AM Saturday in Taiwan. A human would not have to be at the office in Taiwan until 9 AM when the machine was ready to be rebooted.

In advance of all this, gzipped tar files of the root, /usr and /var file systems from the Massachusetts machine were downloaded via FTP to the Taiwan office Friday night Taipei time. The exercise of downloading, building and installing SSH was also accomplished at this time.

Communication between the upgrader in the U.S. and the human sentinel in Taiwan was necessary. To avoid making expensive long-distance telephone calls (although we still ran up a $200+ telephone bill) unless it was necessary, we decided to use computer communication whenever possible. Latency eliminated e-mail as a possible choice. We chose to use talk when it worked and write otherwise.

We started by adding partitions to the disk of the target machine. Three new partitions were created with fdisk in order to hold the new root, /usr and /var file systems. Next, a reboot was needed in order to ensure the new label was in force so that new file systems could be created and the tar files restored. We used **rdev** to set the new root device in the kernel so that it would be ready to boot the freshly installed operating system. Then we needed to localize the machine, changing the name and address of the machine to match the Taipei office network.

Sometime in the middle of this work, it was noon in Taipei. After sending a warning note to the upgrader in the U.S. that no human would be there for an hour to restart the machine in case of a foul up, the Taiwan staff headed off to lunch.

It took two more hours after the Taipei people came back from lunch before things became almost ready. The DNS maps were copied over from dragon so the machine would be ready to step right in as primary name server for the domain.

At that point, dolphin was rebooted into the newly installed system for the first time—all seemed well. It was also almost 3 AM the next morning in Massachusetts. We were now ready to hook up the new dragon to the Internet.

The first order of business was to switch the names and IP addresses of old and new dragon before performing the physical switchover. The files /etc/hosts, /etc/hostname and /etc/init.d/network all contain references to the hostname and IP addresses that needed to be changed. Once done, the modem was unplugged from old dragon and plugged into new dragon and it was time to go for the gold.

Problems

Dragon is connected to the ISP via a dedicated leased line. Its modem is designed for use on a 2/4-wire leased line circuit and is of the type that automatically connects to the ISP whenever the phone line is plugged in.

With bated breath, we waited for the new dragon to connect up. What we got instead were several screens of error messages. Dragon's modem has a large LCD display indicating that the modem was on-line, so the problem had to be in the configuration. It was 4 AM in Massachusetts.

We switched everything back to the way it was, so our upgrader could log in and find the problem. But we now realized that we must send our upgrader off to bed, as he was dozing off while typing. We decided to continue the upgrade the next morning, Taipei time.

Fortunately, it turned out that the problem was quite simple: we had not configured the routing table correctly. After fixing this, the new dragon was able to come up without a hitch and we were able to dispatch our upgrader to bed early that night.

Final Check

After our upgrader had gone to bed and we had the system up and running, it was time to make sure everyone's web browser and e-mail continued to work. Because the internal network is now on the private IP, the IP addresses of all internal UNIX and Windows computers had to be changed to 192.168.*.*. The web browsers also had to be reconfigured to look for the web proxy server on dragon's new private IP address. Finally, e-mail clients had to be reconfigured to look for the POP3 server from elephant, the new mail server.

As access to the internal network from the Internet is through the use of a one-time password, this particular system had to be checked. Finally, we also wanted to serve web pages from the public side of the network, so a plug was put into the firewall toolkit configuration to the Windows NT machine running IIS (Internet Information Server). For a while, the plug was not working reliably—that is, until we found out we had accidentally messed up the name table. With that fixed, we had all the pieces the Taipei office needed in working order.

Still in Progress

We eventually want to replace **fwtk** with IP masquerading. This makes the network more convenient to access from the inside network. We do have a test network that has it all working, so we will be deploying it shortly in the Massachusetts office. We want to be able to make public multiple web servers for corporate, testing and internal uses. These can be UNIX or Windows NT machines. The IP forwarding facility of the Linux kernel should make this fairly painless.

Acknowledgements

Daniel Dee (daniel@wigitek.com) has more than 10 years experience working in the development of GUI software toolkits, using X Version 10 and 11 and then Java since its inception. He is currently the president of Wigitek Corporation (www.wigitek.com/), a company providing software tools and consulting services for the development of Java-based dynamic graphic software.

Dale Nielsen (dale@wigitek.com) has a Bachelor of Science degree in Computer Science from the University of Massachusetts at Amherst and has been administering UNIX systems for over thirteen years and Linux systems for five. He provides system administration services for Wigitek Corporation and is the master planner behind the upgrade effort described in this article.

Archive Index Issue Table of Contents

Advanced search

Focus on Software

David A. Bandel

Issue #61, May 1999

QuickPlanning, tictactoe, gTick and more.

Things sure do move fast on the Internet. Who can keep up? Not I, that's for sure. Since I started this column a few months ago, many of the applications I've written about have been updated. This is always nice to see. Many packages I haven't highlighted have also been updated, so if you've downloaded something and like it, you might want to check back now and then for upgrades.

Many of these packages are maintained by one or a few authors. Drop them a line, be encouraging, mention breaks and problems, and suggest improvements (don't be surprised if they aren't implemented). Let them know you use the program and appreciate their work. Programmers always welcome bug reports, particularly detailed ones that make it easy to track and fix the error (be sure you can reproduce it). If you can code, offer help.

QuickPlanning: http://devplanet.fastethernet.net/files.html#QuickPlanning

QuickPlanning cannot truly be called a planner. It is more of a “tickler” to remind you what to do or what you did on a certain date. It built easily, but could use an INSTALL file containing a hint or two about what other things should be done. The first time I ran it and tried to save an entry, it aborted with a segmentation fault. A quick **strace** showed that it wanted to write to the directory $HOME/.qp/ which didn't exist. You must first create this file, until such a time as the author includes code to take care of it. Also, you must verify that any date you want to use actually exists. The program was happy to let me save data for 30 Feb 1999. For a program the author hacked together in 30 minutes, it works well. Libraries required are gtk-1.1.13, libXext.so.6, libX11.so.6, libm.so.6 and glibc.

tictactoe: http://www.forged.net/~blue/

For those who actually like to play tic-tac-toe, this version can be played over a network. It is a simple SVGA game that allows two players to bore each other to death for hours. (Does anyone over the age of three ever lose this game?) It will drive mathematicians in the crowd crazy since it uses row,column notation rather than an x,y coordinate notation. Lawyers (or those who just like to read licenses) will get a kick out of the license the author wrote for this program. Libraries required are Perl 5 and IO::Socket.

gTick: http://www.bsenet.cjb.net/gtick

Are you a musician looking for a cheap metronome? Well, turn on **gtick** and let it tick, tick, tick away to help you keep the beat. Or, turn it on to annoy everyone else in the room—after a while, they will ask "What *is* that noise?" A volume control is available as well as the choice of emphasizing timing beats: none, every other, every third or every fourth beat. Libraries required are gtk-1.1.13, libXext.so.6, libX11.so.6, libm.so.6 and glibc.

ganesha: http://www.frontiernet.net/~pani/downloads.html

ganesha displays the round-trip time and number of hops of multiple input sites. I think I need to turn it loose the next time I have a 10+ MB download to do over my 56K modem. A quick run-through showed me the sites I normally would favor are more hops and longer times away than a few sites in the opposite direction. Since I am on the west coast of the U.S., Japan is closer and faster than Europe. Now, if Japan has mirrored the big programs I want, I will get them there. **ganesha** is definitely a work in progress, but the author has already identified the shortcomings, so you should see some changes soon. Libraries required are gtk-1.1.13, libXext-6, libX11-6, libm-6, libpthread, libnsl, libdb-2, libgdbm-1, libcrypt and glibc.

Keystone: http://www.stonekeep.com/keystone.php3

Keystone is another job-tracking application. Similar to the program MOT, it has some of the same features and a few differences. Which program is better is a matter of taste. They accomplish much the same goal, just in a different manner. I found Keystone a bit more difficult to set up than MOT, but once set up, I saw little difference. The most difficult part of any program which uses a web interface with php3 against a MySQL database is compiling the ancillary applications (Apache, PHP3 and MySQL), since each relies on the other. Libraries required are Apache with php3 and MySQL.

grpn: http://wilkins.ne.mediaone.net/grpn.html

grpn is a calculator with Reverse Polish Notation (RPN), which I haven't used in a while. Fortunately, this application has a nice help facility—it was a needed and good refresher. **grpn** is very nice—goodbye **xcalc**, it was nice knowing you. Libraries required are gtk-1.1.13, libXext.so.6, libX11.so.6, libm.so.6 and glibc.

terraform: http://www.peoplesoft.com/peoplepages/g/robert_gasch/terraform/

This is a very nice looking terraform program with the ability to show a randomly generated area in several different ways, including 2-D Plane, 3-D Wire, 3-D Height and 3-D Light. All, even the 3-D light, ran fairly fast on my slow system. You can also choose color bands, gray scale, desert and red hot—just a few of the options. It is a nicely done application. Libraries required are libgtkmm, libgdkmm, gtk+1.1.13, libXext, libX11, libstdc++-libc6, libm and glibc.

David A. Bandel (dbandel@ix.netcom.com) is a Computer Network Consultant specializing in Linux. When he's not working, he can be found hacking his own system or enjoying the view of Seattle from an airplane.

LiS: Linux STREAMS

PhD. Graham Wheeler

PhD. Francisco J. Ballesteros

Denis Froschauer

David Grothe

Issue #61, May 1999

Not all networking software is based on BSD sockets. System V UNIX systems and most commercial networking code use STREAMS. The LiS project was developed to make STREAMS available for Linux, with the aim of making Linux the best UNIX platform for developing, debugging and using STREAMS software.

The input/output system in UNIX is far from simple and involves many different modules: networking involves different protocol stages arranged in protocol stacks; terminal I/O involves different “line disciplines” stacked over (perhaps network) devices. All those modules perform some processing on existing I/O data flows.

In Linux and most BSD systems, I/O modules live inside the kernel and the relations between them are more or less hard-wired into the code. As an example, the TCP/IP protocol stack is a carefully programmed set of modules with strong interrelations. It is designed to work well on typical configurations.

STREAMS is a flexible input/output system, initially designed to overcome the inflexibility found in previous UNIX systems (see Resources 1). It is an alternative to sockets and is used in most commercial UNIX versions. Some sort of STREAMS is needed if we ever want networking software from systems such as Solaris, Unixware, etc. to run off-the-shelf on Linux.

STREAM Definition

A STREAM (see Figure 1) is, in essence, a dynamically configurable stack of modules. Each module does some processing on a data flow as it goes from the device to the user or vice-versa. The user perceives a STREAM as a file. It is handled with the usual **open**, **read**, **write**, **ioctl** and **close** system calls. Data written by the user is packaged into messages, which are sent downstream. Data read by the user comes from messages sent upstream by an underlying device driver.

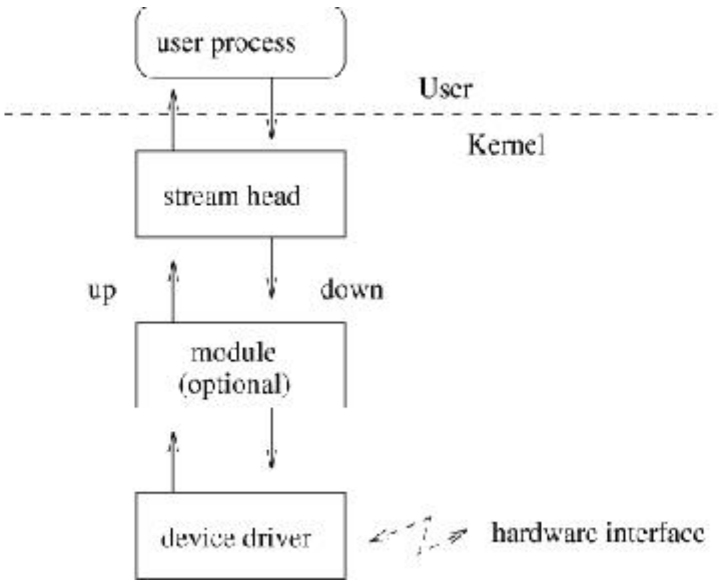


Figure 1. Components of a STREAM

A couple of additional system calls, **putpmsg** and **getpmsg**, allow the user to send and receive STREAMS messages directly. Yet another system call, **poll**, provides an alternate interface for **select**. Therefore, each STREAM is composed of these elements:

- A mandatory *STREAM head* talks to the user process doing I/O. The head fills the gap between user system calls and the message flow. Thus, a write into the STREAM is handled by the head by sending a message downstream. Conversely, a data message going upstream is used by the head to service read system calls on the STREAM.
- A (possibly empty) stack of STREAM modules typically performs some computation on messages passing by and forwards them either upstream or downstream. For example, IP on X.25 encapsulation (IXE) could be implemented as a STREAMS module; IP packets would be (de)encapsulated as they pass by the IXE module. A terminal-line-discipline module is another example; typed characters can be *cooked* as they cross the line-discipline module. A packet-sniffer module could thus be used as a diagnostic or debugging tool.

- A mandatory STREAM driver interconnects the STREAM to the device sitting below it. STREAM drivers can also be software only; for example, a STREAMS driver could be used to implement an SNMP MIB for the kernel, or a driver could be written to emulate the behaviour of a true hardware driver for development purposes.

A nice property of STREAMS is that different modules (or drivers) can be decoupled quite easily. Hence, they could be developed independently by different people who don't know the actual protocol stack where they will be used, provided the interfaces between the various modules and drivers are well-defined. STREAMS includes standard interfaces for the transport, network and data link layers. In addition, modules can be dynamically “pushed” onto (and popped off) the STREAM, which is a very convenient feature.

Finally, special *multiplexor* drivers allow several STREAMS to be multiplexed into another one (or ones). The **ip** module in Figure 2 is a multiplexor. In this example, it multiplexes both TCP and IP messages using either an Ethernet driver or an IP-on-X.25 driver. A full STREAMS network can be built (see Figure 2), and many different protocol stacks can be set up dynamically for operation.

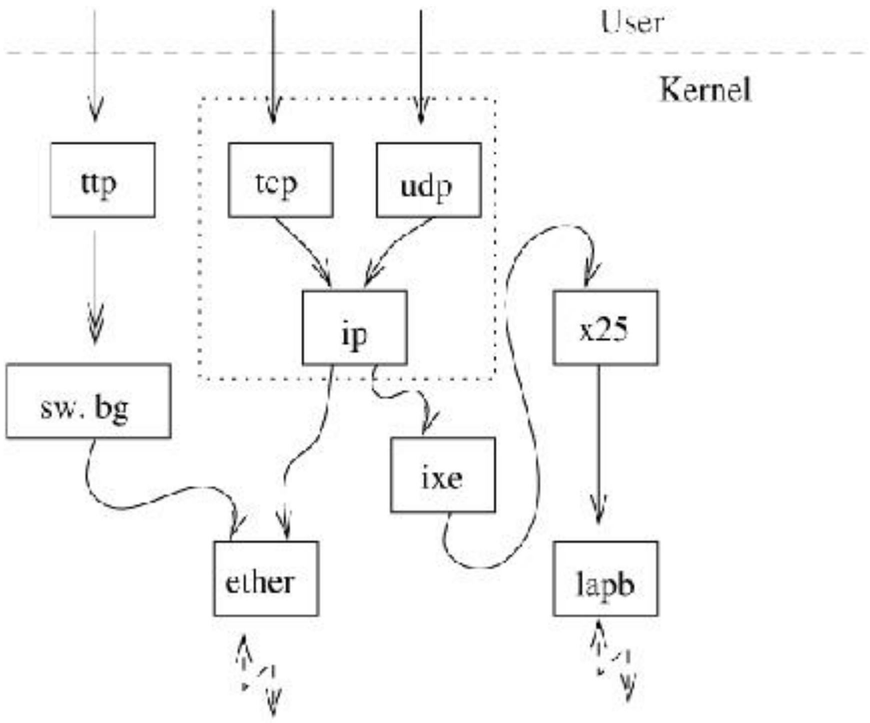


Figure 2. A STREAMS Network

The LiS Project

Before Linux was available, Dennis M. Ritchie designed STREAMS for the ninth edition of UNIX (see Resources 5). Since then, the STREAMS concept has been improved and revised by different operating systems. Variants ranging from

UNIX SVR4 STREAMS (see Resources 1) to Plan 9 Streams (see Resources 3) exist today.

Unfortunately, SVR4 complicated the neat and clean design of the ninth edition's STREAMS mainly to add new features, such as atomic gather/scatter writes and multiprocessor support. Different devices such as pipes and fifos were also re-implemented using STREAMS (see Figure 3).

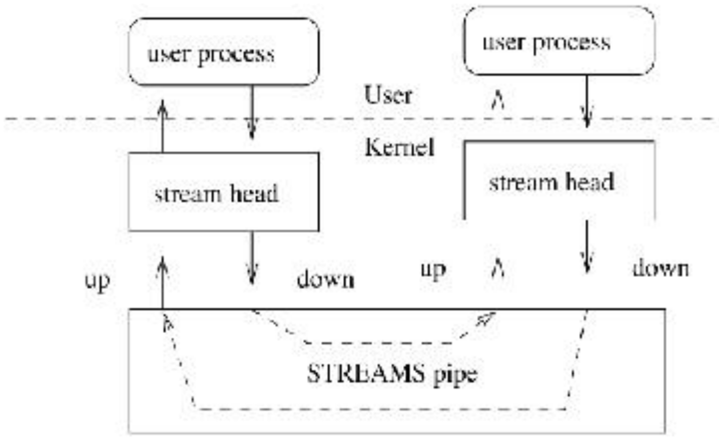


Figure 3. A STREAMS-Based Pipe

Despite being far more complex than Ritchie's Streams, SVR4 STREAMS simplify the construction of network software. Indeed, most networking software for UNIX System V machines is written using STREAMS (including the socket interface). We wanted to be able to run SVR4 driver software under Linux.

After some of us independently started to develop what would become LiS, we met on c.o.l.a. and decided to coordinate our efforts. An LiS project, LSM, was posted in March 1995 and the project began.

To STREAM or Not to STREAM

There are several reasons to use STREAMS: standard service interfaces to different layers (DLPI for the data link layer, NPI for the network layer and TPI for the transport layer), flexibility and SVR4 compatibility.

What we like about STREAMS under SVR4 is we can write device drivers conforming to a powerful but not Byzantine API (DLPI or TPI, in particular) and have existing network services (DLPI) or existing “applications” (read *non-kernel code*) work with the device with no additional effort.

Protocol stages (or modules) can be dynamically added/removed. Imagine you are debugging a transport layer interface (TLI) for X.25, and you can push and pop an **x25_tli** module many times. That is, it is an open framework. Those modules employed can, of course, be shared and reused in different places. With sockets, you have only what the kernel has.

The bottom line is that standards are a “Good Thing”. In the era of distributed systems, this applies equally to kernel-level network and communication interfaces. The STREAMS framework, APIs and service interfaces were designed by intelligent people at AT&T Bell Labs. The result is a mechanism which is clean, comprehensive and elegant to boot.

It has been argued against STREAMS that they are too complicated and too slow when compared to BSD sockets. A related argument is that TCP/IP networking is done more efficiently with BSD-style protocol stacks.

Consider this: Linux TCP/IP networking code can be used as-is with LiS. The purpose of LiS is simply to have the STREAMS framework available, not to replace the Linux TCP/IP protocol stack. Existing network software is perceived by STREAMS users as the dashed box in Figure 2. Fake modules interfacing with existing Linux drivers and protocol stacks are all we need.

With respect to simplicity, STREAMS make certain things, such as “deep” protocol stacks, more simple when compared to sockets. Sockets were designed for implementing TCP/IP-type networking and, although simple, are not *extensible*. That is, you can't easily use the sockets mechanism to build deep protocol stacks of which the sockets have no built-in knowledge. Each time the Internet protocol suite needs another layer, some hack is likely to be made to sockets.

The Internet protocol suites are deeper than the kernel implementors like to think they are. Consider TCP/IP when IP is sent using X.25 packets, transmitted with LAPB frames through a driver. Now you have a TCP/IP <-> X.25 <-> LAPB <-> driver stack. Then add another protocol over TCP/IP (say, NFS) and interpose frame relay (FR). The stack becomes: NFS <-> TCP/IP <-> X.25 <-> LAPB <-> FR <-> driver. Sockets are not equipped to build protocol stacks such as these that were not originally designed into it. It can be done, but is much easier and cleaner with STREAMS.

SVR4 STREAMS and LiS are much more complex than ninth edition STREAMS, but the added complexity is mostly to the STREAMS implementation and is hidden from the driver or module programmer.

Let the STREAM Flow Free

Most UNIX features have been available in source form for people to read and use. STREAMS was a notable exception. Therefore, even though we could have designed LiS to support just the STREAMS interface, we also tried to follow its design. If SVR4 STREAMS code had been available, the project could have been reduced to a simple port. As a result, the design guide was a mixture of a couple of books showing how STREAMS work (see Resources 2 and 4).

Availability of source code for the Linux kernel was crucial, as LiS requires small changes to existing kernel subsystems.

Starting from scratch, our aim was to make LiS portable so that other people could avoid rewriting it for use on different systems. By replacing a single small module, the whole framework can be ported to different operating systems. LiS portability is demonstrated by the fact that Gcom has ported it to QNX (a UNIX flavor). Ports for BSD UNIX system and even NT could be done without much effort in the future.

LiS Features and Implementation

A complete STREAMS description would be too large for this article. Some books you might read to learn more about STREAMS can be found in Resources. In a few words, LiS features include:

- support for typical STREAMS modules and drivers
- ability to use binary-only drivers
- convenient debugging facilities

Typical STREAMS Facilities

Many similarities exist between the implementation of LiS and SVR4 STREAMS. This is because initial project members followed the “Magic Garden” (see Resources 2) as a design guideline. Current maintainers were also heavily influenced by SVR4 STREAMS, because they had been writing STREAMS drivers for SVR4 since 1990. Thus, the stream head structure, queue structure, message structure, etc., follow the SVR4 model.

Differences between the two do exist. SVR4 disallows STREAMS multiplexors to use the same driver at more than one level of the stack. For example, if we had a STREAMS multiplexor driver called “DLPI” and another called “NPI”, the SVR4 STREAMS would disallow the stack: NPI(SNA) <-> DLPI(QLLC) <-> NPI(X.25) <-> DLPI(LAPB). LiS allows these combinations, since we could see no harm in such configurations.

The configuration file used for LiS is modeled after the SVR4 *sdevice* and *mdevice* files. However, LiS syntax is different and combines into a single file the functions that SVR4 used two files to specify. The LiS build process (Makefiles) allow individual drivers to have their own config file. They all get combined into one master config file, which is then used to configure LiS at build time.

In SVR4, the STREAMS executive is a linkable package for the kernel. It is not hard-wired into the kernel. With LiS, the STREAMS executive is actually a runtime, loadable module of the kernel, one step more dynamic than SVR4 STREAMS.

A quick overview of the LiS implementation would reveal a STREAM as a full-duplex chain of modules (see Figure 4). Each one consists of a queue pair: one for data being read and another one for data being written. Each module has several data structures providing those operations (i.e., functions) needed, as well as statistics and other data.

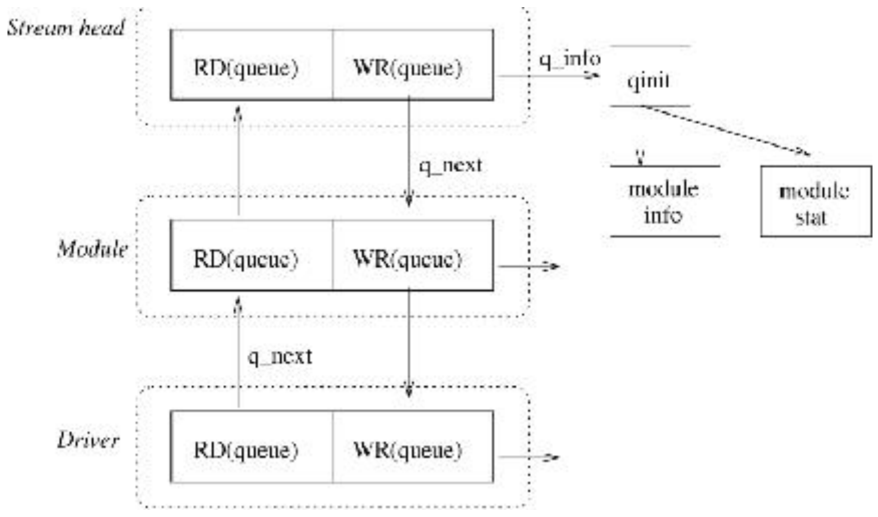


Figure 4. Queues in a STREAM

Module operations are provided by the programmer and include procedures used to process upstream and downstream messages. Messages can be queued for deferred processing, as LiS guarantees to call service procedures when queued messages could be processed.

Most of the LiS implementation deals with these queues and also with the message data structures used to send data through the STREAM. Messages carry a type code and are made of one or more message blocks. Only pointers to messages are passed from one module to the next, so there is no data copy overhead.

The head of the STREAM is another interesting piece of software. In Figure 5, you can see how it is reached from the Linux VFS (Virtual File System) layer which interfaces the kernel with the file systems. Note that even though Linux does not have a clean and isolated VFS layer, Linux i-nodes are v-nodes in spirit and its file system layer can be considered to be a VFS. For an actual description of the implementation, read Chapter 7 of the “Magic Garden” (Resources 2).

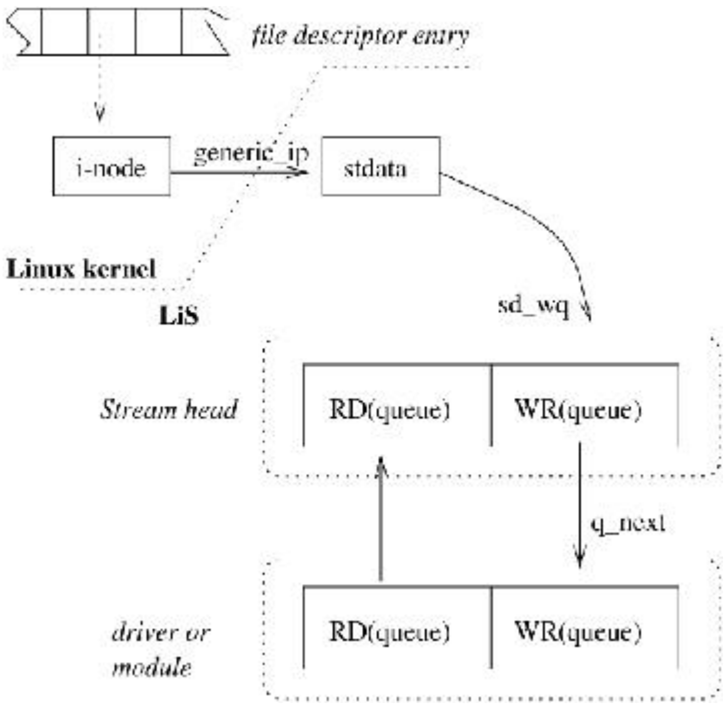


Figure 5. The STREAM Head

Binary-Only Drivers

LiS also makes provision for linking with binary-only drivers. This allows companies such as Gcom which have proprietary drivers to port their driver code to LiS and distribute binaries. This is an important feature if we expect companies to port their existing SVR4 STREAMS drivers to LiS. The more of these available, the more the Linux kernel functionality is enhanced.

Debug Facilities

LiS debugging features are especially convenient and show another departure point from SVR4.

Of course, these facilities include some general-purpose debug utilities such as message printers, but also included are significant aids that can really help with debugging, such as the ability to selectively trace; for example, **getmsg** calls.

The memory allocator keeps file and line numbers close to allocated memory areas. Combine that with the ability to print out all the in-use memory areas, and you have a tool for finding memory leaks in your drivers.

Usage statistics are designed to help, not overload the user with unnecessary information. The **streams** command prints out a good deal of useful information about LiS operation. There is even a debug bitmap to cause LiS to trigger different debug facilities. One of them is the ability to time various operations using the high-resolution timer. Thus, the user can get fine-grain driver timings for those drivers using LiS tools with no extra code in the driver.

Last but not least, LiS allows module debugging in user space by emulating the whole STREAMS framework. A module can be easily developed in user space and then downloaded into the kernel when it works. That is achieved by a “port” of LiS which runs in user space on Linux (in a dummied-up manner).

STREAMS modules can be tested by surrounding them with test modules and then driving known sequences of messages through the module under test. The LiS loop driver is suitable to place below the driver being tested, as it behaves like a simple echo server. The stream head may very well be all that is needed above.

STREAMS Works with Linux TCP/IP

The whole TCP/IP stack can be reused; thus, TCP/IP performance with STREAMS is a non-issue. LiS comes with an adapter driver that fits below standard Linux IP and interfaces off to STREAMS drivers using DLPI. Gcom uses this to interface their STREAMS-based Frame Relay and (soon) X.25.

Also, a contributed driver that will be distributed with LiS (sitting in Dave's inbox as of this writing) sits on top of any Linux MAC (mandatory access control) driver as a client and presents a DLPI interface above. Gcom will probably use this driver to interface its SNA (system network architecture) to the Linux token-ring driver.

LiS Licensing

LiS is licensed using the GNU Library Public License so that companies can port their existing SVR4 proprietary STREAMS drivers to LiS and use them in Linux without having to publish their source code. This is important if we are to encourage companies to support Linux with their “family jewels” products.

Final LiS Needs

It would help if support needed to run LiS could be included in the mainstream kernel. We are referring mainly to the new system calls and other small *hooks*, not to LiS itself. This support would make it easier for people to download LiS and install it without having to patch the kernel.

Resources

Graham Wheeler (gram@cdsec.com) obtained his Ph.D. in computer network performance analysis at the University of Cape Town in 1996. He subsequently spent conciderable time developing STREAMS device drivers and modules for protocol translation to enable a number of financial institutions to connect to PayNet, a large electronic commerce payment clearing center. He is a founder

and technical director of Citadel Data Security, specializing in Internet firewall and Virtual Private Network software development.

Francisco J. Ballesteros (nemo@gsyc.inf.uc3m.es) his Ph.D. in Computer Science in 1998 at the Technical University of Madrid (Spain). He is currently teaching and doing research on distributed and adaptable operating systems at Carlos III University of Madrid in strong cooperation with the Systems Software Research Group of the University of Illinois at Urbana-Champaign.

Denis Froschauer was a significant contributor to LiS development during its early implementation stage.

David Grothe (dave@gcom.com) is president of Gcom, Inc. Gcom produces data communications protocol stacks for UNIX systems, including Linux. Mr. Grothe founded Gcom in 1979 after working for the company now known as Advanced Computer Communications (ACC) where he wrote his first implementation of X. 25 in 1977. Prior to that, he was a professional programmer at the University of Illinois Urbana-Champaign.

Archive Index Issue Table of Contents

Advanced search

Adding Features to Dial-Up PPP Service

Lindsay Haisley

Issue #61, May 1999

Mr. Haisley provides some PPP customization scripts for web hosting services.

I operate a small web hosting service, specializing in services for the arts community and for small local businesses. As is customary, I offer a variety of support services to all my clients, including the ability to dial directly to my server and connect to the Internet using the point-to-point protocol (PPP). All services, including PPP access, are provided by a venerable and extremely reliable 486 running the latest production release of the Linux kernel.

Having spent a couple of years helping to build one of Austin's first Internet service providers during the early days of the Internet, I am familiar with the process of using UNIX to find original solutions to unsolved problems. At that time, there were no Portmasters, and dial-up access was provided by UNIX boxes, Digiboards and clever programming. Many solutions and tools which are commonplace today were unknown or only experimental then.

Modern releases of Linux offer a good deal more to work with than did the early non-Linux kernels; nevertheless, when I developed my own dial-up access, I found that while all parts of my dial-up system were available, several functions important to me were not available in the published packages. In particular, I wanted to offer full Internet access as a subscription service. Customers who chose not to subscribe to this service would still have full access via PPP dial up to the file spaces on their local web site using FTP or http. I also needed a way to implement session timeouts based on inactivity. While all my customers are above average in terms of their sense of responsibility in such matters, everyone has attention lapses from time to time, walking away from an on-line session without logging off and hanging up.

The software I found readily available was the excellent **mgetty+sendfax** package by Gert Doering and the Debian/GNU distribution of **pppd**. **mgetty** manages communication with a modem and provides essential login functions, while pppd manages PPP protocol issues. Recent versions of mgetty (I am using mgetty-0.99.2) are capable of detecting incoming attempts at PPP negotiation and, if properly configured, will invoke pppd with a variety of options, passing over to it the responsibility for user authentication.

While this combination is quite flexible, it is not a totally integrated solution. **pppd** reads and takes setup instructions from user .ppprc files, and my first thought was that setting up a read-only, root-owned .ppprc file for each customer would give me the flexibility I needed to provide the Internet access permissions and limitations I wanted on a per-account basis. Unfortunately, the situation is not quite this simple. Because mgetty runs and invokes pppd as root, the only .ppprc file which pppd reads is ~root/.ppprc. User identification and authentication takes place *after* the ~/.ppprc file is read—not very useful for my purpose.

Fortunately, pppd provides a very nice open-ended hook in the form of two built-in script calls: **ip-up** (Listing 1 in the archive file) which is executed immediately after the network control protocol (IPCP) for PPP has come up, and **ip-down** (Listing 2 in the archive file) which is executed immediately after the link has gone down. Both scripts are provided with the interface name, tty device, speed, local IP address and remote IP address as command parameters, and from these, almost everything one might need to know about a PPP session can be discovered. Both ip-up and ip-down run with a real and effective user ID of root, eliminating any potential problems with user-owned processes executing system commands. Because I could do my per-user configurations from a single script, I could localize all my user information in a single data file.

Proxy ARP

To grant full Internet access to a pppd dial-up client, pppd invokes a technique called proxy ARP. A host using proxy ARP advertises a dial-up client's IP address linked to its own Ethernet interface address. IP traffic destined for the dial-up client is therefore sent to the host, which dutifully forwards it via the PPP interface. **pppd** can be configured on invocation to set up proxy ARP, either on the command line or in any one of its several configuration files, including the user .ppprc file. Proxy ARP can also be set up “manually” using the **arp** command to manipulate the kernel's ARP cache. Because none of the pppd configuration files can be used to distinguish one user from another, I chose to set up proxy ARP using a shell invocation of arp in the ip-up Perl script.

Using the arp command to set up proxy ARP requires two pieces of information: the IP address assigned to the dial-up client and the machine address of the Ethernet interface to which packets should be delivered. The dial-up IP address is passed as a parameter to ip-up. The hardware address of the appropriate Ethernet interface can be obtained from a couple of sources, but the easiest way to is to parse the information returned from **ifconfig**. This address can also be hard coded into the script, since it is not likely to change in the short term.

For ip-up to know whether to set up proxy ARP, it must know the identity of the user for whom it was invoked. Although the identity of the current user is not one of the items provided to ip-up by pppd, the name of the connecting tty device *is* available and associated with a user name in the system's utmp file. Invoking **who** provides a conveniently formatted table, which can be parsed to obtain the name of the user currently connected to any tty device. I use the user name as an index into a small flat file, /etc/ppp/proxyarp, which consists of a series of lines of tab-delimited data pairs, each pair consisting of a user name and either a "+" or a "-" indicating whether to set up proxy ARP for that particular user. With this information, ip-up has everything it needs to set up proxy ARP for a session and determine if appropriate to do so.

One "gotcha" which must be addressed in this scheme is ARP caching by the LAN gateway. The Cisco 750 series router which I use is reluctant to provide any information on or means of manipulating its internal ARP cache, and the default timeout (about five minutes) means that any connection made within this timeout period after a previous call will inherit the packet routing of the previous connection. While it is not a serious problem for me if my non-Internet users occasionally get full Internet access, a busy ISP would need to be able to exercise tighter control in this matter.

Session Time Monitoring

Monitoring session time and network activity is relatively easy under Linux. All the necessary information on packet traffic through each interface is made available by the Linux kernel in the pseudo-file /proc/net/dev, laid out as follows in a format that is both easy to read and easy to parse (see Listing 4). Inactivity timeouts can be triggered by the number of packets received, packets transmitted or a combination of these.

Listing 4.

My timeout mechanism uses both the ip-up and ip-down scripts and a third Perl script, timeout.pl (Listing 3 in the archive file), which runs from the root crontab file every five minutes. **ip-up** creates a session file, /var/run/ppp*n*.session (where *n* in ppp*n* designates the appropriate interface). This file contains six fields:

- The user name of the account owning the session (for logging and notification)

- The process ID of the ppid process
- The time the session began
- The time activity was last observed on the interface
- The total number of packets received on the interface
- The total number of packets transmitted to the interface

timeout.pl reads the session file for each PPP interface each time it runs. If the total session time has not been exceeded, it checks the traffic on the interface. If it observes activity, it records the time and traffic statistics and rewrites the session file. If no traffic has occurred since the last check, the script checks the time since traffic was last observed and exits if the inactivity timeout has not been exceeded. If either of the timeout times has been exceeded, the script sends a SIGINT signal to the pppd process, causing it to execute an orderly hang up, which includes execution of the ip-down script. **ip-down** deletes the session file and any proxy ARP entry for the interface currently in the ARP cache.

With the exception of the reluctant router ARP cache noted above, this system works quite well in all respects. I have included optional e-mail notification in timeout.pl, so it sends me e-mail whenever a timeout occurs. I can also force a timeout by executing timeout.pl manually with a **-t** or **-i** option. Adding system logging of timeout events is on my “to do” list, but should be a relatively simple matter.

Resources

Lindsay Haisley (fmouse@fmp.com) lives and works in the Austin, Texas area where he owns and operates FMP Computer Services, providing web hosting and consulting services for small businesses and arts enterprises. He is a founding member and currently the coordinator of the Central Texas Linux Users Group (http://www.ctlug.org) as well as an officer with the Capital Area Central Texas UNIX Society (CACTUS).

A Toolbox for the X User

Christoph Dalitz

Issue #61, May 1999

An introduction to several small graphical tools for the daily work of system administration.

Under Linux, you can do virtually everything from the command line. For the administrator of a Linux server, this is extremely useful for two reasons:

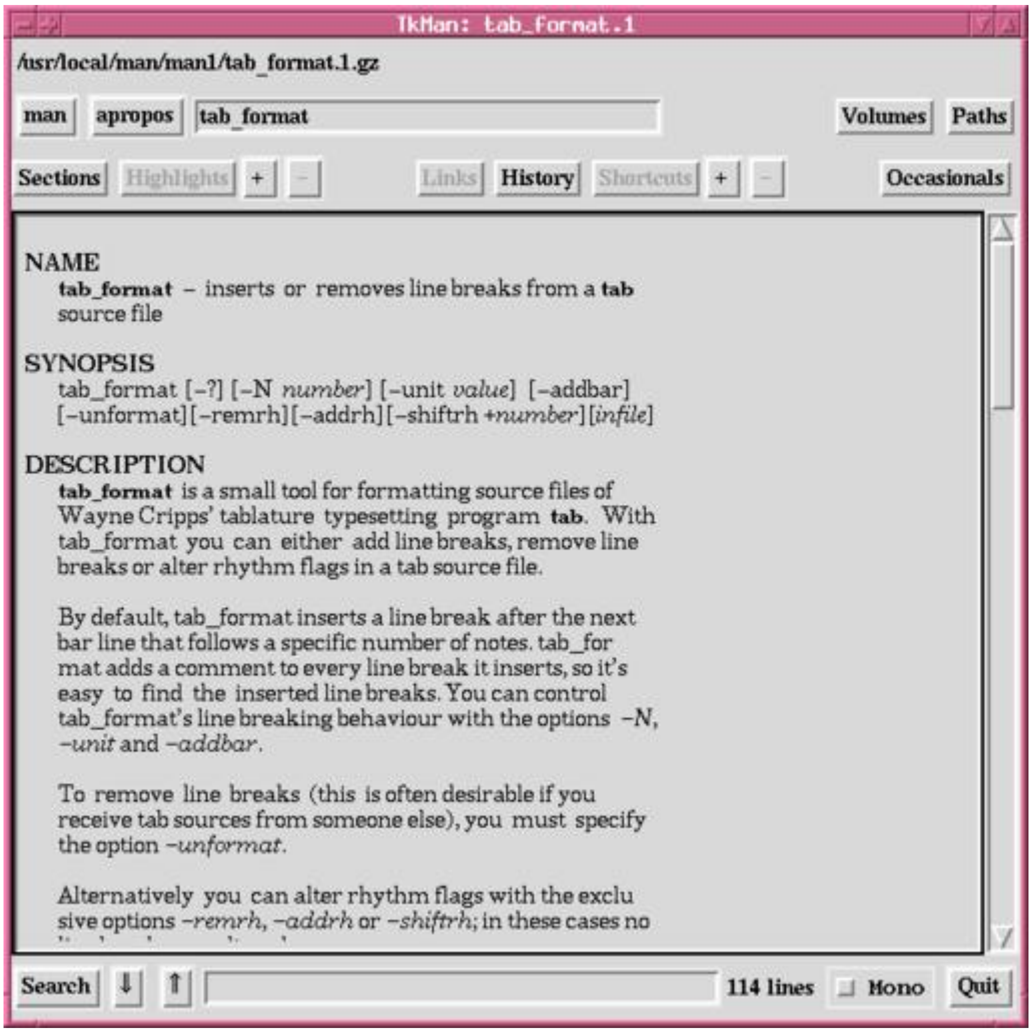
- All administrative tasks from any site in the local network can be done via a simple TELNET session.
- A lot of administrative work can be easily automated via shell scripts.

Compared to a single-user system such as Windows NT, which knows no remote login, lacks a scripting language and has no command-line equivalent to many administrative point-and-click operations, these are serious advantages which save the administrator eons of time and help make Linux an “admin-friendly” system.

However, for those who use Linux as their desktop OS, many operations would be easier with a GUI than by typing cryptic commands at the shell prompt. Even for the conservative user who is not yet ready to exchange his laboriously customized FVWM for KDE, Linux offers many graphical tools for common tasks.

Getting Help

For information on a specific command, you need to read its man page, which can be done by typing **man** at the command line. A more comfortable way, however, is to use a man page viewer such as **xman** or **tkman**. While xman is an ugly grey mouse from the early days of the X Window System, tkman by Tom Phelps is a truly nice GUI for browsing man pages.



```
NAME
     tab_format - inserts or removes line breaks from a tab
     source file

SYNOPSIS
     tab_format [-?] [-N number] [-unit value] [-addbar]
     [-unformat][-remrh][-addrh][-shiftrh +number][infile]

DESCRIPTION
     tab_format is a small tool for formatting source files of
     Wayne Cripps' tablature typesetting program tab. With
     tab_format you can either add line breaks, remove line
     breaks or alter rhythm flags in a tab source file.

     By default, tab_format inserts a line break after the next
     bar line that follows a specific number of notes. tab_for
     mat adds a comment to every line break it inserts, so it's
     easy to find the inserted line breaks. You can control
     tab_format's line breaking behaviour with the options -N,
     -unit and -addbar.

     To remove line breaks (this is often desirable if you
     receive tab sources from someone else), you must specify
     the option -unformat.

     Alternatively you can alter rhythm flags with the exclu
     sive options -remrh, -addrh or -shiftrh; in these cases no
```

Figure 1. Manpage Viewer tkman

tkman displays man pages in a pleasant way (see Figure 1), knows hypertext links to other related man pages, allows regexp searches within the man page and has a built-in **apropos** command that offers man pages for a given keyword. If you want to read a man page from a specific section, you must add a dot and the section number in the command entry field; e.g., if you want to get help on the C library function **printf** (man page section 3) rather than the shell command **printf** (man page section 1), you must enter **man printf.3**.

Moreover, you may print out the displayed man page, but printing is a bit tricky to get working. Printing is started from the menu "Occasionals"->"Kill Trees"->"lp", which invokes the man page text processor **groff** with the options **-Tps** for PostScript output and **-l** to send output directly to the printer. The latter option will not work unless the print command is specified in groff's configuration file /usr/share/groff/font/devps/DESC. Hence, you will need to consult groff's man page if printing from within tkman does not work.

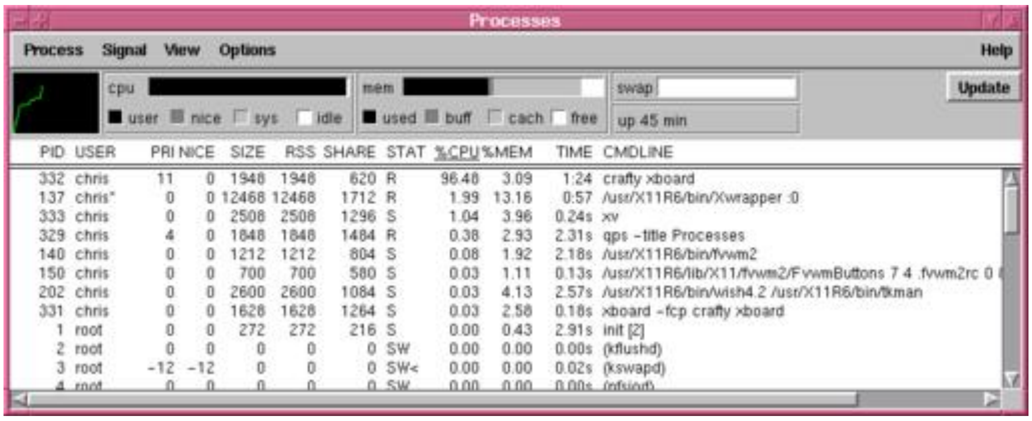
Another source of information is hypertext info pages, which consist of nodes with an internal hierarchical structure. While man pages are more appropriate for the documentation of simple shell commands, info pages are more appropriate for programs or libraries that need extensive documentation—theoretically. In reality, some crazy programmers decided to move the documentation of elementary shell commands like **ls** or **rm** from man pages to info pages; hence, everyone needs an info reader.

On the command line, you can read info pages with the **info** command, which fires up Emacs in its info mode. Alternatively, you can use **tkinfo** by Kennard White and Axel Boldt as a graphical info page viewer. When started without any command-line argument, tkinfo displays the “dir” info node under which the GNU utilities reside. If you do not like tkinfo's display font, you can add a **tkinfo*Text.font** resource in your .xresources or .xdefaults file. Although tkinfo has a better-organized display and is more intuitive to use than Emacs' info mode, it also has a serious drawback. Most nodes entered in the “File”->“Goto Node” menu are not found. For example, tkinfo cannot find the node “ls”, even when invoked via **tkinfo ls** on the command line. To reach the node “ls”, you must navigate through the info hierarchy, a time-consuming maneuver.

Fortunately, most commands are documented completely in their man page as well; hence, there is often no need to bother with info pages.

Controlling Processes

qps by Matthias Engdegard is a graphical incarnation of **ps**, top and **kill** and is based on the Qt toolkit. **qps** is an attractive and powerful tool. You can use it as an advanced version of **top** by selecting “All Processes” from the “View” menu, specifying the “Update Period” in the “Options” menu and clicking on “CPU” in the header line of the process list to make qps sort the list by the used CPU time. This will reveal which processes are eating the processor time on your system (see Figure 2). Or you can use qps as a combination of *ps* and *kill* by selecting a process from the process list and sending a signal from the “Signal” menu.



```
up 45 min

  PID USER     PRI NICE  SIZE   RSS SHARE STAT %CPU %MEM   TIME CMDLINE
  332 chris     11    0  1948  1948   620 R   96.48  3.09  1:24 crafty xboard
  137 chris*     0    0 12468 12468  1712 R    1.99 13.16  0:57 /usr/X11R6/bin/Xwrapper :0
  333 chris      0    0  2508  2508  1296 S    1.04  3.96 0.24s xv
  329 chris      4    0  1848  1848  1484 R    0.38  2.93 2.31s qps -title Processes
  140 chris      0    0  1212  1212   804 S    0.08  1.92 2.18s /usr/X11R6/bin/fvwm2
  150 chris      0    0   700   700   580 S    0.03  1.11 0.13s /usr/X11R6/lib/X11/fvwm2/FvwmButtons 7 4 .fvwm2rc 0
  202 chris      0    0  2600  2600  1084 S    0.03  4.13 2.57s /usr/X11R6/bin/wish4.2 /usr/X11R6/bin/tkman
  331 chris      0    0  1628  1628  1264 S    0.03  2.58 0.18s xboard -fcp crafty xboard
    1 root       0    0   272   272   216 S    0.00  0.43 2.91s init [2]
    2 root       0    0     0     0     0 SW   0.00  0.00 0.00s (kflushd)
    3 root     -12  -12     0     0     0 SW<  0.00  0.00 0.02s (kswapd)
    4 root       0    0     0     0     0 SW   0.00  0.00 0.00s (nfsiod)
```

Figure 2. Process Control with qps

Managing Files

At first glance, the wide variety of file managers (xfilemanager, xdtm, mfm, xfm, xgroups, et. al.) that comes with every Linux distribution seems to be promising. However, almost all of these tools are either very basic, very ugly or both. TkDesk by Christian Bolik is the only exception I have encountered.

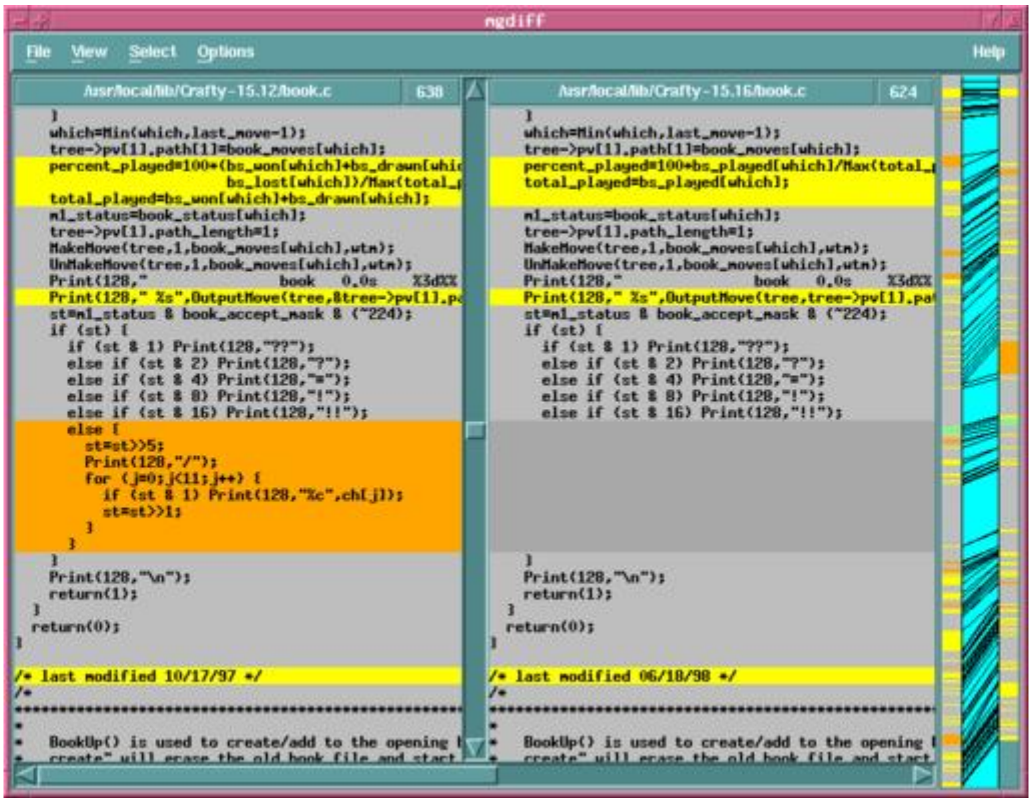
Actually, TkDesk is much more than a simple file manager (see “Introducing TkDesk” by John Blair in *LJ*, March 1998). By default, it even starts a separate button bar that reigns over the desktop. This obtrusive behaviour is easily turned off via the menu “TkDesk”->“Toggle Appbar”. When I started using TkDesk, it often stuck for indefinite intervals with no obvious reason. It took me some time to realize that it was trying to create sound effects not supported by my system. If this default setting causes trouble, it can be turned off in the menu “Options”<->“Use Sound”.

I use TkDesk primarily for browsing files that are buried deep in my system directory trees. This can be done very fast, since TkDesk has a built-in file browser/editor. Moreover, it displays files in three directory list boxes, making it easy to change directories back and forth. Three is the default, but you can change it to any number you wish.

Comparing Files

Often, there is a need to check the differences between two versions of a file. You can use the shell command **diff** for this purpose, but the output of the graphical tools **mgdiff** or **tkdiff** is much easier to read.

mgdiff by Daniel Williams can be invoked like diff with two file names as command-line parameters. Alternatively, it allows interactive file selection from the “File”->“Open” menu; hence, it is possible to invoke mgdiff from your window manager's program menu. **mgdiff** displays the selected files in two boxes and shows an overview of differences in a small bar on the right (see Figure 3). Changes, insertions and deletions are highlighted in different colors, which can be customized by the corresponding X-resources in your .xresources or .xdefaults file; see mgdiff's man page for details. Moreover, you can easily merge the compared files into a new file simply by clicking on the respective versions of the differences and saving the result with the “File”->“Save As” menu.


mgdiff
File
View
Select
Options
Help
/usr/local/lib/Crafty-15.12/book.c
638
/usr/local/lib/Crafty-15.16/book.c
624

```
    }
    which=Min(which,last_move-1);
    tree->pv[1].path[1]=book_moves[which];
    percent_played=100*(bs_won[which]+bs_drawn[whic
                        bs_lost[which])/Max(total_p
    total_played=bs_won[which]+bs_drawn[which];
    m1_status=book_status[which];
    tree->pv[1].path_length=1;
    MakeMove(tree,1,book_moves[which],wtm);
    UnMakeMove(tree,1,book_moves[which],wtm);
    Print(128,"                   book    0.0s    %3d%%
    Print(128," %s",OutputMove(tree,&tree->pv[1].p
    st=m1_status & book_accept_mask & (~224);
    if (st) {
      if (st & 1) Print(128,"??");
      else if (st & 2) Print(128,"?");
      else if (st & 4) Print(128,"=");
      else if (st & 8) Print(128,"!");
      else if (st & 16) Print(128,"!!");
      else {
        st=st>>5;
        Print(128,"/");
        for (j=0;j<11;j++) {
          if (st & 1) Print(128,"%c",ch[j]);
          st=st>>1;
        }
      }
    }
    Print(128,"\n");
    return(1);
  }
  return(0);
}

/* last modified 10/17/97 */
/*
*******************************************************
*
*   BookUp() is used to create/add to the opening b
*   create" will erase the old book file and start
```

```
    }
    which=Min(which,last_move-1);
    tree->pv[1].path[1]=book_moves[which];
    percent_played=100*bs_played[which]/Max(total_p
    total_played=bs_played[which];

    m1_status=book_status[which];
    tree->pv[1].path_length=1;
    MakeMove(tree,1,book_moves[which],wtm);
    UnMakeMove(tree,1,book_moves[which],wtm);
    Print(128,"                   book    0.0s    %3d%%
    Print(128," %s",OutputMove(tree,tree->pv[1].pat
    st=m1_status & book_accept_mask & (~224);
    if (st) {
      if (st & 1) Print(128,"??");
      else if (st & 2) Print(128,"?");
      else if (st & 4) Print(128,"=");
      else if (st & 8) Print(128,"!");
      else if (st & 16) Print(128,"!!");








    }
    Print(128,"\n");
    return(1);
  }
  return(0);
}

/* last modified 06/18/98 */
/*
*******************************************************
*
*   BookUp() is used to create/add to the opening b
*   create" will erase the old book file and start
```

Figure 3. Differences Made Visible with mgdiff

For programmers using a version control system, **tkdiff** by John M. Klassa is another useful difference tool. In contrast to mgdiff, tkdiff can be invoked only from the shell prompt because it requires file names as command-line arguments. **tkdiff** has an internal help function but no external man page, which occasionally makes it inconvenient to get usage information. Beside these trifles, tkdiff offers the same functionality as mgdiff. Moreover, you can check a file versus different versions of that file registered in a version control system (RCS, CVS or SCCS). For example, the command **tkdiff -r** ***`filename`*** compares *filename* with the revision most recently checked in.

Managing Archives

The standard data exchange format between different UNIX computers is the tape archive format (tar). Additionally, these archives are often compressed with **compress** (commercial UNIX systems) or **gzip** (Linux), resulting in tgz files. You can use the shell command **tar** to create, extract or list the contents of a tar file. Alternatively, **xtar** and **tkzip** provide graphical front ends to tar.

xtar by Rik Turnbull can only list and extract archives, but normally that is all you need. When you open an archive with xtar, it displays a list of all files in the archive. A double click on a file in the list starts a built-in file browser. This is very useful for new software packages, because you can read the installation instructions and README files before unpacking the archive.

If you prefer a graphical tool for creating archives, you need a more elaborate package such as TkZip by Robert Woodside. In my opinion, TkZip is too elaborate: each mouse click opens a new window, which quickly becomes confusing, and my eyes cannot get accustomed to the colored frames and buttons.

When fired up, TkZip displays a list of the files in the current directory; archive files can be opened with a double click on the respective file. As with xtar, you can browse text files by clicking on the files in the archive list. However, you must first specify a graphical viewer (e.g., **xless**) from the viewer list; otherwise, TkZip will write the contents of the file to standard output.

Creating archives with TkZip is a bit more involved, since it requires hopping through a suite of file selection dialogs:

- From the main window, select “File”->“New Archive”.
- Enter the archive type and the archive file name and choose “Create”.
- Click on “Add” in the pop-up archive-list window.
- Select single files for the archive by clicking on files in the next window and choose the “Add” button.
- Clicking on “Close” in the archive-list window eventually writes the archive file.

Compared to the tar shell command, creating an archive with TkZip is much more intricate and can save time only in the rare situation when you want to pack several files from different ends of your file system into one archive. In the more common case, when you simply want to pack files from one or two directories, the shell command is both faster and easier to use.

Resources

Christoph Dalitz received a Ph.D. in physics at the University of Bielefeld and is currently designing optical archiving systems for the Comline Company in Dortmund. While on his job, he has to work primarily under Windows NT beside some hours of recreation under HP-UX. He enjoys using Linux exclusively on his home PC. He can be reached at dalitz@infotech.de.

Reading E-mail Via the Web

Reuven M. Lerner

Issue #61, May 1999

How to write your own program to read and send mail to any server on the Internet.

E-mail is one of the unsung heroes of the Internet. The Web makes the Internet fun and interesting and allows me to keep up with most newspapers and magazines from the comfort of my Haifa apartment. E-mail allows me to keep in touch with friends, family and clients, as well as receive electronic newsletters in a convenient format.

I usually travel with my trusty Linux laptop, which means that with the help of a telephone line, I can dial in to my Internet provider and download the latest mail. However, on some occasions I cannot dial in to check my mail, even though I have full Internet access and a web browser. I could get an account at Hotmail, but Hotmail allows you to read mail sent to its server only, not to any mail server on the Internet.

This month, I will show you how to develop a set of CGI programs to read e-mail from any POP server. These programs do not provide a full-fledged e-mail client, but they do fill a niche and are useful in certain circumstances. The software described this month should demonstrate how relatively simple it is to create such applications and will have the added bonus of providing basic functionality for the times when you are away from the office.

What is POP?

Traditionally, e-mail on UNIX systems is stored on the user's computer. If you have an account on a UNIX system, e-mail sent to you is placed in a file on your computer. I receive mail on my Linux system in the file /var/spool/mail/reuven.

However, this system became inadequate over time for a variety of reasons. As users began to have their own full-fledged UNIX workstations rather than terminals connected to a central computer, system administrators wanted to centralize incoming mail on a single server.

The answer was POP, “post office protocol”. Rather than retrieving mail from a file on their own system, users would download it from the POP server, with a single POP server per work group cluster. A POP server typically stores incoming mail in a traditional UNIX-style file, but allows retrieval and deletion of individual messages via the network. Just as some cities and towns require their residents go to a central post office in order to retrieve letters and packages, POP requires users to retrieve their mail from a central server.

POP has gone through a number of updates over the years, with the most recent update named POP3. Over time, additional functionality has been added, but the basic commands have remained the same. POP allows users to check if they have mail, retrieve one or more messages and delete one or more messages.

Users are generally shielded from the underlying mechanics of POP3. Most modern e-mail programs support POP3. Indeed, e-mail programs on non-UNIX systems depend on the existence of POP3 servers, since they are rarely able to run mail servers known as “mail transport agents” or “MTAs”. Sendmail and qmail are two examples of MTAs.

Net::POP3

Before writing a CGI program to read our mail, we must understand how the program can accomplish this feat. We could write our own software to talk to a POP3 server, but as is often the case with Perl, a module already exists to handle this for us. In this particular case, the module is **Net::POP3**, part of the “libnet” package of network modules available on CPAN. (For more information on CPAN and its mirrors, go to http://www.cpan.org/.)

Net::POP3 provides an object-oriented interface to POP, making it possible to connect to a POP server with only a basic understanding of how the protocol works. Import the module with

```
use Net::POP3;
```

then create a new object with

```
my $pop = new Net::POP3($mailserver);
```

where **$mailserver** is a scalar containing the name of our POP3 server. If the connection is successful, **$pop** will be an object with methods allowing us to read and delete messages on the mail server. If the connection is unsuccessful,

$pop will be undefined. Now all methods in Net::POP3 work this way, returning **undef** if the call was unsuccessful. The following code checks for this condition:

```
die "Error connecting to $mailserver."
   unless (defined $pop);
```

In order to ensure e-mail remains private, POP3 servers require users to log in with a user name and password. The **login** method accomplishes that, returning the number of messages waiting for the user:

```
my $num_messages = $pop->login($username,
   $password);
die "Error logging in." unless (defined
   $num_messages);
```

Again, notice the test to see whether **$num_messages** is defined. If it is undefined, then a mistake probably occurred in either the user name or password.

Each message on the POP server is identified with an index number, ranging from 1 to **$num_messages**. The index number should stay constant during a single POP session, but will change during future sessions. You can use the index number to read or delete a message:

```
my $message_ref = $pop->read($index);
```

If message number **$index** exists, the message headers and body are put into an array reference. Thus, if **$index** points to a message on our POP server, **$message_ref** is an array reference. Each element of the array contains a single line of text from the message. We can print the contents of the message by dereferencing **$message_ref**:

```
print @$message_ref, "\n";
```

print-mail.pl

Now that we have seen how Net::POP3 allows us to retrieve and read mail from a POP server, let's look at how we can integrate it into a CGI program. First, an HTML form is needed as a way to enter a user name and password. Here is a simple one:

```
<HTML>
<Head>
<Title>Read your mail!</Title>
</Head>
<Body>
<H1>Read your mail!</H1>
<P>Enter your user name, password, and POP server.</P>
<Form method="POST"
action="/cgi-bin/print-mail.pl">
<P>POP server: <input type="text" name="mailserver"></P>
<P>Username: <input type="text" name="username"></P>
<P>Password: <input type="password" name="password"></P>
<P><input type="submit" value="Show me my mail!"></P>
</Form>
```

```
</Body>
</HTML>
```

The above form sends three parameters to our CGI program—the name of the POP server from which to download the mail, the user name and the password. If you are concerned about the password being sent in the clear, you might want to put the form and CGI program behind a server running SSL, the secure sockets layer. You might also want to investigate POP3's **APOP** login method, which hides the password somewhat.

The program for reading mail is fairly simple; see Listing 1 in the archive file, ftp://ftp.linuxjournal.com/pub/lj/listings/issue61/3359.tgz. The code starts by creating an instance of CGI, providing an object-oriented interface to the CGI protocol. Then an appropriate MIME header is sent to the user's browser, indicating the response will be in HTML-formatted text. Next, the three pieces of information necessary for retrieving the user's mail are grabbed: the name of the POP server, the user name and the password.

Once that information is retrieved, we try to connect to the POP server and log in. Normally, invoking **die** is a bad idea in a CGI program, since it results in a difficult-to-understand message appearing on the user's screen. However, since we ported **CGI::Carp** and specified **fatalsToBrowser**, any invocations of die will send a description of the error message to the browser as well as to the web server's error log. This can be an invaluable tool when debugging, even if your final production code requires you to hide potential error messages.

Once the number of messages waiting on the POP server is known, we can retrieve them with a simple loop:

```
foreach my $index (1 .. $num_messages)
{
   print "<H2>Message $index</H2>\n";
   my $message_ref = $pop->get($index);
   print "<pre>\n", @$message_ref, "</pre><HR>\n";
}
```

We enclose the mail within <pre> and </pre> tags, since most e-mail depends on fixed-width fonts and formatting.

You may be surprised such a simple program can be used to read your mail, but it does and should work on any system with any web browser. It can be used to quickly check if any new mail has arrived, without affecting your ability to download and read messages with your usual e-mail program.

Ignoring Uninteresting Headers

As is often the case with new programs, our first stab was functional but is missing some useful features. For instance, most users do not need to see all of the headers that come with a message. Typically, they want to see only the “From”, “To”, “Subject”, “Cc” and “Date” headers.

Perl makes it a snap to remove unwanted headers by using regular expressions. Headers can be thought of as a name, value pair separated by a colon. On the left side of the colon is the header name, which can consist of any alphanumeric character or a hyphen. On the right side of the colon is the header's value, which can consist of almost any character.

One consideration is the possibility that a header will be spread across multiple lines. That is, the two lines

```
Subject: This is a subject header
   that continues onto a second line
```

should all be considered part of the “Subject” header, since the second line begins with one or more white-space characters.

This problem is solved by creating a hash, **%KEEP**, in which the keys name the headers to keep. For example:

```
my %KEEP = ("To" => 1,
   "From" => 1,
   "Subject" => 1,
   "Date" => 1);
```

The code then checks if a header is to be kept by checking the value of **$KEEP{$header_name}**, where **$header_name** contains the value of the header to check.

Before anything can be done to the headers, they must be put into a scalar separate from the message body. Do that with **split**:

```
my ($headers, $body) = split "\n \n", $contents, 2;
```

Notice split has three arguments, telling Perl to split **$contents** into a maximum of two elements. If the 2 were omitted, **$body** would contain only the first paragraph of the message, rather than the entire text.

Once the message headers are stored in **$headers**, it can be split back into an array, and the code can then iterate through the array elements. Each element of **@headers** is a single header line, which might mark the beginning of a new header or the continuation of an existing one. If this is a new header and its name is in **%KEEP**, the header is written to the user's browser. If the header's name is not in **%KEEP**, it is ignored and the program goes on to the next line.

This does not solve the issue of multi-line headers. This is handled by assuming that every line in **@headers** will begin with either a header (e.g., **Received:** or **X-Mailer:**) or with white space. If the pattern at the beginning of the line matches a header value, the program checks **%KEEP** and if found, prints the line. If the pattern fails to match a header value, it is assumed to be white space, and the line is printed only if the previous line was printed.

Here is some basic code to print the headers:

```
my @headers = split "\n", $headers;
my $previous = "";
foreach my $line (@headers)
{
   if ($line =~ m/^([\w-]+):/i)
   {
      $previous = $1;
   }
   print $line, "\n" if $KEEP{$previous};
}
```

This code is contained in Listing 2, better-print-mail.pl, in the archive file. This is an improved version of our original bare-bones program, incorporating this and other changes.

Handling HTML

Displaying e-mail messages in a web browser has advantages and disadvantages. On the one hand, we must be careful to turn special characters, such as < and >, into their literal equivalents. At the same time, we can take advantage of the web browser to make e-mail addresses and URLs clickable.

Since we want to ensure that characters appear in the headers as well as in the message body, we modify **$contents**, the variable that contains the entire message contents, before separating the header and body. We turn < and > into < and >, respectively, ensuring that literal text will not be interpreted as if enclosed in HTML tags:

```
$contents =~ s/</</g;
$contents =~ s/>/>/g;
```

Making e-mail addresses clickable requires the use of a regular expression to match e-mail addresses. I decided to use the following code:

```
$contents =~
   s|([\w-.]+@[\w-.]+\.[a-z]{2,3})|
   <a href="mailto:$1">$1</a>|gi;
```

which looks for any combination of alphanumeric characters, hyphens and periods, followed by an @, followed by the same combination of characters, followed by a two- or three-letter top-level domain. This ensures we will not accidentally turn something like

```
   three pickles @ 20 cents/pickle
```

into an e-mail address. By turning an actual e-mail address into a “mailto” link, users can click on the link in order to send mail to that address.

Making URLs clickable is somewhat more difficult, since we have to handle more combinations. The code below appears to match a large number of URLs:

```
      s|(\w+tps?://[^\s&\"\']+[\w/])|
      <a href="$1">$1</a>|gi;
```

Here, we look for any letters ending with “tp”, with an optional “s” on the end. This allows us to match “ftp”, “http” and “https”, all of which are valid protocols. We then allow any combination of characters following the two slashes, excluding white space and several characters which cannot be transmitted in a URL.

Quotation marks and white space can be sent if they are URL-encoded first. Characters are URL-encoded when the hexadecimal value of their ASCII code is preceded by a percent sign. For instance, the space character is ASCII 32 or 0x20; thus, it can be sent in a URL as %20. CGI.pm automatically decodes such characters, so you need not worry about it in most cases.

The final part of our regular expression stipulates that the final character of a URL must be alphanumeric or a slash. This ensures that odd trailing characters, such as periods and commas, will not be accidentally dragged into the URL and highlighted.

Viewing Selected Messages

The above program works just fine, if you want to view all the messages in your mailbox. If you receive many e-mail messages, viewing all of them in a single long web document can get frustrating.

The program better-print-mail.pl takes into account the fact that we might want to view only a selected list of messages. For example:

```
   if ($query->param("to_view"))
   {
      @message_indices = $query->param("to_view");
   }
   else
   {
      @message_indices = (1 .. $num_messages);
   }
```

An HTML form element can be set multiple times, meaning that the element **"to_view"** might contain zero, one or more elements. All of those are put inside of **@message_indices** unless **to_view** was not set, in which case all messages are displayed by default.

How can we get a list of current messages? A program called mail-index.pl (see Listing 3 in the archive file) should do the trick. This program can be invoked from the same sort of form we have seen already; simply modify the “action” to point to mail-index.pl, rather than better-print-mail.pl. As with print-mail.pl and better-print-mail.pl, mail-index.pl must receive the user name, password and name of the mail server in order to function. With that information in hand, it logs into the POP server and displays the message headers for mail waiting to be read.

Each message is presented with a check box. By checking the box next to a message, the user indicates he would like to read that particular message. When the user clicks on the “submit” button, better-print-mail.pl is sent not only the user name, password and mail server, but also the list of checked messages. As we have seen, better-print-mail.pl already knows how to handle this list and prints only requested mail messages.

Conclusion

Setting up a web-based mail system is not all that difficult. I would hesitate before adding a **delete** function, since I would worry about deleting my only copy of a message. (My e-mail program makes automatic backups, so I never have to worry about that on my own computer.) However, adding such functionality would be quite easy, technically speaking.

Next month, I will show you how to build a system that allows you to send mail as well as read it. We will build on the software we examined this month, adding some functionality to it and tying it into our own mail-sending CGI programs. With a bit of software, you too can begin to compete with Hotmail!

Reuven M. Lerner is an Internet and Web consultant living in Haifa, Israel, who has been using the web since early 1993. His book Core Perl will be published by Prentice-Hall in the spring. Reuven can be reached at reuven@lerner.co.il. The ATF home page, including archives and discussion forums, is at http://www.lerner.co.il/atf/.

Archive Index Issue Table of Contents

Advanced search

Letters to the Editor

Various

Issue #61, May 1999

Readers sound off.

Excellent Article on X Administration

The article “X Window System Administration” by Jay Ts in the December 1998 issue was well-written and full of relevant information. I am the IT Manager at a Novell/NT shop and have been using Linux at home off and on over the past couple of years. I still consider myself a novice user. A couple of weeks ago, I secretly switched a few of my users from an NT/IIS server to a Linux/Apache server running our Intranet. They noticed a definite increase in performance, and I plan to eventually move everyone over to the Linux server. However, on my end I was having problems setting up X on the server and finally decided that the command line would do. Then I read the article on X administration in *LJ*. Now X is up and running and configured to my specifications. Thanks for the help.

—Barry Julien bjulien@wallacefunds.org

User Friendly

Just wanted to let you know that I think adding the comic “User Friendly” to *Linux Journal* was one of the coolest things you have done. Well, on top of the awesome tech articles, etc. Thanks.

—Shawn Nyczd scordia@eden.rutgers.edu

Linux Threatens More Than Microsoft

Everywhere I look, I see articles describing the threat Linux poses to Microsoft. While there is some truth to this, I think what everyone seems to be overlooking is the threat it poses to other UNIX systems. I think this is clearly demonstrated by the fact that Sun and SCO have started offering free licenses of Solaris 7 and SCO 5.0.4 for educational and non-commercial use (users must pay a fee of approximately $20 US for the media and shipping). Admittedly, commercial users must still pay full price for a license, but by making their systems available to home users, hobbyists and students, they are acknowledging the threat Linux poses to their systems. After all, the reason Linus started developing Linux was to make it easier for him to learn UNIX. It would seem that Sun and SCO have come to the realization that anyone wanting to learn UNIX will *not* be learning their versions unless they make them affordable.

As a side note, I have already received and installed Solaris 7. While it is a good package, I found it a little disappointing. Having used various Red Hat distributions, I found Solaris to be a rather bare-bones system. I expect this will also be the case with SCO 5.0.4 when I get a chance to experiment with it. Sun and SCO should watch out—their efforts may be too little too late.

Keep up the great work with *LJ*.

—Mark Mathews garcin@earthlink.net

VNC

Thanks to Brian Harvey for his excellent article on VNC, “Virtual Network Computing” in the February issue.

I have tried several commercial tools to allow me to maintain an NT server from my desktop (the server rooms are *cold*). At best, I have had mixed results, often serious disappointment and consequences. I work in a semi-homogeneous networking/computing environment, mostly Novell and NT, with ERP/MRP management on OS/400, Win95/NT at the desktop, and a smattering of other UNIX workstations (mostly Sun). Linux is hiding all over the place on an increasing number of “dedicated service” boxes. We don't talk about it too much, since our IS upper management is still very skeptical.

Encouraged by Mr. Harvey's comments, I tried VNC the morning I read his article. I am delighted at both the cleanness and the benign operation on the several platforms of interest to me. It is a great effort on a strong computing foundation with room to grow. What more could anyone want?

Hats off to the good folks at Olivetti & Oracle Labs for such a fine addition to the rapidly expanding Open Source universe.

—Charles Cluff charles.cluff@cwix.com

Issue 58, February 1999

LJ is to be congratulated in consistently publishing a technical journal of high quality for a diverse readership. It clearly merits being classified as a journal even though it is not published under the auspices of some professional society.

Moreover, and this bears upon the ideas of the first paragraph, *LJ* is to be thanked and applauded for including articles and editorials dealing with the social issues pertaining to open software. The February 1999 issue stands out for both the guest editorial by Alessandro Rubini (citing prior work by Russell Nelson, August 1998) and the article by Dr. Steve Mann. A journal should take on such social responsibility.

The ubiquitous computer, as no other machine invention before, has impinged upon the workings of society, for the most part to its benefit. It is necessary for the commonweal that computers be developed in the open, both to accelerate the benefits they may provide and to prohibit their misuse and the stifling of progress.

Societies make laws permitting the existence of corporations and their exclusive exploitation of inventions and intellectual property, not for the benefit of a clever elite, but for the common good. Monopolistic practices may be tempered by restrictions when they become antithetical to social welfare. The current state of computer software suggests that such change is needed.

We, as citizens, can bring about necessary changes through political action aided by open discourse and the publication of ideas.

—David E. Baker debaker@pacbell.net

Response to LTE in February

I felt compelled to respond to a letter submitted by David Briars to the *Linux Journal* editor in the February 1999 issue. I applaud David for being relatively informed on the issues of security. Yet I am disappointed at the solution that he devised. Of course, a properly configured Linux box is a safe house in regards to people breaking in. I emphasize *properly* because a default Red Hat 5.2 Linux install is extremely insecure from some of the default services running. I found out the hard way that somewhere between the included POP2, POP3 and IMAP services and the way they are configured, a significant security threat exists. I had a Linux machine on the network, accessible by the Internet for web services, and I noticed an IRC bot running illegally. All this from the most secure operating system available, in my opinion.

I learned not to blame the operating system, but to go to the source. Windows 9*x* is not insecure by default. Faulty applications (earlier versions of Internet Explorer, Netscape, etc.), malicious programming such as Back Orifice, or perhaps enabling File and Print Sharing for a personal home network but not removing the binding to the dial-up connection are examples of how a good thing goes bad.

Don't be so quick to blame the OS; be informed and stay on top of the game. As long as a human creates the code, a human can break it.

—James W. Radtke james.w.radtke@uwrf.edu

Non-X-Based Office Suite

Regarding the “Best of Technical Support” letter in issue 58, a non-X-based office suite called Cliq is available from Quadratron. Look at http://www.quad.com/linux.htm.

—George Toft LinuxAdvocate@iname.com

On-line Only Articles

You recently began having articles which are available only on-line. Could you please tell me the reasoning? I find it very annoying because I do not always get a chance or even remember to come look at the site when the next issue is available to see what you have left out of the magazine. Some of these articles are excellent and I don't see why they are not included in the magazine.

It is especially annoying as I receive my copy of the magazine about a month after it is available and so your site is usually showing a couple of issues ahead of what I am reading. Thanks.

—Sean Preston spreston@icon.co.za

We added this feature to our web site because we are very fortunate in having an excess of articles for each issue. We think they are excellent too and do not want them to go to waste because a particular issue has no space for them. If we hold on to them too long while waiting for space, they can become dated. All of these articles are listed in the magazine's Contents, so there is no need to look at the site to see “what we left out”. Also, since the Contents is put on the web site about three weeks before the magazine is shipped, these articles are available for your perusal in advance. I hope you will come to see this as an asset rather than an annoyance —Editor

Grace Hopper's Computer Bug

The January issue's article on women in technology repeats a persistent myth about Grace Hopper coining the term “bug”. Hopper herself was not present when the moth was removed from Harvard University's Mark II computer in 1947 and “First actual case of bug being found” entered in the log book. The term was popularized by Hopper's telling of the story, but was in use before then, as Hopper herself noted, and as the log entry makes clear. It was used as far back as the end of the last century, applied then to electrical equipment.

—Niall Kennedy nkennedy@acm.org

Re: Red Hat Phenomenon

In his letter in the March 1999 *Linux Journal*, Reilly Burke states that Red Hat “is unconventional in layout, difficult to install, extremely difficult to reconfigure and deficient in basic tools. The worst problem is that Red Hat requires extensive editing of C source code and rebuilding of the kernel.”

I use Red Hat Linux every day at home and work and have installed it on several machines, both Intel and Alpha-based. I don't understand Mr. Burke's complaints. While I don't have much experience with non-Red Hat flavors of Linux, I have installed and used several other operating systems, and I find Red Hat Linux easier to install than most. The base distribution contains almost every tool I have ever needed and I've never had to do extensive editing of C source. However, I have needed to recompile my kernel a few times and have had a few configuration problems, mainly due to lack of knowledge.

While Red Hat Linux does have a few warts, especially on my Alpha system, I do not agree with Mr. Burke's objections. Thank you.

—Richard Griswold richard@home.com

LinuxWorld Conference & Expo

Marjorie Richardson

Issue #61, May 1999

This was a major conference with more than just the usual suspects in attendance, and everyone had a big announcement.

Back home from LinuxWorld, the first Linux conference held on the West Coast, I am finding it difficult to concentrate and get back in the normal groove. I spent a remarkable two days, March 2 and 3, in the San Jose Convention Center and everyone who didn't go has been dropping by to find out about it. This was a major conference with more than just the usual suspects in attendance, and everyone had a big announcement.

Over 6,000 people turned out to join in the excitement. Described by one attendee as “heady stuff”, I can't think of a better way to describe it. The attendance by the big-name vendors is a sure sign that Linux has made the big leagues. When introducing Dr. Michael Cowpland's keynote speech, Jon “maddog” Hall described this conference as Linux's “coming out” or “sweet 16” party, with the business community embracing the Linux community and Linux embracing business—“Welcome to the world of Linux!” he said.

Dr. Cowpland gave an articulate speech, focusing on the ways Corel is using Linux now and in the future. While I was a bit surprised to learn the first keynote was a company presentation, it certainly gave a clear picture of how big business perceives Linux as an excellent opportunity for promoting growth and profit. Dr. Cowpland said again that Corel would be porting all their products to Linux and continuing to support the WINE project. His presentation of the Quattro Pro spreadsheet program running on WINE was quite impressive—fast and quite attractive. He announced WordPerfect Office 2000, stressing their goal of “value, performance and compatibility”, and a Corel distribution which will combine the best features from each of the current distributions and be ready for release in the fall. He predicted that by the end of the year, we will be able to buy high-performance computers, such as Gateway, for $600 to $800, preloaded with Linux. Sounds good to me.

Linus gave a well-received keynote address and participated on a panel discussion of "The Continuing Revolution", moderated by Eric Raymond. He also showed up at the Compaq booth with Jon Hall for fans to visit, take photos and get autographs.

I attended only one other talk (too much booth duty) and that one was by Larry Wall (See in this issue). Larry has a casual speaking style that fits well into this environment. Two quotes I enjoyed were:

> Perl does one thing right—it integrates all its features into one language.

> Journalists who give Perl bad press should experience more angst in their writing.

Speaker Dan Quinlan will also be appearing in our pages soon. Dan is writing a feature article for our June Standards issue.

I spent a good bit of time talking to various vendors. Here's a bit of what I found out:

- Tripwire Security Systems, Inc.: Tripwire is a file integrity security system (see "Tripping up Intruders with Tripwire" by Kevin Fenzi, August 1997) that became commercial in January of this year. All 2.*x* releases are still freely available and include the basic support of 48-hour e-mail response. They provide extended support (4-hour response) for a fee. Biggest additions are Algomol encryption for the database, damage inventories and e-mail warnings when damaged files are found. I brought back a Tripwire CD for our system administrator to play with, so expect a review in a future issue.
- GraphOn: This company is promoting their Go-Global thin PC X server software designed for high-speed access to UNIX/X-based applications on the server from any desktop. A web site has been established at http://playpen.graphon.com/, where Windows users can enjoy the experience of running Linux. Corel, a partner of GraphOn, has embedded the thin-client software in WordPerfect 8, making it web-enabled.
- ICP Vortex: This German-based company has the number one RAID controller in Europe and they have a fully bootable implementation for Linux. Drivers for their controller can be found in all major distributions. Beginning this month, they are shipping a 64-bit PCI-fibre channel RAID controller, which can also be run in 32-bit slots. ICP is committed to continuing support of the Linux operating system. One of their big users in the U.S. is Linux Hardware Solutions, and they have promised us a review of this excellent RAID device.

- Precision Insight: With funding from Red Hat and XGI, this company is creating an OpenGL 3-D infrastructure within XFree86 servers that will enable developers to access device drivers which permit access to OpenGL clients.
- Cygnus Solutions: Cygnus Solutions has had an open source business model since 1989, providing support for open source software. It is now a member of the Fortune 500. At this show, they talked to me about the cross-compilers included in their GNUPro Toolkit for Linux. The Toolkit includes all the popular GNU tools, along with added features and custom enhancements such as a graphical user interface to the tools.
- IBM demonstrated several of their products that now run on Linux, including the WebSphere product line, the Andrew File System and the DB2 database system. Also on display was the first commercial, Java-based emulator for Linux called IBM Host On-Demand. This product provides secure access to data and applications via a web browser. When I asked how it happened that IBM was entering the Linux market, the answer was “user demand”. How about that—asking for what you want truly works!
- Informix: Janet Smith of Informix graciously came by the *Linux Journal* booth to visit me while I was on booth duty. Informix has a very large presence among value added resellers (VARs) and recently formed an alliance with Hewlett-Packard to deliver Linux-based Internet solutions through their Covision program. Informix also announced an alliance with Jones Business Systems, by which Informix Linux products will be distributed through JBS' reseller channels. HP was also at the conference to show off OpenMail, but I didn't get the chance to talk to them.
- Stalker Software: Ali Liptrot of Stalker Software also came by to say “hi”. When I went by their booth, it was flooded with traffic to see the demonstrations of their CommuniGate Pro messaging system.
- Appgen Business Software, Inc.: Jim Kelly stopped by to tell me about their financial software. I had thought financial software was one place Linux was weak—looks like I was wrong.

All the major distributions were there giving away t-shirts and other goodies, and in general amazing everyone with their new releases. I saw a beta demonstration of Caldera's next release of OpenLinux which has the easiest install I've ever seen. They've provided a GUI using QT from Troll Tech, and it just zips through, probing for mouse and other information, completing the install without the user having to do a thing. It even provides a window so you can play Tetris while waiting for the install to complete. Not having a “smart” install is one item many people have said was a major drawback for Linux—well, now Linux has it. One more reason for not using Linux has just been blown away.

While doing booth duty on Wednesday, I got to meet many of our readers and authors as well as introduce new people to *Linux Journal*. I had a lot of fun. We shared our booth space with our publisher SSC, there to promote their latest book, *The Artists' Guide to the GIMP* by Michael J. Hammel. Michael was there and many fans showed up to meet him and have their books signed.

All in all, it was a great show and IDG is planning a repeat performance in August. So if you missed it, come and drop by the *LJ* booth then for a visit. Heady stuff, indeed!

Competition for photographs of and autographs from Linus was fierce

Marjorie with Craig Oda and Lonn Johnston from Pacific HITech

Donald Becker of 3M and Phil Hughes

Michael J. Hammel signs *GIMP* books

Bob Young of Red Hat chats with Phil

The *LJ* booth was slammed

New Products

Ellen Dahl

Issue #61, May 1999

Cobalt Qube 2, Power Boot 3.0, CyberCop Scanner on Linux and more.

Cobalt Qube 2

Cobalt Networks, Inc. unveiled the Cobalt Qube 2, a simple, low-cost server appliance running Linux that provides Internet connectivity, e-mail, web publishing and other network file services for small to medium-sized businesses and schools. The Qube 2 offers a high-speed 250MHz processor and up to 10.2 gigabytes of disk space for increased performance, functionality and storage. Other important features include e-mail filtering, aliases, IP Firewall security, file sharing and dialup on demand. The product can be purchased for as little as $999 US, depending on configuration options.

Contact: Cobalt Networks, Inc., 555 Ellis Street, Mountain View, CA 94043, Phone: 650-930-2500, Fax: 650-930-2501, URL: http://www.cobaltnet.com/.

Power Boot 3.0

BlueSky Innovations LLC announced the availability of Power Boot 3.0, a cost-effective and easy-to-use boot manager for PC operating systems. Power Boot allows one to boot multiple operating systems on a PC. Version 3.0 is as easy to use as 1.0 while adding more flexibility. It does not require a FAT partition. Power Boot 3.0 is available for purchase and download via the Internet. Single-user licenses cost $25 US and include free software upgrades.

Contact: BlueSky Innovations LLC, 2530 Berryessa Road, Suite 321, San Jose, CA 95132, Phone: 800-414-4268, Fax: 910-350-2937, E-mail: info@blueskyinnovations.com, URL: http://www.blueskyinnovations.com/.

CyberCop Scanner on Linux

Network Associates announced their new version of CyberCop Scanner now supports Linux. CyberCop Scanner is one of the leading network vulnerability scanners on the market and is designed to provide a high level of integrity assurance in settings where network security is a serious concern. It reliably and accurately allows a network administrator to perform vulnerability assessment. Please call for licensing and/or purchasing information.

Contact: Network Associates, Inc., 3965 Freedom Circle, Santa Clara, CA 95054, Phone: 408-988-3832, Fax: 408-970-9727, URL: http://www.networkassociates.com/.

e-smith Server & Gateway

e-smith, Inc. introduced its e-smith Server & Gateway. This open-source Internet server software transforms a PC (P133 or higher) into a Linux-based server with routing, firewall, e-mail and web servers. It is priced at $400 US and includes CD-ROM, documentation and one year of e-mail and phone support. (e-smith, Inc. was formerly known as Powerframe Internetworking.)

Contact: e-smith, Inc., 173 James Street, Ottawa, ON K1R 5M6, Canada, Phone: 613-236-0743, Fax: 613-276-0065, E-mail: info@e-smith.net, URL: http://www.e-smith.net/.

Lotus Notes and Domino

Lotus Development Corp. announced their latest version of Lotus Notes shipped in February. A Linux version of Domino, the server software that powers Notes, is on the way. The new Notes version resembles an Internet browser, so that many advanced Notes features can be used by simply pointing and clicking. The Notes client can be used for all types of e-mail, not just mail from a Lotus server, and features an instant messaging component. List price is $29 US. The new version of the Domino server has been upgraded for easier control by system administrators and migration to Notes from other messaging programs.

Contact: Lotus Development Corporation, 400 Riverpark Drive, North Reading, MA 01864, Phone: 800-343-5414 (outside US, 617-577-8500), Fax: 800-859-8369, URL: http://www.lotus.com/.

MIMER DBMS for Linux

Sysdeco Mimer AB in Uppsala, Sweden, released its MIMER DBMS for Linux. MIMER 8.1 is a complete release of the MIMER database management system. Based on the efficient and extremely easy-to-use MIMER database server, it is a scalable and portable solution for database applications. The final product release, which is a complete developer version, is available for free download from Sysdeco's web site. Full support agreements are available.

Contact: Sysdeco Mimer AB, Box 1713, SE-751 47 Uppsala, Sweden, Phone: +46-18-18-50-00, Fax: +46-18-18-51-00, E-mail: info@mimer.se, URL: http://www.mimer.com/.

PGI Workstation 3.0

The Portland Group, Inc. (PGI) announced the availability of PGI Workstation 3.0, its newly updated suite of parallel FORTRAN, C and C++ compilers and tools. All users with a current software subscription can receive release 3.0 at no additional charge. PGI Workstation 3.0 is supported on Intel-based workstations, servers and clusters running Linux and other operating systems. PGI Workstation 3.0 pricing starts at $299 US for F77-only or C/C++-only packages, and $499 US for full F90/HPF packages.

Contact: The Portland Group, 9150 SW Pioneer Ct, Suite H, Wilsonville, OR 97070, Phone: 503-682-2806, Fax: 503-682-2637, E-mail: sales@pgroup.com, URL: http://www.pgroup.com/.

DocFather Professional 2.2 and Siteforum Database Exchange

SFS Software announced the release of DocFather Professional 2.2, a fast, easy-to-navigate on-line and off-line search utility for any Internet web site or intranet web-based documentation. Site visitors can search DocFather-enhanced web sites by keyword or document site map. DocFather is able to run on any Java-supported operating system including Linux. The product can be ordered on-line from SFS Software or through its US-based partner, Proactive

International. An Internet license is $349 US, an intranet license $990 US, and a CD-ROM publishing license for 10,000 CD-ROMs is $1,990 US.

SFS Software also announced the release of Siteforum Database Exchange, a 100% pure Java solution capable of importing and exporting existing data stored in any JDBC/ODBC-compatible database into another JDBC/ODBC-compatible database (i.e., Sybase to Oracle). The software allows one to create, delete and modify tables and columns. In addition, one is able to modify content, field types and attributes. Siteforum Database Exchange runs on Java-supported operating systems, including Linux. It is priced at $495 US for a single-user license and $1,990 US for a 5-user license.

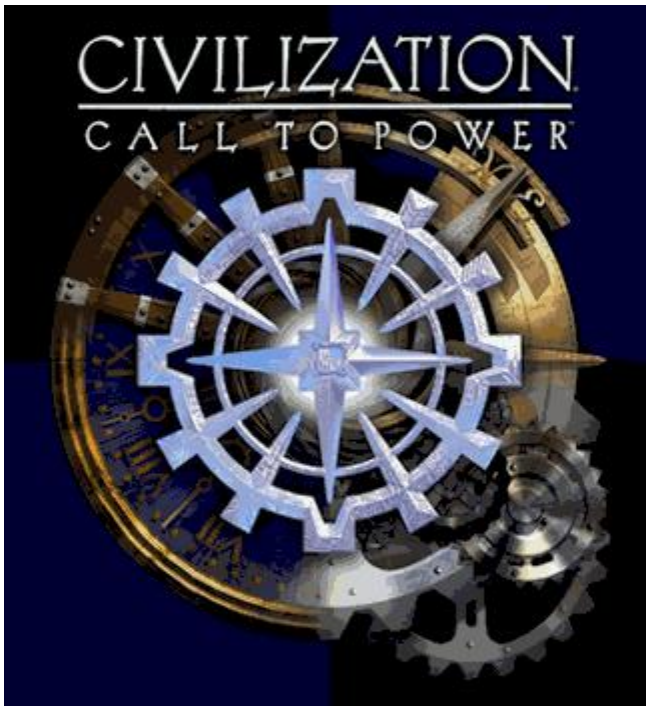
Contact: SFS Software, Allende Strasse 68, 98574 Schmalkalden, Germany, Phone: 800-332-9966, +49-3683-798-170, Fax: 888-467-1806, E-mail: sales@sfs-software.com, URL: http://www.sfs-software.com/.

Contact: Proactive International, LLC, 6107 SW Murray Blvd #151, Beaverton, OR 97008, Phone: 503-520-0191, Fax: 503-643-9877, E-mail: info@proactive-intl.com, URL: http://www.proactive-intl.com/.

Velocis Database Server v.2.1 for Linux

Raima Corporation has released Velocis Database Server version 2.1 for the Linux platform. Velocis is an embedded client/server database engine which now provides robust new interfaces for several popular development environments. Velocis 2.1 extends its SQL support with scrollable cursors and customized comparison functions and introduces a powerful new database utility, dbrepair. A free trial download is available from Raima's web site.

Contact: Raima Corporation, 4800 Columbia Center, 701 Fifth Avenue, Seattle, WA 98104, Phone: 800-327-2462, 206-515-9477, Fax: 206-748-5200, E-mail: sales@raima.com, URL: http://www.raima.com/.



Civilization: Call to Power for Linux

Under an agreement with Activision, Inc., Loki Entertainment Software announced it will port and market a Linux version of the strategy game, Civilization: Call to Power. Based on its agreement with Activision, Loki Entertainment Software is porting the game from the original source code so that the graphics, action and user interface will be the same as the PC version. Loki plans to ship the Linux version in early spring 1999. It will distribute the Linux version through retail outlets and over the Internet with pricing similar to the PC version.

Contact: Loki Entertainment Software, 17602 Seventeenth Street, Suite 102-245, Tustin, CA 92780, Phone: 888-522-5602, Fax: 714-505-3207, E-mail: info@lokigames.com, URL: http://www.lokigames.com/.

Contact: Activision, http://www.activision.com/.

RiverTools

Research Systems released the first commercial data modeling tool for terrain and river network analysis, RiverTools. It provides a valuable planning resource for use in fields such as civil engineering, insurance, construction, hydrology and land planning. RiverTools is designed to help hydrologists view and process digital elevation model (DEM) data. With pricing starting at $999 US, RiverTools is available for a wide variety of computers and operating systems, including Linux.

Contact: Research Systems, Inc., 2995 Wilderness Place, Boulder, CO 80301, Phone: 303-786-9900, Fax: 303-786-9909, URL: http://www.rsinc.com/.

DIMM-PC/486

A Stanford University professor has built the world's smallest web server using new technology from Jumptec Industrial Computers. Professor Vaughan Pratt used the DIMM-PC/486 from JUMPTec, which measures 40 x 68 mm (1.57 x 2.68 inches). DIMM-PC/486 has the same capabilities as a standard 486 PC and is much faster than a Palm Pilot. Small enough to be worn in a shirt pocket, Pratt's DIMM-PC web server sports a 66 MHz CPU, 16MB of RAM and 16MB flash ROM—big enough to hold Red Hat 5.2 Linux and run the HTTP daemon. To learn more, visit http://wearables.stanford.edu/.

Contact: EMJ Embedded Systems, 1434 Farrington Road, Apex, NC 27502, Phone: 800-436-5872, Fax: 919-363-4425, URL: http://www.emjembedded.com/.

Archive Index Issue Table of Contents

Advanced search

Best of Technical Support

Various

Issue #61, May 1999

Our experts answer your technical questions.

TELNET Permissions

How do you disable users from using TELNET to log in to a specific machine (i.e., server)? And is it possible to allow some users to TELNET to a specific machine and some not? —Ethan Bambock, ebambock@hotmail.com

You can disable TELNET in the /etc/inetd.conf file. Look at the /etc/security/access.conf file to allow access on a per-user basis. —Marc Merlin, marc@merlins.org

[See also man pages for hosts_access(5) and hosts_options(5). —Ed.]

FTP Stuck

I cannot seem to get the FTP server/service to work. When I attempt to use **ftp** to access the Red Hat machine from another computer, I get the message “connection is closed by the remote host”. It won't even give me the opportunity to type in a name or anything. I have not had this problem with previous Linux versions. I am on a Gateway Pentium 133 on a 16MB token-ring network. Incidentally, using TELNET works fine. —Steve Mitchell, mitchells@co.monterey.ca.us

There are two possible answers to this problem, depending on the FTP client you are using.

If you are using NcFTP, and you are not forcing it to ask you for a user name and password with **-u**, it will automatically try to log in as root, if you are logged in as root locally. Most distributions of Linux prevent root logins via FTP for security. You can change this by editing /etc/ftpusers, which is a list of users who may not log in via FTP.

If you are not using NcFTP, the service is either not installed or not correctly configured on the remote server. This is less likely to be the problem, and you can test it by trying to connect from another machine on the local network, as opposed to your home system. In this case, **inetd** (the service that handles incoming connections and spawns the handler daemons) is either not finding **ftpd** where it has been told to look or is not able to start the process for some reason. **inetd** is configured via /etc/inetd.conf, and you may want to look at that file to see if the FTP service is commented out. —Chad Robinson, chadr@brt.com

Sluggish Server?

For a long time I had no problems with my Linux distribution, but then I switched over to a new system, which had a K6 266MHz and 64MB of SDRAM. For the most part I have no problems, but when running certain applications, namely Emacs, NcFTP and Netscape (I am using Communicator 4.04), I find they load up too slowly and Netscape runs too slowly to use. I've heard this may be because of slow FPU speeds on the AMD but either way I want to know if there is a way to avoid this problem. I would use Lynx for web browsing except that it doesn't support a proxy connection. (My Internet access is through a proxy server.)--Derek Wollenstein, fortmax@geocities.com

It sounds to me as if Linux is not seeing all of your memory. You should verify that it is all there with the command **free**. The total column will indicate how much is recognized. If it does not show (in kilobytes) close to 64MB, you must specifically tell Linux there are 64MB. This is done at boot time, so you must edit /etc/lilo.conf and add the line **append = "mem=64M"** to the options. Then run **lilo** from the command prompt and reboot.

One other thing could cause problems. Some BIOS revisions come with an option for “Memory Hole at 15M”, which you should disable. This option is for OS/2, so unless you are running OS/2 you do not need it. —Andy Bradford, andyb@calderasystems.com

I don't know about the slowdown, but Lynx does support proxies! Look for the file lynx.cfg; it has examples of how to set up a proxy right in the comments. —David M. Brown, david@calderasystems.com

Installing New Packages

Sometimes when I am installing an RPM package, I get messages saying the package is already installed and cannot install; however, when I run the **rpm -q** to query the package, it says it is not installed. I need certain packages such as Perl that I cannot get installed and do not work. Please help! The frustration is setting in. —Carlo Wise, 141618@bellsouth.net

Try using **rpm -qa | grep perl** to list all the packages that might be installed with the name of **perl**. You can obviously change "perl" to whatever package name you are looking for. —Andy Bradford, andyb@calderasystems.com

There is some confusion at times as to the distinction between a package name and an RPM file name. There is a difference! When you wish to install an RPM, you use the RPM file name, e.g.:

```
rpm -ivh
```

*where **`filename`-2.0-1.i386.rpm** is the actual RPM file name. When you wish to reference an installed RPM, you must use the package name (with or without the version information), e.g.:*

```
rpm -q filename
```

or

```
rpm -q filename-2.0-1
```

*In this case, **rpm -q filename-2.0-1.i386.rpm** will not work, as that is not the package name. —David M. Brown, david@calderasystems.com*

More HOWTOs Needed

I have Red Hat installed, but need HOWTO information to use it. How do I get my printer, a "dumb" HP, to work? I downloaded the Ghostscript file to usrs, so what now? How do I get it to open, and in the right place? Is there a manual that is for truly dumb dummies? I have the *Linux For Dummies* book, but it skips a lot. I do not know how to get into the cc disks except to install. I have been using computers for 10 years, self-taught with books, but these books are short on HOWTOs. —Haroldel, haroldel@ix.netcom.com

In your installation CD is a User's Guide rpm which addresses a lot of questions, especially for beginners. For your printer problem (assuming that is already connected to the parallel port):

1. *Log as root and start X.*
2. *Start Print Tool from the control panel or by directly typing **printtool** at the xterm prompt.*
3. *Click on Add.*
4. *Specify the printer type (in your case, local) and click on OK.*
5. *Click on Select (next to Filter) and choose the HP model closest to yours.*
6. *Click OK.*
7. *Restart* **lpd** (under Lpd menu entry).

You should now be ready to print; you can test using the Tests item. —Mario Bittencourt, mneto@buriti.com.br

Cross-Platform E-mail

I have a requirement to dual-boot my PC (Linux/WinDoze). I would like to be able to share an e-mail mail box between the two operating systems. Other than Netscape Mail and Pine, is there an e-mail client that runs natively on both platforms and has the ability to share a common mail box? Netscape is good, but the mail filtering rules are limiting. It also handles the summary files differently between Linux and Windows. This results in the Windows summary files being seen as mail boxes in Linux. It is quite frustrating working on one platform only to realize the e-mail you need to read was retreived from the server on the other platform. —Larry Johnson, larryj@cyberramp.net

Rather then answer your questions directly, I propose an entirely different solution. When checking your e-mail, just make sure you “leave mail on server”. Most clients support this. I have set up many a corporate employee who wanted to synchronize their e-mail on a laptop with their e-mail on a desktop computer. —Mark Bishop, mark@bish.net

I recommend using an IMAP-compatible client to retrieve your e-mail. Pine, Netscape, Outlook Express and many other mail clients support the IMAP protocol. The benefit of using IMAP is that your folders are kept on the server, so your client does not need to store this information locally and attempt to share it with other clients. I do this with Pine under Linux and Outlook Express in Windows and have been quite happy with the results. Just be sure you refresh your folder lists frequently, as most clients will not do this automatically and will miss updates made in other clients. —Chad Robinson, chadr@brt.com

Problem with Boot Disks

I'm a beginner. After creating the boot disk using **rawrite** with boot.img as its source, I tried to boot using the diskette. After I pressed <ENTER>, my PC froze. Here is the last line of the message:

```
RAMDISK : Compressed image found at block 0
CRC errorVFS : Cannot open root device 08:22
kernel panic : VFS:Unable to mount rootfs on 08:22
```

Help. —Rohaimi Razali, rohaimi_raz@hotmail.com

Use a pair of brand-new floppy disks, and this problem should go away. The compressed file system placed on the root disk consumes almost all the disk, and any errors on the disk will cause this problem. Usually, replacing the disk with a fresh floppy will solve the problem. The worst-case scenario is a bad floppy drive, but that is unusual. —Chad Robinson, chadr@brt.com

Wiping Out LILO

I cannot remove LILO from my Master Boot Record. Even reformatting the drive completely back to a Windows FAT 16 configuration doesn't help. A fragment of LILO somehow remains and tries to boot a nonexistent LINUX system, denying me access to Windows, and freezing the system. How can I completely delete LILO from my MBR? —Robert Morgan, rcm612@prodigy.net

There are ways of restoring your original MBR, but since the drive has been formatted, that is not an option. Another method is to first boot from a DOS boot floppy or Win95 rescue diskette. Then run **fdisk /mbr** which will write a new MBR. —David M. Brown, david@calderasystems.com

Linux Apprentice: Improve Bash Shell Scripts Using Dialog

Mihai Bisca

Issue #61, May 1999

The dialog command enables the use of window boxes in shell scripts to make their use more interactive.

Shell scripts are text files containing commands for the shell and are frequently used to handle repetitive tasks. In order to avoid typing the same commands over and over again, we put them in a file with a few modifications, give it execute permission and run it.

To control the program at run-time, an interactive shell script is needed. For this case, the **dialog** command offers an easy way to draw text-mode colored windows. These windows can contain text boxes, message boxes or different kinds of menus. There are even ways of using input from the user to modify the script behaviour.

The current version of the dialog program is cdialog-0.9 and can be freely downloaded from Sunsite's /pub/Linux/utils/shell directory. Dialog uses the ncurses library, so it too must be installed. Some Linux distributions (i.e., Slackware) include the dialog program because of utilities which rely on it (**setup**, **pkgtool**). By the way, these utilities are great examples of using dialog.

Let's examine the dialog version of the most popular example program around. With your favorite text editor, create a file named hello containing these lines:

```
#!/bin/sh
# First shell script with "dialog"
dialog --title "Dialog message box" \
       --msgbox "\n Hello world !" 6 25
```

The first line of this file identifies it as a shell script for the “sh” shell. Every shell script must start with the characters “#!” followed by the name (and path) of the shell to execute. For example, we could have written this line as **#!/bin/bash**. The next line is just a comment, like any line starting with “#” other than the first line in the file. Then comes the dialog command, which will draw a message box 6 lines high and 25 columns wide on the screen, containing the title “Dialog message box” and the message “Hello world !”. The message box has an OK button and when it is selected, the script will end. Notice the general format of the **--msgbox** option:

```
--msgbox
```

After writing and saving this file, type:

```
chmod a+x hello
```

=“./2460f1.gif” Figure 1. Screenshot of a Dialog Box

The resulting screen is shown in Figure 1. This example is so simple it could have been produced with just one command at the shell prompt. However, things get more complicated when user input is needed in a shell script.

For example, to list the contents of a directory, use dialog as shown in **Listing 1**. This introduces two new dialog boxes: an input and a text box. The input box has the general format:

```
--inputbox
```

In Listing 1, the default value displayed in the input box is obtained by running the command **pwd** which returns the present working directory. Whenever a command is enclosed in reverse quotes, bash replaces it with its standard output.

Of course, this default value can be changed at runtime using the backspace key to delete and regular letter keys to write. The final value is printed by dialog on STDERR. In order to use it from the shell script, it must first be redirected to a file. Do this with the redirection:

```
2>/tmp/dialog.ans
```

The next line is necessary in case the user decides to select the Cancel button in the input box. When that happens, the exit status of the dialog command will be 1. Bash keeps the exit status of the last executed command in the variable **$?**, so if this is 1, the shell script will stop after clearing the screen.

If **$?** is 0 (the user clicked the OK button), the answer file is read to set the variable ANS. Again, reverse quoting proves useful. Another method of doing this is to use:

```
ANS=$(cat /tmp/dialog.ans)
```

The contents of the chosen directory are output to the same file used before. This can be done safely, because the > operator overwrites the previous contents of this file.

All is now set for the next dialog command, which generates the text box to display the contents of a text file. It has the general format:

```
   --textbox
```

The text box allows navigating with the arrow keys or home/end/pgup/pgdown keys and even has simple searching facilities. Typing / while the text box is displayed causes another window to appear, which prompts the user for a string to be searched forward in the file. Typing ? performs reverse searching, just as for the **less** pager. The first line containing the string is displayed at the top of the text box.

The experienced programmer might complain about an obvious flaw in this shell script. What if the directory name is wrong? The shell script will not complain, but will show an empty text box since there are no files in a nonexistent directory. To solve this problem, a check is made to see if the specified directory exists. Actually, the **ls** command returns an exit status of 0 if the directory exists, and 1 if it doesn't. Thus, the script can be modified by adding these lines:

```
   ls -al $ANS > /tmp/dialog.ans 2>/dev/null
   if [ $? = 1 ]; then
      clear
      echo no such directory
      exit 1
   fi
```

First, the ls line is changed, redirecting standard error to /dev/null. This means no error messages from ls will appear on the screen. Then, if the exit status (**$?**) is 1, the script will exit with an error message.

This script can be made even more useful by allowing the user to examine more directories before the script exits. (See Listing 2.) A few changes have been made. First, the entire script has been included in a **while-do** loop which is always true. This allows it to run more than once. Now the only way of exiting the script (besides typing ctrl-c) is by selecting the Cancel button in the dialog input box. The second change is the introduction of a message box which will be displayed when the ls command returns an exit status of 1. The command **continue** deserves a special comment. Its meaning is to skip the current iteration of the **while** loop (i.e., the part which shows the text box) and start a new one. Thus, after the error message, the user will again see the input box, prompting for another directory name.

The menu box is produced by running dialog with the **--menu** option with the format:

```
dialog --menu
   tag2 item2...
```

This option displays a box with two buttons (OK and Cancel) and a menu consisting of one or more lines. Each line has a "tag" (a number or word) and an "item", which is usually text describing the menu entry. When a user selects an item and then clicks on the OK button, the corresponding tag is printed on STDERR. Also, the exit status of dialog is 0 for the OK button being selected and 1 for the Cancel button.

Menu boxes are useful in that they allow the user to choose from several fixed alternatives. For example, when producing a LaTeX document, three steps must be taken: editing the source file with a word processor, compiling it with LaTeX and viewing the resulting .dvi file. It is easy to build a shell script to do these steps. (See **Listing 3** which assumes the text editor is **jed**, the .dvi file viewer is **dvisvga** and both are in the path.) The complete script is again included in a "while" loop for the purpose of making it work more than one time. The only way to exit this script is by selecting the "Cancel" button in the first menu box. Otherwise, the user has to choose between three alternatives:

- Edit a text file.
- Compile a LaTeX file.
- View a .dvi file.

The answer is stored in the file /tmp/ans and retrieved in the variable **R**. If the user chooses to edit a file, a new dialog box appears. It is an input box and prompts for a file name. The answer goes into the variable **F**. Then the script checks whether the file exists and runs the command:

```
jed $F # where $F is the name of the file
```

If the file does not exist, it is either a new one or a typing error. To distinguish between these two possibilities, a yes/no dialog box is provided. The general format of such a box is:

```
--yesno
```

The box has two buttons, YES and NO. The text is usually a question, which the user answers by selecting a button. If YES, **$?** (the exit status of the dialog command) is 0; if NO, **$?** is 1.

In Listing 3, if the answer is YES, the text editor is invoked; if NO, the script returns to the main menu through the continue command. The other two choices work in the same way, the only difference being the commands for processing the file with LaTeX or for viewing the resulting DVI file:

```
latex $F
dvisvga $F
```

Several other dialog boxes are available, such as the checklist or the radiolist; however, their use is quite similar to that of the menu box.

I would like to end with an example of the **--guage** dialog box. This is used to graphically display a percentage. The syntax is:

```
dialog --guage
```

Once started, the guage box keeps reading percent values from STDIN until an EOF is reached and changes the display accordingly. Here is a simple (but not very useful) guage script:

```
#!/bin/bash
{ for I in 10 20 30 40 50 60 70 80 90 \
      80 70 60 50 40 30 20 10 0; do
   echo $I
   sleep 1
done
echo; } | dialog --guage "A guage demo" 6 70 0
```

Copy this into a file, give it execute permission, run and enjoy! The first part of the script (included in braces) is a group command. Every second it sends one of the listed values to the guage dialog box. The final echo command is used to terminate the dialog box.

Shell scripting is a convenient way of making your Linux system “smarter”. These examples of the most common dialog boxes should help you make your scripts more attractive.

Resources

Mihai Bisca is an ophthalmologist who is crazy about Linux. In 1998 he published the first Romanian introductory book on Linux. Currently, he is competing with his three-year-old daughter Andra for the place at the keyboard. You can reach them at mbasca@ottonel.pub.ro.

Archive Index Issue Table of Contents

Advanced search

Advanced search

A Standard for Application Starters

Rui Anastacio

Issue #61, May 1999

Mr. Anastacio demonstrates how to write an aplication starter in a standard format.

Most X users start applications from an X terminal. To do this, you must know the names of the programs, pass parameters each time you run them and include the programs in your search path.

Instead of calling applications from the terminal, you can use an application starter, a program that shows a list or menu of installed applications and lets you choose the one to start. Some starters show pretty icons and are very appealing with features like clocks, load meters, etc.

The problem is that each starter has a different way of describing the list of installed applications. Usually, this information is written in a text file in some format. For example, the starter of FVWM reads the .fvwmrc file for this information. Other window managers (WM) use different formats and files. If you use various WMs, things can get a bit messy.

Creating a standard format, location and name for application starters simplifies the process of creating, changing and exchanging information. Another advantage is in program installation. The installation process can read this file (open format and location) and automatically add the necessary entries to access the installed components. For example, when installing StarOffice, it would be nice if a group called StarOffice was created automatically with scalc, swrite and the rest.

This article proposes a standard format and a standard location for this application starter file and presents QStart, a starter which I have written (using the Qt Toolkit) in this format.

The File Format

The file, plain ASCII, consists of two parts: configuration parameters and menu definition.

The first part is used to define parameters, such as the directory in which icons are located. Each line starts with a reserved keyword, followed by the necessary parameters separated by a semicolon:

```
ReservedWord param1;param2;...;paramN
```

Only one standard reserved word, **IconDir** ***`dir`***, is defined in which ***`dir`*** is the directory where the starter searches for the icons referred to in the Menu Definition.

Other reserved words can be added for different starters. For QStart, I have defined one more word (see next section).

As an example, the next lines can be used to configure QStart to search the icons in /usr/local/icons and place the button, which pops up the menu, at position 0,0 of the screen.

```
IconDir /usr/local/icons
Position 0;0
```

To avoid future problems, consider carefully whether to add new reserved words. My idea with standard words is that these are words which are absolutely necessary to any application starter.

As new starters arrive, new words will appear. It might be a good idea to use generic words. Here are some ideas:

- **ConfigFile** ***`file`***: define a specific file for extra, specific configuration.
- **Show** ***`elem1`*;*`elem2`*;...:** show a clock or the work areas in the starter, for example **Show Clock;WorkArea;IconsOnly**.
- **Style** ***`style`***: use a different style to show the list of applications, for example **Style Modern**.

In order to keep track of future development, I have mounted a site at http://w3.ualg.pt/~ranasta/starter/ to centralize all related information.

The second part is the menu definition with the same syntax: a reserved word at the start of the line, followed by the parameters separated by a semicolon. All the applications are defined inside groups or menus. The main menu has the name “Main” and is the starting point. The menu name, or id, should be interpreted in a case insensitive manner; that is, writing “Main”, “MAIN” or “main” should have the same effect. The reserved words should be interpreted in the same manner. Starting and ending a menu definition are the reserved words **Menu** and **End**.

```
Menu
```

Here, ***`id`*** is the name of the menu for internal identification; the menu with an *id* of **main** is the starting point. ***`title`*** is the title of the menu and ***`icon`*** is the icon file name.

Between **Menu** and **End** are menu items:

- **Separator**: draw a separator, normally a horizontal line.
- **Text *`text`***: draw the ***`text`***.
- **Image *`filename`***: draw the image stored in ***`filename`***.
- **Program *`text;icon;command`***: an application. ***`text`*** is the text that appears on the menu entry; ***`icon`*** is the associated icon; ***`command`*** is the command invoked when this option is called.
- **SubMenu *`id`***: an entry point to menu ***`id`***. The title and icon of the menu ***`id`*** should appear as data to this entry.

Two examples of **Menu** blocks are shown here:

```
Menu Main;Applications;apps.xbm
 Program Terminal;xterm.xbm;xterm
 Program Editor;edit.xmb;nedit
 Separator
 Text Groups
 SubMenu Graphics
End
Menu Graphics;Graphics;graph.xbm
 Text Image
 Program GIMP;gimp.xbm;gimp
 Program Paint;paint.xbm;paint
 Separator
 Text Draw
 Program tgif;tgif.xbm;tgif
End
```

The File Name and Location

In order for programs to know where to look for this file, it must have a standard name and location. The name is .apps and the location is found in this way. First, the home directory is searched so that different users can have different configurations. Next, the system directory /usr/local is searched. This is the default configuration for all users, and can be managed by the system administrator.

QStart

Most of my experience in GUI programming has been with Motif, Xforms and TclTk. To write Qstart, I chose to use QT because it is available for many platforms and is a powerful toolkit. Also, by choosing QT, I got to learn something new.

QStart reads the .apps file from the standard location. The icon of the main menu is displayed on-screen at the position indicated by the reserved word “Position” as a button. When you press this button with the left mouse button, the applications pop-up menu will appear and the list is shown. (See Figure 1.) Pressing the right mouse button pops up a configuration menu. This menu has the options Quit and Restart. Quit does just that; Restart runs the QStart program (have it in your path) and then quits. These are useful options when you make changes to the .apps file; calling restart automatically updates the applications list.

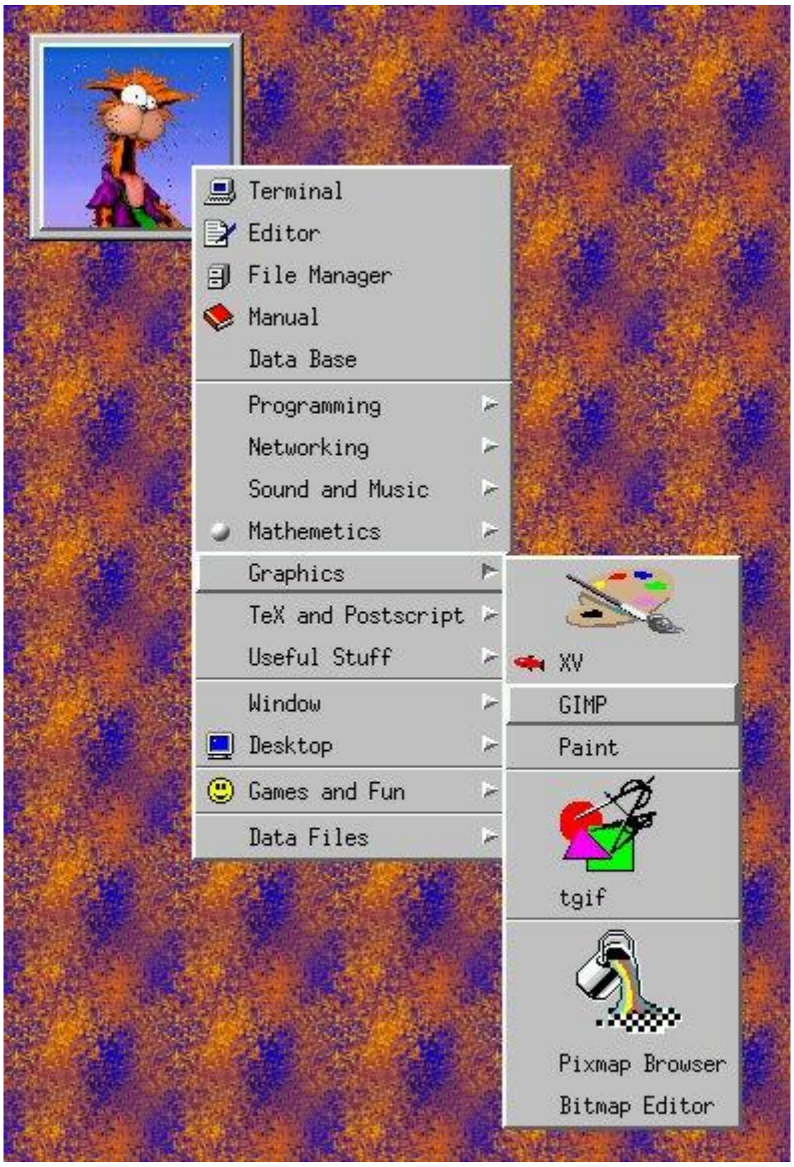


Figure 1. Applications Menu

QStart defines the following reserved word: **Position *x;y***. This uses ***x*,*y*** as the position on the screen for the button which pops up the menu.

Qstart can be found in the archive file http://w3.ualg.pt/~ranasta/starter/qstart/qstart-1.0.tgz. This includes both binaries and source. Anyone interested is encouraged to use this code to build better starters.

The Future

Here are some points to think about for the future:

- A stable format for this file
- New starters built using this format—prettier, more efficient, etc.
- Installation programs which automatically add entries for the installed components
- Creation of a set of routines to facilitate the installation programs finding, changing, adding and deleting entries in the file
- Support of this format in existing window managers

The future is unknown, but we can shape it or at least give it a try.

Rui Anastácio is currently teaching mathematics and computer science at Escola Superior de Tecnologia—Universidade do Algarve and uses Linux for part of his work. His free time is spent in sports such as tennis, table tennis, bicycling, jogging, kung fu, swimming and others. He also likes good books, music, photography, traveling and programming. He can be reached via e-mail at ranasta@ualg.pt.

Archive Index Issue Table of Contents

Advanced search

Linux at California University

Jeremy Dinsel

Issue #61, May 1999

Mr. Dinsel tells us how his former college is using Linux as a web server and teaching tool.

Nestled along the banks of the Monongahela river of western Pennsylvania rests California University of Pennsylvania, a college engulfed in the world of Microsoft Windows NT, Windows 3.11 and the old age of Vax. Yet new hopes arise for a productive network of powerful servers and real world applications that don't cause the budget to keel over and die.

Until recently, the Math and Computer Science department was without its own server. The faculty had to rely on other departments for a storage utility for software. Methods for teaching classes about the World Wide Web and the Internet were limited to the amount of help other departments could offer. Creating and maintaining departmental web pages was a difficult task—creation was done on an individual's machine, copied to disk and given to the public relations office for uploading. These tasks were not only tedious, but also inconsistently available and relied heavily on the schedule of other departments.

Several solutions were available for the department. It could deal with the situation as it was, or sacrifice a good chunk of the budget by acquiring an NT server. As politics sometimes dictate, the University departments and staff are governed by a guideline or trend in the computing industry where the only options visible to them are the ones being put into place in other locations on campus. With NT on the rise throughout the campus, the department felt that changes would have to wait, as the money for a machine capable of running NT (not to mention added software costs to make it perform the required tasks) was not in the budget.

Linux had been taking the world, and the author, by storm for a few years. Unfortunately, the computing community is sometimes blind to inexpensive alternatives such as a free operating system. For the last few years, new software has meant new hardware because programs were getting larger and more memory intensive. The consumer has been eased into accepting this as a fact of life, so much so that when they aren't required to spend money on upgrading, they feel uncomfortable. Convincing people that "free" is not a sign of poor quality can be quite a challenge.

Solutions

After months of discussions and meetings, the Math and Computer Science department and computer center agreed to give Linux a try. Even though Linux had appeared in other locations on campus, the department heads were still reluctant. Fortunately, a mid-grade 486 machine was available on the department floor. Because of the availability of an "older" machine, and the excellent quality of Linux software, the budget was not affected since there was no need for a a hardware upgrade. Linux is content to run on yesterday's hardware.

As a stepping stone into the world of non-traditional computing, a web server was set up initially for use by the department. Because of strict computing policies and the thin ice the project started out on, only faculty members were given accounts on the server.

The department now had a place to hold their web pages, that also allowed easy access for updating and maintaining them. CGI, server side includes (SSI) and other web page tricks could be utilized by the departmental web pages, because the department now had control over security on the server. Previously, security policies were defined by outside forces, and web pages were limited to basic hypertext tags.

Linux was proving its worth. The uptime on the server had surpassed the typical operating cycle on the NT servers. With continued discussion and meetings, other possible uses of the server became apparent.

The university offers a course that introduces students to the World Wide Web, the Internet and Windows (typically 3.11 and NT). Before the existence of the Linux server, the department was limited by its dependency on other departments, and so could not offer a very well-rounded view of the topics. Students were limited in what they could do with their web pages, and Internet/ Intranet working skills were weak.

Using the Linux server, anonymous FTP (uploading and downloading) became available. The lab assignments could now be placed on the server for students to retrieve individually. Essential software packages (that have a tendency to be broken or removed from the workstations) could be placed in a Samba directory for students or professors to access and reinstall.

With the advent of Linux, labs ran more smoothly and students had the potential to learn more because of the dynamic and powerful tools Linux has to offer. At the same time, campus policies regarding student web pages and computer security could be upheld. The use of TCP wrappers, **httpd** configuration and IP firewalling can limit access to web pages, shares, FTP and more. Thus, students' access to the server was limited to inside the University domain.

All of these tasks were accomplished without weeks of waiting for another department to find the time to implement requests from the Math and Computer Science department, and without tying up valuable space on the campus NT servers. Linux has proven its worth and reliability to the department: new ideas and uses for the server are discovered and pursued weekly. Most of all, the fear of not being able to afford software solutions has disappeared.

Building a place for Linux on campus has been an ongoing struggle. With persistence, and the backing of the Linux community (through excellent software and boundless innovation), there is no stopping the powerful force of Linux.

Jeremy Dinsel is a former student at California University of Pennsylvania, where he studied Computer Science and operated the Math and Computer Science Linux server. He welcomes questions and comments and encourages western Pennsylvanians to join wplug—a Linux organization (http://sighsy.cup.edu/~dinselj/wplug/). He is now webmaster for SSC and can be reached at info@linuxjournal.com.

A Look at the Buffer-Overflow Hack

Eddie Harari

Issue #61, May 1999

Mr. Harari disects the buffer-overflow hack, thereby giving us the necessary information to avoid this problem.

The best system administrator is not always enough to take care of site security. Sometimes a nice program such as **mount** can be exploited by a user to gain a higher system permission or remote access to an unauthorized location on the World Wide Web.

This article explains the logic behind a popular hack to exploit a program's code so it executes different code then was intended. This hack is known as the buffer-overflow hack and can be used to exploit a program with **suid** set to gain better permissions on a Linux machine—sometimes even root or remote access. (The examples are taken from “aleph-one” with his permission and have been somewhat modified by me.)

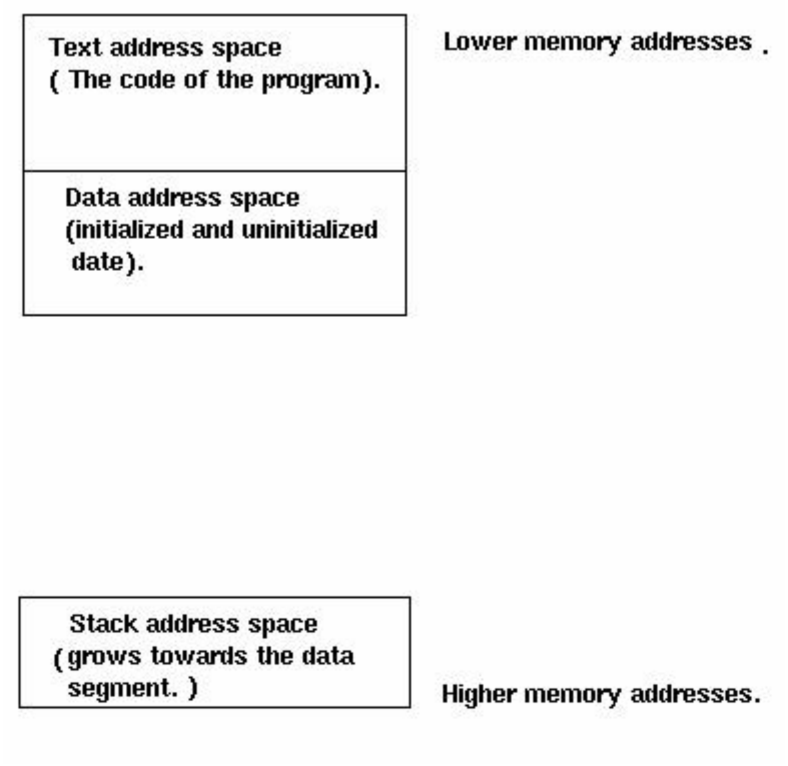


Figure 1. Virtual Memory Layout

First, let's have a look at Figure 1 and see how a process organizes its virtual memory. The TEXT area is where the actual code of the program resides. The DATA area is where the initialized and uninitialized data of the program resides.

The STACK area is a dynamic area which becomes bigger as data is pushed into it and smaller as data is popped from it. It is called a stack because it works in the LIFO way (last in, first out). The stack is used to hold temporary data for the process and helps the processor in its implementation of high-level functional programming. To understand exactly how the processor makes use of the stack, look at the following example:

```
void func(int a, int b)
{
  /* This function does nothing */
}
main()
{
 int num1;
 int num2;
 func(num1,num2);
 printf("This is the next instruction after " .
   "the function ...");
}
```

The instructions of the **main** function are executed until the processor needs to “break” the normal flow of the program and go to the **func** instructions. When this step of “jumping” to func is executed, the parameters to func, **num1** and **num2** are transferred with the help of the stack. That is, they are pushed to the stack, and func can pop them from the stack and use them. Immediately after pushing these values on the stack, main should push the address to which func will return on completion. (In our example, this is the address of the **printf** instruction.) When func is finished, it knows to read this return address from the stack and go back to the “normal” flow of the program.

One other value on the stack is called a frame-pointer, since the processor refers to values on the stack by their offset from the stack pointer (SP). Whenever the SP value changes, the processor saves the current value on the stack. (The Intel does not have a dedicated frame pointer (FP), so it does it with the help of the **ebp** register.) The frame pointer is pushed to the stack following the return address.

To clarify this, let's look at another example:

```
void func(int a, int b)
{
 int *p;
}
main()
{
 int num;
 num = 0;
 func(num);
 num = 1;
 printf("num is now %d \n",num);
}
```

Let's compile it with the **-S** option to get assembly output using this **gcc** command:

```
gcc -S -o ex2.S ex2.c
```

We see that main's code is actually:

```
main:
pushl %ebp
movl %esp,%ebp    /* Save the SP before changing
                   * its value */
subl $4,%esp      /* SP should subtract 4 so it
                   * points to num on the stack */
movl $0,-4(%ebp)  /* Push num on the stack with
                   * value 0*/
pushl $2          /* Push 2 on the stack*/
pushl $1          /* Push 1 on the stack*/
call func         /* Push return address on the
                   * stack and jump to the first
                   * instruction of func*/
...
```

The main code pushes the arguments for func, then calls it. The call instruction puts the return address on the stack, then moves on to the func code. **func** puts the four-byte frame pointer immediately following the return address, then pushes the **p** pointer onto the stack. Thus, if we dump the stack's status now, we get the structure shown in Figure 2.

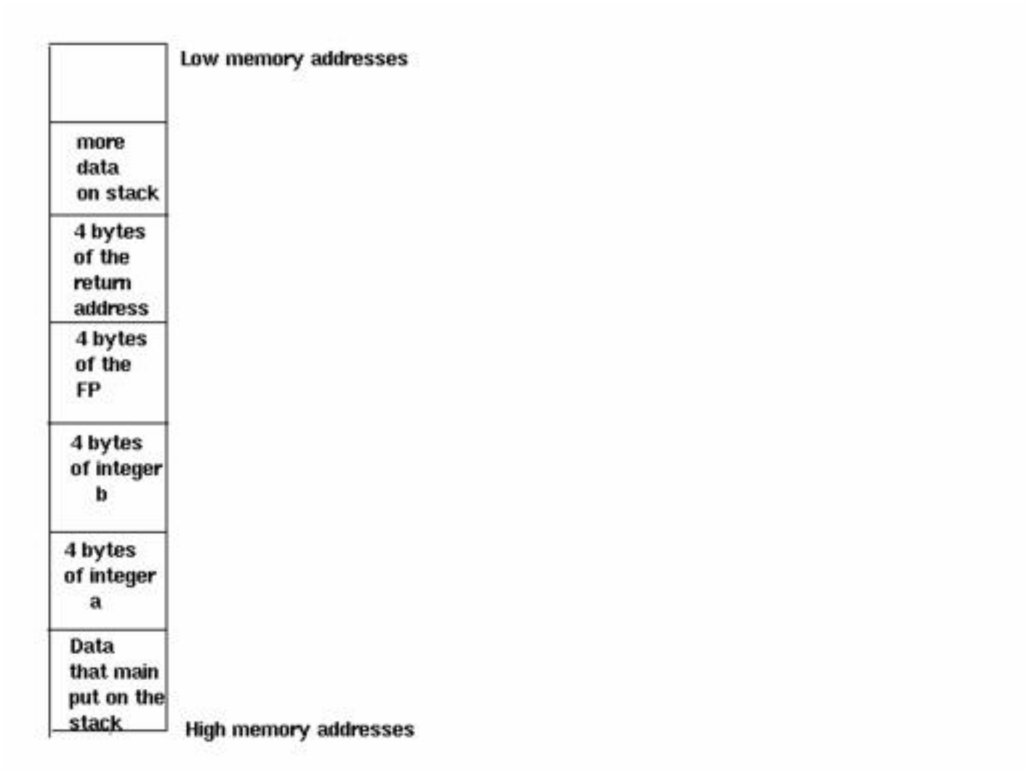


Figure 2. Stack Structure

We can use func to print the addresses of **a** and **b** in a hexadecimal format; to do this, we simply add printf instructions:

```
void func(int a, int b)
{
 int *p;
 printf("The address of a on the stack is %x\n",
   &a);
 printf("The address of b on the stack is %x\n",
   &b);
}
```

When we run the modified program, we get the following output:

```
The address of a on the stack is bffff7ac
The address of b on the stack is bffff7b0
```

Integer **b** is four bytes from integer **a**. Looking at Figure 2, we see that integer **b** is followed by the four-byte frame pointer, then the four-byte return address.

We can look at the return address using the **disassemble** option of **gdb**. (See **Listing 1**.) The call instruction in **<main+17>** is at address 0x80484b1, which means the next instruction in 0x80484b6 is the return address. As we just calculated, when this address is pushed on the stack, it is offset eight bytes from **b** and 12 bytes from **a**.

Since the stack is writable, we can use the pointer to the return address, then change its value. By doing so, we manipulate the normal flow of the program so we can, for example, skip some instructions. In **Listing 2**, we have changed the return address so our program skips an instruction. Compile and execute:

```
gcc -o ex4 ex4.c
ex4
```

This output is returned:

```
The return address is 80484d2
The new return address is 80484dc
Num is now 0
```

In the Listing 2 code, we point to the address of integer **b** with the help of a pointer **p**, then subtract eight bytes down from **p** so it points at the return address printed in the first output line. Next, we add ten bytes to the return address, so it skips the **num=1;** assembly code. (**disassemble main** shows the exact offset of the instructions, so I used it to know how many bytes to skip.)

In this way, a programmer can regulate the normal flow of his program from within. The big question is, can someone change this return address from the outside? The answer is *sometimes*. Not only can this address be changed, but it can also be changed to point to code not within the program.

Listing 3 is a very simple program that can be exploited from the outside. On first execution, the output looks like this:

```
bash# ex5
Please enter your input string:
short
This is the next instruction
```

On second execution, the output is:

```
bash# ex5
Please enter your input string:
long string
This is the next instruction
Segmentation fault (core dumped)
```

Since **strcpy** does not check the length of the string it copies, we inserted the 12-byte string **long string\n** to a buffer which is eight bytes long. The first eight characters from my input completely filled the buffer, then the remaining four characters *overflowed* the buffer. That is, these four characters overwrote the adjacent address in the buffer --the return address. Thus, when func tried to go back to main, a segmentation fault occurred, since the return address contained the four-character string **ing\n**, most likely an illegal memory address.

The strcpy function is the classical example for buffer overflow since it does not check the copied string size to ensure it is within the buffer limits. Note strcpy is not the only way to exploit a program with a buffer-overflow hack.

The actual buffer-overflow hack works like this:

- Find code with overflow potential.
- Put the code to be executed in the buffer, i.e., on the stack.
- Point the return address to the same code you have just put on the stack.

Since this is not the Linux “hack.HOWTO”, I will not go into details on these three stages.

The first stage is very easy, especially in a Linux system, since a huge amount of open-source code applications are available for Linux. Some of these applications are in use on almost every Linux system. Good examples of such programs were **mount** and some early versions of **innd**. **mount** did not check the length of the command-line arguments the user entered and its permissions set to 4555. **innd** did not check all of the news message headers, so by sending a specific header, a user could get a remote shell on the server.

The second stage has two parts. The first one is to find how to represent the code to be executed; this can be done using a simple disassembler. The second part depends on where the program reads the buffer: in some cases, a mail header; in others, an environment variable whose length goes unchecked; in still others, some alternate means.

The third stage is not so simple, as one cannot know the exact address of the code to be executed. Basically, it is done by guessing the address until the correct address is found. Several ways can be used to make this guessing more efficient; thus, after only a few guesses, we can specify the right address and the code gets executed.

Conclusions

The fact that an application is used all over the Web does not mean it is secure, so take care when installing a new application on your machine. In fact, WWW applications are more likely to be searched deeply for security holes by crackers with bad intent. System administrators should read the security newsgroup and related web pages in order to keep applications known to have security holes off the system and to upgrade them when patches become available. Application programmers should take care to write tight code containing proper checks for array and variable lengths in order to foil this type of hack.

Finally, I would like to briefly mention three other things. One, a kernel patch is available that makes the stack memory area a non-executable one. I have never tested it, since applications do exist which count on the fact that the stack is executable, and these applications will most likely have problems with this patched kernel. Two, a special mode to the Intel processor is available that has the stack grow from the lower memory addresses to the higher memory addresses, thus making a buffer overflow almost impossible. Three, a set of libraries available on some systems helps the programmer write code with no such errors. All the programmer has to do is tell the library functions the assumptions about a variable and these functions will verify that the variable meets the specified criteria.

Eddie Harari works for Sela Systems in Israel as a lecturer and a consultant. He is currently involved in networking security projects and can be reached via e-mail at eddie@sela.co.il.

Archive Index Issue Table of Contents

Advanced search

Memory Access Error Checkers

Cesare Pizzi

Issue #61, May 1999

A look at three programs designed to help the C programmer find the cause of segmentation fault errors.

All C programmers have seen, at least once, the horrible words “Segmentation fault—Core dumped” after a run of their latest creation. Usually, this message is due to errors in the memory management. (As all C programmers know, this language does not care about bounds or limit when accessing memory.) In this article, I plan to compare three products used to track down this kind of error:

- Checker 0.9.9.1
- Electric Fence 2.0.5
- Mem-Test 0.10

I will explain how to use these three different products, using small C code examples containing very common errors. I will show how (and if) each product detects the errors. All three packages replace the usual memory accessing functions with their own code and can detect memory problems when they appear. I performed the tests on my Pentium 133 Linux box with 32MB RAM and the 2.0.34 kernel.

A Bit About Installation

Checker comes in the usual tgz format (gzipped tar file), with a simple installation procedure. Run the “configure” script, then **make** all the files. The installation went fine for me; I didn't see any problems. One note: you need **gcc** 2.8.1 to use the latest version of Checker.

Electric Fence is available in binary and source format and requires kernel 1.1.83 or higher.

Mem-Test is available in the tgz format, and it is very simple to build using the provided Makefile.

How They Work

Electric Fence (EF) is a library—link it to your program, then run it. EF will cause a segmentation fault on the exact line of the wrong instruction (not 100 lines after), so by tracking the program with a debugger, you can get to the root of the problem. Place an inaccessible memory page after (or before, by using the correct option) each area allocated by your program, and EF will cause an immediate error when the program goes out of the bounds.

Mem-Test is another library you can simply link to your object—just remember to include the header file mem_test_user.h before. As we will see, this program is a bit different from the other two, and it detects particular errors. When the program runs, it creates a log where it stores all memory allocation/ deallocation. By using a Perl script provided in the package, it will show you the memory leak present in the code. Since Electric Fence doesn't detect this particular error, it can be used in conjunction with Mem-Test.

Checker is also a library and exploits the **-fcheck-memory-usage** option of **gcc**. A different compiler is actually used to build your program: **checkergcc**. It is a stub which calls gcc and compiles the program with its own memory access libraries. Once the program is compiled, you can run it and checker will show you a complete report with the errors it found in your sources. Checker uses a bitmap to store any memory area the program is using. This bitmap will contain the access right of each memory area. For example, an area could be write-only (when the variable is not yet initialized), readable and writable, not accessible and so on. In this way, it will be able to detect the memory access error.

Example Code

The six pieces of C code we will look at are:

- **postr.c**: this code (Listing 1) performs a read (with **printf**) of an uninitialized memory area. The lack of the string terminator (\0) will force the printf to read after the **malloc** area.
- **prer.c**: this piece of code (Listing 2) contains two errors. The printf is accessing a byte before the allocated area (the pointer was decremented), then the **free** is done with an address not returned by malloc.
- **postw.c**: in this code (Listing 3), the **strcpy** is writing 12 bytes (with the \0) in a 10-byte area. Moreover, the printf is reading the uninitialized last two bytes.

- **prew.c**: this code (Listing 4) is writing before the allocated memory. The free and the printf will cause an error as in the previous examples.
- **uninit.c**: this code (Listing 5) makes an assignment to a NULL pointer. This is a common error for programmers new to the C language.
- **unfree.c**: in this example (Listing 6), I missed freeing some allocated memory.

Post-Read

To test Checker, I compiled **Listing 1** with this command line:

```
checkergcc -o postr postr.c
```

All the gcc command-line options can be used with Checker. The compilation went fine, and when I ran **postr**, I got this output:

```
From Checker (pid:00411): (ruh) read uninitialized byte(s) in a block.
When Reading 5 byte(s) at address 0x0805ce1c, inside the heap (sbrk).
0 byte(s) into a block (start: 0x805ce1c, length: 10, mdesc: 0x0).
The block was allocated from:
  pc=0x08054e2b in chkr_malloc at stubs-malloc.c:52
  pc=0x08048812 in main at postr.c:10
  pc=0x08054ee1 in this_main at stubs-main.c:14
  pc=0x0804875a in *unknown* at *unknown*:0
Stack frames are:
  pc=0x08054ebf in chkr_stub_printf at stubs-stdio.c:54
  pc=0x080489f1 in main at postr.c:17
  pc=0x08054ee1 in this_main at stubs-main.c:14
  pc=0x0804875a in *unknown* at *unknown*:0
exa
```

Checker executed the program and found the problem—an uninitialized read at line 17 (the printf line). This was caused by the lack of a string terminator in the memory area. At first look, this output seems quite messy, but if you read it carefully, you will find a lot of information: which type of error it found, where the memory was allocated (line 10) and where the problem occurs (line 17).

To compile the program with Mem-Test, you must perform a slight modification to the postr source. Add header file (**#include "mem_test_user.h"**) to wrap the various memory allocation functions and use a modified version. Compile the program with the command:

```
gcc -o postr postr.c -lmem_test
```

I added another library (mem_test) to the compilation command. When you run the postr executable, the new library will create a file named MEM_TEST_FILE in which all memory accesses and leaks will be logged. In this particular situation, Mem-Test does not find a problem because it was built to identify only memory leaks.

For Electric Fence, we need to recompile the program, including the reference library:

```
gcc -g -o postr postr.c -lefence
```

I added the **-g** option to include the debugging information in the executable. This is needed because EF will cause a segmentation fault exactly at the buggy line, so you will need to walk through the code to find the exact line causing the problem. This is the output of the executable:

```
Electric Fence 2.0.5 Copyright (C) 1987-1995 Bruce Perens. exa
```

EF didn't find any problem in the code, so no errors were generated.

EF has four different switches, which can be enabled by setting one of these environment variables: **EF_ALIGNMENT**, **EF_PROTECT_BELOW**, **EF_PROTECT_FREE** or **EF_ALLOW_MALLOC_0**.

EF_ALIGNMENT sets the alignment for each memory allocation done by malloc (or calloc and realloc). By default, this size is set to **sizeof(int)**, because this is usually the alignment required by the CPU. This could be a problem when you allocate a size that is not a multiple of the word size. Since the inaccessible page must be set to word-aligned address, you have a hole after the allocated memory to the inaccessible page. You can fix this by setting the environment variable to 0; in this way, you will be able to find a single-byte overrun. This will force malloc to return a non-aligned address, but this is not a problem in most cases. In some cases (when you have an odd-size allocation for an object that must be word-aligned), you will get a bus error (SIGBUS). I never saw a SIGBUS error using EF (and I used it in real-life programs); I got this information from the EF documentation.

EF will usually place the unaccessible page after each memory allocation. By setting **EF_PROTECT_BELOW** to 1, it will place this page before the allocation, so you can check for under-runs.

EF allows you to allocate freed memory. If you think your program is touching free memory, set **EF_PROTECT_FREE** to 1. EF will not reallocate any freed memory, and any access will be detected.

A malloc call with zero bytes is considered an error. If you need to use such a call, you can tell EF to ignore this error by making **EF_ALLOW_MALLOC_0** non-zero.

I set **EF_ALIGNMENT** to 0 in order to see if the postr error would be detected, but again EF did not see it.

Pre-Read

Checker found the problem at the correct line (printf) in Listing 2, and it pointed out the freeing of an address different from the one returned by malloc. Actually, I decremented the **foo** pointer and tried to free this address.

Mem-Test didn't find the problem, but this was expected.

If I link the EF library without specifying any switch, Electric Fence returns only an error regarding the freeing of a non-malloc returned value:

```
Electric Fence 2.0.5 Copyright (C) 1987-1995 Bruce Perens.
ElectricFence Aborting: free(400b3ff3): address not from malloc().
Illegal Instruction (core dumped)
```

Then, I tried to setting the protect below switch:

```
export EF_PROTECT_BELOW=1
```

With this variable, EF caused a segmentation fault. With gdb, I tracked down the program to the printf where the segmentation fault occurred.

Post-Write

For Listing 3, Checker found the bound violation at line 14 (strcpy). Moreover, it also found an uninitialized data read at line 16 (printf). Actually, the print is going to read after the allocated area.

Mem-Test gave no reports as expected.

Again, the first run (without any switch) of Electric Fence did not report any error. I then set **EF_ALIGNMENT** to 0, and the strcpy caused a core dump.

Pre-Write

The error in Listing 4 was correctly detected by Checker as the incorrect free. Mem-Test gave no reports. When I didn't set a switch, Electric Fence detected only the wrong free, but with **EF_PROTECT_BELOW** set, it also found the pre-write.

Uninitialized Pointer

Checker found the exact line in Listing 5 where the bad assignment was made. No reports were expected or created by Mem-Test. There was a core dump, but the log was not created. Electric Fence did not detect this error. When you run the program, you will get a core dump whether you use EF or not.

Unfreed Memory

By default, Checker does not find memory leaks. The documentation shows several switches you can set to modify the checking. Different switches are set by defining the environment variable **CHECKEROPTS**. The more interesting options are:

- **-o=*file***: redirect the output to a file.
- **-d=*xx***: disable a type of memory access error.
- ***-D=end***: do leak detection at the end of the program.
- ***-m=x***: define the behavior for a malloc(0).

I ran **export CHECKEROPTS="-D=end"** and then recompiled. Now it found the memory leak of 50 bytes in **Listing 6**. Checker implements a garbage detector to find out this type of error. You can call it by setting this option or by calling a specific Checker function inside your program.

Mem-Test easily identifies the memory leak, with a clear report:

```
50 bytes of memory left allocated at 134524624
134524624 was last touched by licalloc at line 12 of unfree.c
```

Electric Fence returned no messages.

Summary

From these tests, it appears clear that *Checker* is a complete product which found all the errors without any problems. It is quite easy to use, and you don't have to set a lot of switches because, by default, it checks for a wide range of errors. It does have a little problem when you use external libraries and functions (such as the GDBM). Actually, to ensure it will check for everything, you should recompile all the external programs with it. If you call a function not compiled with it, the memory bitmap used to track the memory accesses will not be updated; this will create holes in your checks. You have two ways to do this: recompile the library with checkergcc, or create function stubs. The stubs are particular aliases for each function, which perform some checks on the parameters passed to and returned from the function. In particular, you must check for pointers to see if the memory area you will access by using them has the correct status (readable, writable, etc.).

Provided in the package are many ready-to-use stubs for the most popular functions (such as **stdio** and the string functions). After a look at these stubs, it should not be difficult to write your own for libraries you cannot recompile with checkergcc.

On the other hand, *Electric Fence* showed some hesitation in error detection but is easier to use. It is enough to link it to the program and run it (no problem with external libraries). If used in tandem with *Mem-Test*, it will also detect memory leaks. For best results, be careful with the switches: use the correct spelling and the correct alignment and protect (below or after the memory allocation).

Resources

Cesare Pizzi started to play with computers on a VIC-20. When not playing with electronic stuff, he frequents the taverns of his mountains with his girlfriend Barbara. He can be reached at cpizzi@bigfoot.com.

Archive Index Issue Table of Contents

Advanced search