Preface

I have many fond memories of writing for ComputorEdge magazine.

In 2003, and for many years ComputorEdge magazine could be found for free beside every supermarket checkout or convenience store in Colorado. It was the place to go for light-hearted local technology news, and to find local stores with the best deals on computers.

I first started writing for ComputorEdge after returning with my family from some time abroad. I had been working as a telecom consultant in London. The work was exciting, and involved learning a number of new technologies, including lots of Visual C++ work. I found London to be incredibly exciting and fun, but also outrageously expensive. Money seemed to evaporate from my wallet at an unexpected rate, and yet we were living a lifestyle far less extravagant than American life in Colorado.

At the time, the technology magazine industry was in its final flourish. The internet was new and google still young. We had not yet formed the habit of looking up everything on-line. Professionals in the IT industry looked to magazines to stay current and expand our skills. Magazines like Byte and Dr Dobbs Journal were extremely useful and very widely read. There were (and still are) magazines for every segment of the tech market.

Needing money, I started supplementing my income by writing for various magazines, first in the UK, then in America. In those days one could easily make over $1000 for a good technical article, even in the second-tier of magazines. And they were all so hungry for content!

As a contrast to the more technical magazines, ComputorEdge offered several attractive features for me. It was weekly, with shorter, lighter articles. Heavy technical content was not what the editor wanted - he wanted entertaining writing for a readership who were not all computer programmers, but rather people with other professions who nonetheless enjoyed learning a little about these amazing machines.

At ComporEdge I was one of a number of writers who addressed the world of Linux. In those days Linux was still the province of computer-geeks. It was just entering the lexicon of the mainstream computer user. Together with the other ComputorEdge writers, I helped explain what was going on under the hood. What was Linux, where did it come from, and where is it going? These were the sorts of questions we answered for our readers.

Writing for ComputorEdge became my most fun writing gig. I got a chance to look up all sorts of historical oddities and wonders, and I learned a lot. I also took the chance to experiment with different writing styles, ranging from straigh-up instructional to humor.

This collection of ComputorEdge articles provides an easy-to-read ntroduction to the world of Linux as it developed. I think they are still relevant, entertaining, and informative, and I hope you will too.

Putting together this material as an ebook also provides yet another technology learning adventure. The means of production have truly been placed in the hands of the people - one person, working alone, and produce a book in a few hours. What will this do for our information based civilization?

Historical Notes

The articles in this section relate primarily to two giants of computer science: Richard Stallman and Donald Knuth. These two men have, with the work of their minds alone, fundamentally changed how we use computer systems. The articles below are written with the greatest respect.

Richard Stallman: Innovative Genius or Tinfoil-Hat-Wearing Nutcase?

Richard Stallman is one of the most influential and productive computer programmers alive today. He may also be one of the oddest.

Stallman's claim to programming fame began when he wrote a text editor called "emacs." The emacs text editor is more than just an editor. It was, in fact, the first (and still the best) integrated development environment—the perfect tool for programmers in any of the many programming languages used in the UNIX world.

Emacs is still one of the most popular programming tools in use today, and would alone have ensured Stallman's fame. But it was what he did afterward that was destined to have a far greater impact on the world: Richard Stallman founded the "free software movement."

Stallman is famous for his devotion to careful terminology. He has acknowledged that his use of the term "free software" has caused a lot of confusion. Free software doesn't mean software for zero cost; it means software that you have the freedom to use fully—to modify, to inspect, and to give to your colleagues and friends.

As Stallman put it: Free as in freedom of speech, not as in free beer.

If you buy software from Microsoft, you have only the company's word for what the software does (if it even tells you).

When you use free software, you have the source code. Just look and see for yourself what the software does. If you don't like it, or find a mistake, you can change the software directly.

These pragmatic reasons for preferring free software have been persuasive enough to convince many hard-nosed corporate executives to depend upon (and contribute to) many free software packages.

But these are not the reasons that inspired Richard Stallman. His motives are more idealistic.

The Idealism Behind Free Software

Stallman's ideas of software freedom were inspired more by the "golden rule" than any attempt to improve the software-development lifestyle. As explained in the original announcement of the GNU Project, Stallman said that he wanted programmers to work together to benefit society, not to work against each other in a quest for profit.

Stallman never objects to people working for a profit—merely to the way in which the proprietary software industry currently makes a profit: by restricting the freedom of users to fully use the software.

Stallman's idealism led him to break with the direction of software development in academia and industry. His GNU Project was so far from the norm that few knew what to make of it. No one could have predicted the effect that Stallman's efforts would have.

Stallman turned the world of software development upside down.

Almost 25 years later, the free software movement is a major force in software. Many of the tools used in the Internet are free software.

The GNU tools have also enabled a new operating system, Linux (or as Stallman refers to it, Linux/GNU). The Linux kernel would have been both impossible and useless without GNU, because it was inspired by and developed with GNU free software tools. And, without the GNU tools to run, the Linux kernel would be no more than an academic curiosity. The Linux kernel is the core system code that makes Linux work, but the GNU tools are what makes having a Linux system worthwhile.

Why Free Software Really Works

Stallman's vision, though it seemed completely crazy, has revolutionized software engineering.

Free software projects, unlike their proprietary counterparts, continue to improve over generations of developers.

The efforts of a proprietary software company necessarily stop with the end of that company, or that product line. No matter how many loyal fans there may be, no matter how many users might

depend on a piece of software, if the corporation that owns it decides to drop it, or if it goes out of business, then that software package is dead. No one can maintain it. No one can expand it. No one can port it to new computer systems.

With free software, good software becomes immortal.

The result is that useful, free software products tend to get better over time, and there are many software packages that have had several generations of developers, each taking over from the one before, without starting from scratch or throwing away any useful work.

Stallman has often pointed out that free software does not have to be developed for free, or given away for free. But the free software model does change the economic model of the software industry.

With proprietary software, you are locked into one company and need to pay whatever that company asks, or else abandon the software. With free software, that blackmail can't take place.

Plenty of companies can still make money.

IBM is one of the biggest contributors of free software to Linux, obviously not from idealistic motives. Free software allows the company to generate more profits.

IBM's free software success has demonstrated that free software is not anti-capitalist; it provides yet another way in which the free market provide solutions.

The Ways of Genius

It's customary for geniuses to be eccentric, and Stallman is no exception. He has numerous quirks and causes, many of which can be found on his personal Web page (www.stallman.org).

Other programmers report that Stallman is extremely difficult to work with. He has been called a "control-freak" who doesn't play well with others.

Although his reputation for eccentricity may be well-deserved, it does not detract from his accomplishments and contributions to the field of software engineering. As one of the most famous programmers alive today, he is entitled to his quirks.

Is Stallman a genius or a nutcase? Perhaps both. But he has changed the world forever.

Hang 'Em High With TeX

In 1962, the Beatles released their first record, and a young computer scientist name Donald Knuth started the first in a series of books called The Art of Computer Programming.

Knuth's books had an impact that was immediate and widespread. They became some of the most important texts on computer science, and they often took a rigorous approach to describing computer algorithms with mathematical formulas.

What Knuth originally intended to be one book has expanded into at least four, and the series continues, 45 years later. Volume two was released in 1969, volume three in 1973, and the first installment of volume four in 2005.

Knuth labors on at Stanford University, and the entire computer science community eagerly awaits his releases of new material. He's a sort of J. K. Rowling of the computer science world. But this article is not about Knuth's famous books, but rather about something that happened to Knuth along the way.

An Historical Inconvenience

In 1976, when Ford was president and the Viking 2 spacecraft landed on Mars, Knuth was trying to get out a new edition of volume two of The Art of Computer Programming.

At the time, most word processing was done with paper and pencil, or, for the more advanced, typewriters. If a document needed to be created for distribution, it was done so by a specialist typing these manuscripts into a typesetting machine, a giant computer system about the size of a small car, which cost thousands of dollars a month for the organization to maintain.

The typesetter would type in the manuscript and then print the output, called the proofs, and take it back to the author of the document. The author would then edit the document (by writing on it with a pen), giving the changes back to the typesetter, who would then type them back into the typesetting system.

Naturally, this took time, and mistakes crept into the process. It was a painful process.

When he planned his reprint, Knuth was horrified to learn that the typesetter used for volume two was no longer around. It had been a proprietary system, as were all typesetting systems, and the company had folded, leaving Knuth with the prospect of redoing the whole process again.

He was frustrated.

The Personality of the Programmer

Are you, or is any member of your family, a computer programmer? If so, you will notice certain personality characteristics.

Programmers will often obsess about minutiae. Lots of other people were confronted with an illogical and inconvenient system for printing books. Every author right up until Knuth had to face these problems. No doubt with some grousing and complaining, they managed to deal with whatever they had to deal with to get their book in print.

Any other author, confronted with the problem of redoing the typesetting of a book, would either get to work or abandon the whole idea. Not Knuth. He has the soul of a programmer, and he took the usual programmer path: writing software that can do it better.

He developed a truly modern word processing and typesetting system that is still in use today. It's called TeX, usually spelled with both the first and last letters capitalized. And it's not pronounced like the first syllable in "Texas." It's pronounced like the first syllable in "technology."

TeX Today

TeX lives on, and has got to be one of the most stable and bug-free software tools in widespread use today. Knuth no longer changes TeX, except to fix any reported bugs. These are few, however, because of Knuth's programming skill, and because he has long offered a reward of $2.56 for each verified bug.

Other than bug fixes, TeX is frozen. No new features will be added.

However, that doesn't mean TeX development is at an end. It is freeware, and Knuth encourages the development of other variants and versions of the core technology. One prominent example of this is LaTeX, a layer of software that sits atop TeX and makes producing mathematical, engineering and scientific documents easier.

He asks only that, whatever you do, you don't call it "TeX." As with his seminal series of computer science texts, TeX will always be associated with him alone.

TeX to Texinfo

The TeX engine is used by the Free Software Foundation—influential developers, maintainers, and distributors of the most important tools in Linux: the compilers and system tools that put the UNIX in Linux.

Texinfo (texinfo.org) is a program that offers a simple way of encoding typesetting information in a text file. It allows you to produce a document in a variety of formats from the same text source, so that you can write one manual and have it available as a Web page, PDF file, plain text file, and PostScript file, as well as the info file used by GNU Emacs.

Since it's the official typesetter of the GNU project, it is well supported on all Linux systems, and it is used for all GNU documentation. A quick Web search for your favorite GNU tool will show the output of the texinfo system, and if you download a printable copy you can see the typesetting done by TeX.

The Triumph of TeX

Knuth wrote TeX because of his own personal frustrations dealing with typesetters in the production of his series of books. Instead of just doing the smart thing and accepting the limitations of text processing systems, he invented his own system—and it was great enough that it still forms the basis of many typesetting systems today.

In typical programmer fashion, Knuth estimated that one year would be required to solve the typesetting problem. In fact, it took him 10. It's nice to know that even a genius can get the schedule wrong.

While his schedule may have been inaccurate, his software is nearly perfect, and its influence has been profound over the last several decades.

Curses, Foiled Again!

In January of 1984, two things happened that would change the course of computing forever.

One of them was the release of the Macintosh, the first real consumer computer with a graphical user interface (GUI), a mouse, and bitmapped graphics as a standard. Each of these technologies had existed in other computers, but the Macintosh united them in a single consumer package, and made computing history.

It was the bitmapped graphics that made the Mac so pretty. With a screen resolution that was far greater than any PC of that era, it presented stunning graphics, which rapidly made the Mac the computer of choice for publishers, graphic artists, and those who found the command-line interface to be more than a little confusing.

When Apple released the Macintosh, it launched the Age of the GUI.

Apple's GUI did more than provide cute little icons and windows: It changed the way computers are used forever. Today, there is not one general-purpose computer that comes without a GUI. Even UNIX, always the stronghold of software conservatism, is now so GUI oriented that many Linux users never even learn to use the command line.

But like every important technology, the GUI didn't just create opportunity—it also destroyed technologies.

The invention of the automobile destroyed a thriving horse-and-buggy industry, and the invention of the electric light bulb destroyed a huge gas-lamp business. Similarly, the dominance of the GUI came at the expense of another fledgling technology.

It was a technology that could draw a screen for interaction with the user, and allow the user to move the cursor around on the screen, select text and, in short, do much of what the GUI can now do.

These days we call it a textual user interface, but back then it was known as "curses."

A Terminal for Every Season

Long before the GUI, in the era of disco music and 8-track tapes, the so-called "smart" terminals had replaced the clanking teletypes that output everything on sheets of computer paper. The computer user interface was just being born.

At first, all that was sought was a way to reproduce what had previously been done with punch cards—that is, the input and display of 80-character text strings, with no formatting. The first modern terminals had little chunky green numbers and letters. Nobody complained about the font; we were all happy to read the screen instead of looking at holes punched in a card.

Back then, no one even knew what a font was. If you'd asked, I would have guessed some sort of snack food.

Soon, however, the interactive nature of the so-called glass teletypes started to assert itself. Programmers started playing with menu and help text that popped up, and then went away. The text editor vi appeared, which allowed the user to drive the cursor around the screen, and modern word processing began in earnest.

Some terminal manufacturers even started experimenting with colored text, underlined text, and even blinking text. We'd come a long way from the plain old punch card!

In those heady days there were dozens of different terminal types, with different capabilities and qualities. Soon, programmers had more terminal options than they knew how to make use of.

Unfortunately, these different types of terminals all spoke different languages. While it was possible to make software work on any one or two terminals, it was extremely difficult to manage more than that. So software written by one batch of scientists couldn't be shared with any other batch of scientists, because they were using different terminals.

The answer, of course, was to write a software library that would present a common interface to everybody—one that would know the details of every different type of computer terminal.

With such a library (which was called "curses"), programmers could write their program to talk to the common interface, and it would work on any terminal that curses knew about. There was even a way for users to add information about a brand-new smart terminal, so that they could start using it, perhaps before the curses programming team had even heard of it. Thus, a brand-new terminal could be put into almost instant use.

Curses was written as part of the Berkeley version of UNIX in the early '80s. It was wildly successful, and can still be found on every UNIX system, as part of the archaeological strata of the UNIX operating system.

The GUI doomed curses to obscurity, but that doesn't diminish the technical achievement of a team of early programmers, struggling against a hostile universe of hardware incompatibility.

The Other Thing That Happened in January 1984

January of 1984 was a fruitful month for computing technology. Not only did Apple introduce the Macintosh to much publicity and fanfare, but (with no publicity and zero fanfare) Richard Stallman announced the GNU project.

GNU sought to produce a free software version of UNIX. The GNU project launched the free software movement, which led to Linux, and so much more.

More than 20 years later, we can look back on the influence of the Macintosh and GNU, and wonder what current technology is about to be made obsolete.

Introduction to Linux

The ComputorEdge readership were early adopters of fun new technology. But Linux has a somewhat scary reputation as being hard to install and configure. While this is certainly true historically, there are many new tools that make installing Linux a breeze these days. Some ComputorEdge columns were aimed at computer users thinking of taking the plunge. As I'm a huge Linux fan, I had no trouble coming up with reasons to switch.

Helping Windows users switch to Linux is greatly assisted by Cygwin, a terrific port of many Linux tools to the Windows platform. It's easy to install, and co-exists harmlessly with Windows on your system. But all the Linux tools and programs are there for you. It's a great way to learn Linux without getting rid of Windows.

ComputorEdge was certainly the most light-hearted magazine I wrote articles for. Other technical magazines would need to look up hiumor on the internet, but Jack Dunning, the editor of ComputorEdge, let me try to have some fun. But the article I thought most funny got a cold reception from the readership. A parady of a very popular TV relationship drama, it examines the emotional aspects of switching to Linux.

To be a successful computer user, you must be constantly learning. Some columns were about the Linux documentation tools and how to use them effectively to learn more about Linux. Once ready to install Linux (other than Cygwin), a new user faces the choice of which Linux distribution to use, and there are a plethora of choices.

Getting Starting With Linux

Linux is really one of the best deals ever. You get millions of dollars worth of software, absolutely free. How can you beat it?

The hard part, sometimes, is just getting started. How does an ordinary user begin?

In this series of columns, I will take you from zero to 60 with Linux. By the time we are done, you won't be an expert, but you will be well on your way to using the best operating system ever developed.

The first task is to get Linux on one of your machines. You can install it on an old machine, install it on top of Windows with Cygwin, or just buy yourself a Linux computer.

Put Linux on an Old Machine

Give me your tired CPUs, your poorly upgraded,
Your small-disked systems yearning to breathe free,
The wretched refuse of your Windows network.
Send these, the RAMless, documents lost, to me,
I lift my lamp beside the golden door!

These days there are hundreds of thousands of old PCs around that are no longer capable of running the latest versions of the operating system from Redmond, Washington. These machines were hot in their time, but like an aging Hollywood starlet, they languish while everyone chases the younger and newer talent. Today's fickle and demanding Windows user has no interest in these machines!

Lots of people have old computers just sitting in the basement (I've got at least three). They are fully functional computers, with five- or 10-year-old processors, hundreds of megabytes of RAM, and tens of gigabytes of disk space. If you could send one of these computers back in time about 20 years, it would be the most powerful computer in the world. If you could send it back to 1940, they would give it to the atom-bomb scientists, who would have kept it in use 24 hours a day, 7 days a week, running calculations that would vastly contribute toward the war effort.

These days, we can't even be bothered to keep it powered up. Such are the casualties of Moore's Law!

These computers just need an operating system upgrade to be turned into useful machines, suitable for Web browsing, office tasks (such as word processing and spreadsheets), and the other tasks of your average Windows computer. The documents you get will be compatible with Windows; your Web browser will look and work just the way it does on Windows.

Unless you use Internet Explorer. In which case, stop now! Please.

The great thing about one of these castoff machines is that they are completely expendable. Just cheerfully reformat the disks, and away you go. (But do check with the wife to make sure the computer isn't the only one holding photos of the kids for the year 2002. Don't just assume that because the computer is in the back of the garage, she has nothing on the disk. Don't just blithely reformat the disk without backing it up somewhere.)

There are lots of different distributions of Linux, but you won't go wrong with any of the big favorites, such as Ubuntu or Fedora. You can download CD images, or pay a few bucks to get a set of Linux install CDs, or ask around at work and someone will have a set.

Old laptops also make great Linux machines. I recently converted my old Windows laptop, a Toshiba Satellite. With Windows it was slow and clunky; with Linux it is fast and wonderful!

Use Cygwin on a Windows Machine

For those who can't bear to boot a machine without Windows, there is the Cygwin option. Cygwin is a port of Linux to Windows. That is, Linux runs right on top of Windows. Just go to the Cygwin Web site (www.cygwin.com) and click the Install Now button to begin the Cygwin installation.

This is not the most efficient way to run your hardware but, in general, any machine that can run Windows will be able to handle Cygwin.

With Cygwin, you get a full set of Linux tools, with the same old Windows desktop and tools, side-by-side.

This is a great choice if you have recently invested in new Windows hardware, or need to run some Windows applications as part of your work, or if you want a game system that can do Linux on the side.

The Cygwin distribution is not a cut-down, limited version of Linux—it is the whole enchilada, ported to Windows. There's (almost) nothing that a full Linux box can do that can't also be done on a Cygwin box. You can even allow others to log into the machine remotely, and get a full Linux command line and windowing system.

And with a Cygwin-based machine, Windows is still in the driver's seat when you need it to be.

I don't like Cygwin machines, but that is not Cygwin's fault. The problem is that they still are running Windows, and that is taking a lot of resources. Any machine that can run Cygwin at a reasonable speed would be a real screamer with just Linux installed. So why not just cut out the middleman and take all the advantages of Linux?

Alas, sometimes you can't. If you're imprisoned in Windows, Cygwin at least allows you to use decent tools.

Buy a Linux Machine

I'm a big fan of doing things the easy way, especially when it comes to computers. And you can't get any easier than just buying a Linux machine off the shelf. In many ways, this is the best and easiest alternative.

Are you looking for a new machine? Many Windows users find themselves doing this every few years.

If so, give serious consideration to buying a Linux box.

Since it does much more with much less, a new Linux machine can be a lot cheaper than a new Windows machine, yet still do the same job just as well. Buying well behind the curve, Linux users can reap the benefit of mass-produced, commodity hardware, and let Windows users bear the cost of getting the cutting-edge stuff.

And next year, when that cutting-edge stuff has dropped to half the price, you can always add it if you need it. But you will probably find that a Linux machine remains useful just as it is for years and years. After all, why should a machine not keep working year after year, with reasonable performance? This is not much to ask, but computer users have been conditioned to think it's impossible.

Not only is it possible, it is now even easy.

More and more desktops and laptops are being offered with Linux installed. Just take it out of the box, plug it in and turn it on.

This doesn't just save you the $100 that you would otherwise send to Bill Gates every time you buy a computer—it also ensures that you won't have any trouble using any of the hardware devices on your machine, which can happen when you switch a Windows machine to Linux. (This has not happened to me in a number of years, with modern Linux distros being so good.)

And believe me, even the chintziness and most economical new machine simply flies with Linux instead of Windows. No need to lay out the cash for the top-of-the-line hardware.

What You Get With Linux

All operating systems are layers of software upon software. Each builds on the past, and the nifty new features of today depend on the nifty new features of yesterday.

This can be done poorly, or it can be done well. When it is done poorly, you get a huge, bloated, poorly organized mass of software. Like a skyscraper built of jello, it can't get too high without spreading out over a huge area. It becomes a giant, cherry-flavored, quivering blob.

When done well, the system software is like the Eiffel Tower, tall and straight, and occupying a minimum amount of horizontal space.

In Linux, things are done well, not poorly. That's why Linux users don't have to upgrade their machines every three years just to keep running the same software. And that's why old Windows boxes can be readily converted into useful Linux machines.

Whichever way you get to a Linux machine, you will be treated to a wealth of high-quality, stable and efficient software. With Linux you can get off the merry-go-round of hardware and software upgrades, and get a system that just works—and keeps on working.

How to Make the Switch to Linux

Across the land, from a host of Windows users, the question echoes from the teeming coastal cities to the sweltering central plains: "How the heck do I switch to Linux?!?"

As one who has reached the Promised Land, let me assure you it is worth the journey. Linux and Windows both do the same things—but Linux breaks much less often. Linux seems to work with you, while Windows seems to be working against you. Windows is the evil twin of Linux.

Switching to Linux can be accomplished in many different ways, ranging from easy to pretty challenging. You decide how much time and energy you have to put into the process.

Taking the Easy Way Out

When I was a young child, my teachers frequently condemned my laziness. "You always take the easy way out," they would wail, as if that is somehow a bad thing. (Little did they know that my love for the low-hanging fruit is exactly the quality needed for effective engineering.)

With a Windows-to-Linux switch, the easy way out is called Cygwin. Go to the Cygwin (www.cygwin.com) Web site and hit the button on the upper-right, which says, "Install Cygwin now." This will download a program called setup.exe. Run this program, and you will be directed to answer a few questions. (When asked to select a Cygwin repository, just make a random choice.)

If you play around with setup, you will see that there is a huge list of available software, with some items already selected. The selected items are the minimum Cygwin installation, and you should just accept them and let setup install them for you. If you have only a dial-up connection, you will have to leave the machine connected for quite a while for this.

Once the Cygwin setup has worked its magic, you will have Linux installed right on top of Windows. Now you may continue to use Windows, but gradually transition to Linux in the background. That is, you can start using the Linux flavors of your favorite tools. You will find an icon for Bash, a Linux command line, on your Windows desktop. Bash is like DOS (in the same way that the Space Shuttle is like a bottle-rocket.)

After the basic install, you will want to run setup.exe again and get a bunch more software. (All free, of course.) You can also install X Windows, the Linux GUI, and run X-based Linux programs on your PC. (Or, you may choose to use the Windows-based versions of free software, like OpenOffice, and just use Cygwin for command-line tools.)

The great thing about Windows boxes is their incredible computing power. With a beefy Windows box, Cygwin programs run at a screaming pace. The bad thing about them, of course, is that they have Windows on them. Which leads to the next path to Linux: buying a Linux computer.

Buy It, Don't Build It

Back in engineering school, they told me "never draw what you can copy, never copy what you can tear out and paste." (The expression reveals my age in an era where engineering students think of "drawing" as something done, like everything else, with a computer keyboard.)

Despite its age, there is a valid principle at work. You can't do everything, after all, so why not let someone else install Linux on a computer, and just buy it?

If you take this path, you must shell out some money, but you get a ready-made solution that works from day one, right out of the box. And with Linux's svelte runtime profile, much less hardware is needed for a Linux box than would be needed for a Windows box, so you can get something inexpensive, and still get the kind of capability Windows users chase with top-of-the-line hardware. (They chase it, but they don't get it, because it's about the software, not the hardware.)

If you're looking for a Linux box without a high price tag, take a look at the Asus ultra-portable Eee line. For $400, you can get a nicely powered Linux laptop that weighs about two pounds. Plunk it down on your desktop, and plug in a VGA monitor and USB keyboard and mouse (and don't forget to plug in your speakers). You can use the screen, keyboard and mouse from your Windows computer, in fact. Now you have a very nice Linux desktop machine.

And when you want to go on a business trip, you fold it up and slip it in your coat pocket or carry-on bag, and you can make it to your destination without an aching shoulder. Once there, you can work on its tiny (but adequate) keyboard and screen until you get back home.

It comes with wireless networking, but no hard drive. There's a few gigabytes of flash RAM storage, plus a microSD slot where I have another 2GB of storage that I bought for about $25. With the SD slot and the USB ports, you can get a lot of extra storage without a hard drive. The upside is that the Eee is very durable, as it has no moving parts, other than the keys.

Wipe a Computer, and Install Linux

Lots of Windows users have old machines lying around that are no longer powerful enough for the software from Redmond. Rest assured that they are more than powerful enough for Linux. In particular, old laptops can be used as power-efficient network servers. If you run Linux without a GUI, you can make use of even very old machines.

To install Linux, you'll need an install CD. You can make your own by downloading a giant file called the disk image file (or ISO file). With the ISO file, and a read/write CD, you can make

your Linux install CD. You must select your Linux distribution and get its ISO. Ubuntu is a good choice for those new to Linux. It has everything you would expect in a home/office workstation.

The easy way to do it is to wipe out whatever is on that machine's hard drives. Take a copy on some other media, and get ready to hose down the storage. If you have more than one disk, you can disconnect one and retain its contents (Or leave it connected, but be very careful when you are at the installation step where you reformat hard drives.)

Then you boot with the install CD and follow instructions that pop up on the screen. It's as easy as installing Windows.

Build a Computer, Install Linux

Why, oh why, do I always skip all those easy ways and go right to the hardest? Perhaps because it's also the most fun.

Pick out the parts, put together your own box, and install Linux yourself. You will get a real powerhouse for a very good price. And you will learn a lot. Afterward, your experience level in Linux (and hardware) will be much higher.

Which reminds me of another lesson from engineering school: Experience is the thing you have just after you need it.

Linux and the City

Vespa is Denver's last, best refuge of hipness and original food, but last night someone made the mistake of telling the crowd at the Libertarian convention about it. As a result, the place was crowded with East Coasters, looking nervously about, visibly trying to find something comfortable to look at, like someone in a three-piece suit. Instead, all they could see were relaxed Denverites, in their jeans and sneakers.

I was waiting for the bartender's attention in this crowd of leisure-suited freedom-lovers when my significant other turned to me and, giving me that look with her beautiful brown eyes that always caused my knees to go weak, asked the question I have been secretly dreading for years: "Will you help me switch to Linux?"

As the part of my brain responsible for getting along with my sweetie mechanically caused my mouth to accept graciously, neurons in the part of my brain responsible for staying married to her

started firing. The path to Linux may be fraught with difficulty, and we don't always love those who lead us along this path. In fact, frequently we get mighty cranky with them.

Later, when I had had a chance to get used to the idea, I asked myself the question: Is it a good idea to help your significant other switch to Linux, or should you hand him or her off to a competent geek friend, knowing that, however rough the switch, at least you won't get the blame?

The Brunch Bunch

The next day I broached the question to my three closest friends at brunch. The first to answer was Sam, the oldest and most experienced member of our group, who regularly took on the task of teaching Linux to friends, acquaintances and people he barely knew.

"Go ahead and help her switch! Sure, it'll be hard, and she'll do some whining and complaining about how everything was different under Windows, but soon she'll get the hang of things, and learn to love Linux. In the end, she'll thank you. And if she doesn't, probably you weren't right for each other anyway."

My friend Charlie immediately objected. "No, Sam, you never stop to think about how people feel. The emotional attachment that people feel for their operating system is much more powerful than you think. Right or wrong, her relationship with Windows has lasted over a decade. And now she'll have to break it off and move on. It's never easy!" Charlie turned to me. "You need to take the time to understand her feelings, and validate them."

Our other friend, Marty, offered the most cynical opinion, as usual. "Don't do it. The first time she hits something that Linux doesn't do well, she'll blame it on you. She'll be at one of her meetings and some dork with Birkenstocks and a ponytail will show her his new Vista laptop from HP, and she'll have a seg fault. Find someone else to take the rap."

I couldn't help but think that Martyhad a good point. Switching to a new operating system can be challenging, and even if the old operating system is annoying and stupid, it's what you're familiar with.

After brunch, Charlie and Sam went shopping, as usual. A new CompUSA had opened in town, and Charlie was going to splurge on a new quad-core server, and Sam was explaining that he needed to max out his RAM right away. Marty, on his way back to his law office, offered one parting piece of advice.

"Whatever happens, don't just reuse her old Windows laptop. She'll want something shiny and new, and having the old laptop around will be useful when she does hit something that she can't figure out on Linux, but that needs to get done right away. Also, she's bound to have a lot of files on the thing, and she may want some of those old files in the coming months. Good luck."

Introspective Scene of Examining Our Relationship (to Operating Systems)

As I walked down the 16th St. Mall, window-shopping, I reflected on my friends' advice, and our emotional attachment to operating systems. It reminded me of something I had learned from my children: People want to stick with what they know. A child might not like what he is used to, but he knows that he doesn't like what he is not used to. Doubt it? Then try feeding a brand-new recipe to a 6-year-old.

Windows, with all its faults (and there are many), is what my wife has been used to for these past 12 or 13 years. During that time, it has been with her in good times and bad. (And, computers and software being what they are, that mostly means bad.) She knows its quirks and pitfalls, and she is used to the way things are done on Windows.

I tried also to think how it would be for someone to try and convince me to give up Linux and Unix-like operating systems. I have been using them daily since the first time I sat down at one in 1986. That's 22 years of Unix knowledge in my brain and fingertips, and that is not something I would give up easily.

(Thankfully, I use Emacs, so all operating systems look the same to me anyway.)

Neat Wrap-Up

In the end, I decided to help my wife switch from Windows to Linux. It all comes down to trust. But while the switch from Windows to Linux may not be wrapped up as quickly and conveniently as an episode of your favorite female relationship drama, it can help to make the computer, and the tasks you use it for, seem a little less threatening and annoying. It can make life better.

And this, dear reader, is the secret to a good marriage: Try and make your spouse's life a little bit better in some way. Because you surely do cause your share of problems and difficulties. Like a good operating system, it's important to make sure the positives add up to more than the negatives, that the features outweigh the bugs.

Cygwin: The Perfect Setup

As a longtime fan of free software, there's not much that I admire about proprietary software. The software that is used with a commonplace operating system originating in Redmond, Washington, has little charm for me.

Sitting around the lunch table with other free-software geeks, I am always loud and proud in my derision of that other operating system and the software that goes with it. We spend hours talking about software, and never is the software from Redmond given a kind word. (Perhaps we should talk about something other than software sometimes, but we are geeks, after all.)

However, there is a dark, shameful secret that I have never revealed to my geek buddies. It is something I could never admit to, something they would laugh at.

I admire the way Windows software installs.

Easy Installation for Windows Users

When you get a piece of Windows software, you put a CD in the drive, or run an executable file, and a friendly dialog box pops up. It explains everything in small pieces and small words. It gives reasonable default choices for everything, allowing lazy system admins such as me to just click "OK" again and again, without even reading the one sentence it has put up there for me.

I can't even be bothered to spend five seconds reading and understanding the installation instructions and choices. How pathetic is that? I blame MTV.

But on Windows—that operating system that gets so many things wrong—the installation program always works. Perfectly. Every single time.

Doing Hard Time on UNIX Installs

UNIX, which is a very capable and professional operating system, can do so much that Windows can't. (continue working for long periods of time, for example). But one thing it does not do well is software installation.

Instead of getting some nice pop-up box with instructions simple enough for a clever monkey, UNIX software is distributed in a wide array of strange ways, ranging from source code (which you build with your own compiler, kind of like building a radio from a kit), to binary distributions, which are almost, but not quite, as easy as Windows distributions.

When installing UNIX software, you almost always have to read the instructions. We free software geeks tell each other that we prefer it this way. But, deep in my heart, I don't. MTV has made it very difficult for me to concentrate on anything for long periods of time.

Come Home to UNIX With Cygwin

That's why I am always delighted to work with the Cygwin setup program.

Cygwin is a complete port of UNIX/Linux to the Windows platform, and it is as easy to install as any Windows software.

Unlike Windows software, Cygwin software is free, and the source code is also available, in case you want to modify it yourself, and it always, always, always works!

Once you have Cygwin installed on your computer, you click on the Cygwin icon on your desktop, and a bash shell command-line window opens up.

Immediately, my heartbeat slows and my palms stop sweating. I forget that I'm working on a Windows machine. Cygwin gives me a sane, powerful, and delightfully normal UNIX tool set. It's like coming home again.

Even the X Window system—the UNIX windowing system that provides windows, menus and tool buttons, and all the other graphical gunk that some people seem to need on their screen—is supported. (As for me, I'm happy with a smart terminal emulator and emacs in ASCII mode. What more could anyone possibly need?)

UNIX Therapy for Windows Computers

To get this wonderful injection of quality software into your Windows laptop or desktop box, just go to the Cygwin Web site (www.cygwin.com). Click on the icon "Install Cygwin Now."

This will download the wonderful Cygwin setup program setup.exe. Save this program somewhere easy to find, such as your desktop. You will run it whenever you want update your Cygwin software, or to get new software packages.

Run the setup.exe program. Go ahead and accept all the default settings, just like on any other Windows software installation. When asked to choose a download site, I just pick one at random. I try to choose a .edu site, for some reason. I don't know if they are faster than commercial sites or not.

After you choose the download site, the setup program downloads a list of software packages. Now comes the fun part.

Ali Baba's Cave of Software

Maximize the setup window; you will need plenty of room. Here you will see listed the software wealth of our generation.

You'll find bash, the Bourne Again Shell—the best command-line interface ever developed. Or, for those with different tastes, csh, sh, zsh, or several other varieties of command-line environment.

You can get emacs, the first and still the best integrated-development environment. For the more primitive, there is pico, nano, joe, and even the old UNIX line editor ed. There is even a vi editor, for those who are seriously brain-damaged.

Get such acclaimed scripting languages as Perl, Python, and Ruby. There is also the GNU Compiler Collection (GCC), which will handle Java, C, C++, Fortran, Pascal, Ada, and other programming languages. No matter what your preferred (non-Windows) programming environment is, you can find it here.

There are too many to list—enough tools to keep you busy for a long, long time. There are more than enough to fill an undergraduate computer science curriculum, and add in a graduate-level curriculum as well. More than enough to do very serious work.

These are not knock-off tools, Windowized in some way. They are the actual UNIX tools, built for you on a Windows box. The scripts that you write for Perl, the C programs you compile with GCC, are exactly the same as those you would produce on a UNIX system.

These are not toys; they are the same tools being used in the cutting edge of industry and research labs.

All free! Amazing.

Who Should Get Cygwin?

Get Cygwin if you are a UNIX programmer with a Windows box. You will be very happy to have your usual tools always at hand, and you will find that you can use the Windows box as a UNIX development platform.

Get Cygwin if you are a computer science or engineering student, so that you have access to industry-standard tools. These tools will help you in the classroom, and in the interviews when you're job hunting.

Get Cygwin if you want to know more about UNIX without making the commitment to switching to Linux on your laptop or desktop. Or, if you are sick of the limitations of Windows tools and want something a little more serious.

Then enjoy it, and remain in the UNIX cocoon of safety instead of venturing into the wilderness of the world of Windows!

How to Become a Linux Guru

Software is one of the most poorly taught topics on the planet.

Get a civil engineer, and he or she will know all about roads, concrete and steel. Get a mechanical engineer, and he will know all about locks, hinges, pistons and pulleys. Get a software engineer and she might not even know what language you're programming in. There is little agreement as to what a software engineer should know — even about what a software engineer is.

It's an observed fact that some people are good with computers, and other people aren't. We call the good ones gurus and the rest users.

How do you tell the difference? A guru is the one who can solve unexpected problems, come up with comprehensive solutions, and get the darned computer working the way you want it to work. A user can not. It's the kind of thing that's easy to see, but hard to explain.

But the guru you seek in times of computer trouble started out just as you did, with no formal training in the Linux operating system or software. How, then, does the guru manage to tame the information wilderness of the computer? Why is it that he or she knows what you don't?

Dedicate Yourself to Learning

To become a guru, you must first understand that there is far too much to know, far too many interesting nooks and crannies for even a lifetime of effort to explore. The amount of creativity, effort, time and money that went into the creation of the modern Linux box is staggering. With the almost unique easy reproducibility of computer software, it is possible for software to remain in use for decades, forming the foundations for the next layer of software, which will, in turn, form the foundation for even more layers.

In every Linux box there are hundreds of thousands of lines of software, created over the last four decades by some of the most talented and productive programmers ever to have lived. This intellectual heritage, this wealth of functionality, is given to everyone free with Linux.

You can't learn it all. You can't even learn 10 percent of it all. But that doesn't mean you can't try.

Gurus are always trying to learn more about the operating system. If you want to be a guru, you must cultivate this attitude of study, this dedication to continued learning. There is no end to this road except death. (A bit depressing when you look at it that way, isn't it?)

Finding the Answers the Old-Fashioned Way

Given this huge amount of software, how do you find the answer to a specific problem?

Back before the Internet was invented, the UNIX world had developed several documentation strategies to cope with this problem. The first was the simple but robust man page. ("Man" is short for manual.) This is documentation, distributed with the operating system. It's displayed with the man command. To learn about the find command, type: man find. (To learn more about the man system, type: man man.)

Although the man system is extremely useful and still widely used to this day, there are limitations. Since each man page is a separate document, there are no hyperlinks between documents. There is also no good search capability.

To address these issues for the free software projects they were undertaking, the Free Software Foundation (FSF) developed the texinfo system, which is based on the popular text editor emacs, but can be run in stand-alone mode with the info command. The info systems allow documents to be arranged in a book-like organization, with chapters, sections, sub-sections, etc. It supports links between documents and various forms of searching. Emacs users can get the texinfo documentation right within emacs.

For FSF tools, which make up much of Linux, the texinfo documentation is more detailed and more useful than the man pages. It can also be used to generate nice-looking printed manuals, PDF documents and Web sites, all from the same source. This is very handy when documentation is maintained by busy programmers, as with most free software. You can get at it with emacs or the info program, but it's also available on the Web (see below).

Finding the Answers the Newfangled Way

The Internet is the central repository for human knowledge on free software. I don't know any guru who does not use Google or Yahoo to find new information to supplement what can be learned from the formal documentation in the man or texinfo documentation.

Frequently, the very annoying obscurity of error messages can be used to help find the information you need. When faced with some error message that means absolutely nothing to you, just cut and paste it into your favorite search engine to get a wealth of user comments and experience.

Since free software documentation is also on the Internet, you can sometimes just skip the man and texinfo documentation and jump right to the Web. The same documents you would get from the man or texinfo systems will be right there in your search results, along with other hits from related Web sites.

Read Slightly More Than the Minimum

There is only so much that you can learn in a day, and the best way to continue to learn is to try and learn a little every day, or even every time you interact with the computer.

One difference that I have noticed between gurus and non-gurus is that the gurus will spend an extra five or 10 minutes reading the documentation, even after they have found the answer they were looking for. The non-gurus will crack the documentation open when they can't figure out how to do something, and as soon as they have found the answer they will close the books and hit the keyboard.

That extra 10 minutes of reading each time allows the guru to take away a few extra pieces of information about the operating system. Instead of reading the absolute minimum of documentation, the guru reads a little bit more.

You too Can Be a Guru

To be a guru, you must adopt the attitude that learning about computers is something you will continue to do your whole career. You must recognize how little you know of the topic, and how much you have to learn. You must try to learn something every day, and from every computer problem you run into. Don't just work at it until the problem is solved; work at it until you have a good understanding of how the problem was solved.

Over time your knowledge will grow, and your ability to find the right information will grow with it. People will start coming to you for answers, and you will hear yourself referred to as an expert or perhaps even a guru. Yet you still won't feel that you really are an expert, or that you really know all the answers.

But gurus don't know all the answers; they just know how to ask good questions.

The Distribution

One aspect of Linux that confuses new users is the huge number of choices available. On the other hand, the commercial operating system from Redmond comes in two flavors, Pro and Home (which presumably means amateur!).

Linux is all about giving you choices. It comes in hundreds of flavors.

But there are only so many choices that your brain can handle before it throws up your arms in disgust and gives up!

Back in the Old Days

The computer hardware sitting on your desk is just a big, general-purpose and extremely complicated abacus (but the beads are electrons). It's so complicated that if you had to start with just the hardware, and tried to write a letter to someone, it would take you years. And add a few more years to make it print.

This was what life was like for the early users of computers, back in the 1940s and 1950s. But gradually, the body of existing software began to accumulate. It wasn't necessary for every new user to spend a month to write a letter. Once the first word processor was written, everyone could use it.

It was this ability to build atop the work of others that inspired Richard Stallman to start the Free Software movement and the GNU tools. And it was those tools that allowed Linus Torvalds to develop the Linux kernel in 1991. He built upon a body of software that already existed—which he downloaded, one-by-one, and installed on his computer.

So it was throughout the world. Excited users would get the Linux kernel and then use it for the core of their system. But, just like the users of the 1940s and 1950s, the kernel alone would get them only a working piece of computer hardware. They still needed to assemble various useful programs (like a word processor) and get them working with their Linux kernel. Most of these useful programs came from the GNU Project. The graphics system came from the MIT X Window project.

But getting them all working together was up to the user. Sometimes this could be a real challenge. Back in those days, Linux users in many cities would hold meetings where Linux newbies were encouraged to bring their systems in to get Linux installed. Expert volunteers would be on hand, with pre-recorded CDs containing the software, and years of experience to draw on to get recalcitrant systems working.

This was fun, but it challenged all but the geekiest. It's hard to take a bunch of different software packages and get them all to work together. The more software involved, the more work it could be.

Distros to the Rescue

These difficulties slowed the use of Linux considerably. Linux was something for computer science grad students and electrical engineering undergrads, not the average computer user.

The development of an enthusiastic Linux community made the next step inevitable. Users started packing up their successfully integrated software systems, and mailing the CDs to whomever wanted them, or distributing the disc image over the Internet. Special graphical tools were developed to help new users install Linux, and sophisticated package-management systems took the sting out of getting and installing new software.

Since it's all free software, you don't have to own it to package it up and give it away, or even sell it. It's free, so go right ahead!

And that is what many users did. A set of integrated GNU/Linux software is called a distribution, or distro. And there are thousands of Linux users out there who, for commercial or altruistic motives, will take the trouble of putting together their own unique Linux distro.

Some distros are aimed at new users, and give only the most basic and simple tools. Others are aimed at the hard-core geek, with the latest bleeding-edge version of every cool software you've ever heard of, and hundreds more that you haven't. Some distros are intended to quickly convert a PC, and some to run on embedded platforms or supercomputers. There's a distro for every taste and set of needs and, if there isn't, someone is probably putting it out there right now.

Companies were formed that sold the distro and its support. They don't sell the software—that's free. They will sell you the smart person who can come down to your office and get it working on your hardware. Some of these companies developed even better tools to help the new user.

These days, there are hundreds of distros out there, each with its own supporters and (sometimes) detractors. Many distros are related to each other, each based on the same predecessor distribution. Many use the same package management or installation tools, but offer a different set of software tools to the user.

Which Distro Is Right for You?

Any distro makes it possible for any ordinary computer user to install the Linux kernel, the windowing system, plus a ton of extra tools and useful toys. No longer do you need to lug your PC to a Linux user meeting to get it working. The install tools are so well developed now that getting Linux working on a new system is fairly simple.

With all these choices, which distro should you pick?

A good list of an arbitrarily chosen top 10 best Linux distributions can be found at distrowatch.com/dwres.php?resource=major.

Unless you have special needs, the choice of distro is not too critical. Any of the top 10 will work about as well as any other. Remember, any distro is just a starting point. It gives you a base of software tools, but you may add any other free software to your system, often with a handy package-management tool that takes all the sting out of finding and installing software.

So just pick whichever distro strikes your fancy. If you're buying a new machine, take whichever distro they offer. If you have a geek friend who is willing to help, use whatever distro he or she likes (and has CDs for).

(If you still can't decide, pick Ubuntu (www.ubuntu.com) if you're a newbie, Fedora (fedoraproject.org) if this is a work machine, and Debian (www.debian.org) if you are a computer geek.)

Where to Get It and How to Use It

Where and how you get your distro depends on how you want to use it. The most straightforward way to get a distro is to order the CDs through the mail. For a nominal charge (usually less than $5 plus shipping), you'll get your distro in a few days, and can install from CD.

For the less patient, you can download an install a disc image (about 700MB), write it to CD, and have your installation disc without having to wait.

For the even less patient, Debian allows you to download a small install utility. Run it on your target machine (which must have an Internet connection), and it will guide you through the installation. I have never tried this, but it sounds so nice that I think I will next chance I get!

All of the distros have a Web site with instructions and downloads. They, or other third-party companies, offer the CDs for sale through the mail, and there is always the old-fashioned way—ask any Linux geek for a loaner.

Choices, Choices, Choices

Choosing a distro is just the first of many choices, but it doesn't limit you as much as you might think. The distributions are just different collections of already free software. If you see some interesting software and you like it, you can get it, whatever distribution you started with. As time goes by, you will install various packages on your machine, adding to those installed from your distro. Your collection of tools will become your own unique set of software.

Perhaps then, inspired by a need to give something back, and sure that your unique collection of free software offers value, you will produce and give away your own distribution!

System Administration

Once a user sets up a Linux system, he or she faces the tasks of system administration. Mundane chores mostly, but due to the highly-developed state of Linux tools there is a lot to learn. Many convenient features are available, but an orientation is required.

Backing Up the Linux Way: Sticking to tar

It's one of the oldest pieces of computer lore, handed down from the stone age of electronic computing: Always back up your files!

Sometimes the old ways are the best ways, and on my very first day of computing (all the way back in the last millennium), I heard a horror story that demonstrated the importance of backups.

Back at the Dawn of Time

I was starting work for a pair of atmospheric scientists who had something very rare: their own computer. It was a small computer (only about the size my wife's minivan), and required a room with its own power system, air-conditioning plant, and one of those computer room floors made of sturdy tiles—any one of which could be lifted to access the maze of cables beneath the floor.

It was about as powerful as my current laptop, only without the graphical user interface. For meteorologists, with their giant models of the atmosphere, it was what you might call a personal computer.

On the front were mounted two magnetic tape drives, and one wall of the computer room held row after row of neatly labeled magnetic tapes. These tape drives aren't even built anymore, but they used to decorate the front of every serious computer.

Those computer tape drives looked pretty high tech in the '60s, but they were a lot of work to use. The tape had to be threaded through the heads and onto the take-up reel. As the tape was written, it would be transferred back to the starting reel. Since they didn't hold all that much data (by modern standards), it was common to have data sets spread across multiple tapes.

Oh, the Horror!

My first day in the lab, I saw one of the scientists loading and unloading tapes onto the machine. He was backing up the disks on tape, and he did it every day, though it took almost an hour, and he was always pressed for time.

When I asked why he didn't have one of the lab's students perform this chore, he told me his tale of woe: A grad student, assigned to the task, had come in early every day to perform the backups. This grad student, seduced by his warm bed one winter morning, had skipped a day, with no untoward consequences, and without even attracting the notice of his late-sleeping co-workers, who assumed the backup had taken place as usual. The exception became the rule, and soon this lazy grad student had not performed a backup for three whole months—a fact that was discovered when the disk crashed, wiping out three months' work for the whole lab.

Now, this scientist was a bit of a forbidding character—not someone I would like to cross. And the steely glint in his eye made me fear for the fate of the poor grad student. He had forgotten the first rule of computing: Always back up your files.

The Modern Era

These days, disks are more reliable. I've never experienced mechanical failure with a modern disk drive, though it does happen, allegedly. The greatest risk to data today seems to be accidental deletion.

Although disks are quite reliable, the same can't always be said for the software engineer. Mistakes will happen, and when data is mistakenly deleted, a good backup discipline can save the day—and maybe your job.

There are many different programs to help back up your data, but it's hard to beat the ubiquity and simplicity of the old classic tape-archiving program tar. tar has been with us so long that the tape has vanished from the equation, and most home backup these days seems to be from one disk to another (or even onto one of those newfangled data sticks) rather than to magnetic tape. Yet, tar still gets the job done.

tar Me Up, Scotty

The simple idea behind tar is that you provide it with a directory full of files and subdirectories, and tar will package it all into one file, the "tarfile" (which usually has an extension .tar). The tarfile itself is in a very simple format—one that makes no attempt at compressing the data. For this reason, it is frequently run though a compression program, such as gzip, which produces a much smaller compressed tarfile, the "tarball."

The tarball can be saved to anther disk, perhaps on another machine. This ensures that if the original data is deleted or destroyed, the files can be recovered by uncompressing the tarball and then running it through the tar program again, which will unpack the file into its original directories, with all the contents intact.

There are many different versions of the tar program out there. Linux users have the GNU tar, one of the best. Not only does it meet all the tar standards, but it also includes command-line options to allow the compression to be done on the fly, without having to invoke the gzip utility separately.

Wasting Time

Although the scientist from my story always continued to run his own backups every day, I did manage to save him some time by showing him how to do incremental backups.

Incremental backups address the problem of backing up an unchanged file. Since most files don't change every day, there's no real need to back them up every day. But how can the poor user easily determine which files need to be backed up without making the tragic error of missing an important file?

With GNU tar, the problem is handled for you with the incremental backup feature. This means that you do a full backup at some reasonable interval, such as weekly, and then an incremental backup each day. The incremental backup will contain only files that have changed since the last full backup. To restore the files, you would first restore the full backup and then the incremental backup. Although this means you need to do more to restore your data, it also means that the daily backup is a lot smaller—and a lot quicker.

Using incremental backups, the scientist was able to use just one tape for his daily backups (except on Mondays, when he did his full backups). This meant less time in the computer room, and more time in front of his computer terminal, programming in Fortran.

More Ancient Wisdom

These days, with so much storage space available, it's not required to have someone come in early each morning and swap tapes around. However, it's still possible to mess up your backup process in some way that might leave you trying to answer some awkward questions to a fire-breathing boss who has just seen valuable data disappear.

Listen then, to the wisdom of times past. Always back up your files!

Who's Got Your Back?

We are putting more and more important data on our electronics, but are we taking more and more care that our data are safe?

Backing up your computer storage, like flossing your teeth, sending thank-you cards, or changing the filter in the furnace, is one of those important activities honored more often in the breach than the observance. It's just not very exciting - it's a chore. And this chore has been with us from the earliest days of computing.

Back in the Olden Times

One of my University computer jobs involved the backing up of the data on what was then called a mini-computer. These were between the toy-like micro-computers of the day, and the equally toy-like (but much more expensive) mainframe computers of the day.

The mini-computer looked like a mainframe - a bunch of refrigerator-sized chassis in the refrigerator-temperature computer room. (It was always lovely to go in there on a hot summer day!) To back it up I had to load reels of magentic tape on to the front of the machine, enter a command at the console, and go back to doing my math homework while the computer wrote data to tape after tape.

The consequences of a mistake could be severe - I was hired for the job after the previous incumbent was ignominiously fired. The disk failed, and it was found that he had not been making proper back-ups for weeks. As a result, weeks of scientific work had been lost.

Sometimes the Old Ways are the Good Ways

In the years since then, computational hardware has advanced at a dizzying pace. The giant set of disks (which I spent so much time backing up) were a massive 300 MB. Today I have a half-dozen data sticks much larger than that, just kicking around my desktop and briefcases. They give them away at conferences and trade shows.

But the need for back-ups has not changed, because people have not changed. Our hardware is more reliable, but we are not. One of the largest causes of lost data is human error. Someone deletes something that they shouldn't. And even our vastly increased hardware reliablity is not going to compensate for the fact that we now carry our computers everywhere.

My old mini-computer had disks that would crash if you looked at them the wrong way. My most recent computer, Yum-yum the EEE net-book, doesn't even have a disk drive - its all flash memory. I can drop it on concrete and it would still work. But unlike my old mini-computer, which sat in a secure building, Yum-yum comes with me everywhere. It's less likely to break, but far more likely to get lost or stolen.

Backing up my data is more important than ever - but I have even less time available than that harried student I was in my youth. How can I have the back-ups without the tedium? As with so many questions these days, the answer is the Internet.

Sometimes the New Ways are Better Ways

These days, we don't back up to tape any more. We demand near-instant retrieval of our backed up data, which is hard for tape systems to manage. We also benefit from super-cheap, super-dense disk technology. It's easier and cheaper to back up data to a disk server than it is to back it up on tape.

Most of use, though, don't have any disk servers running in our garage, and this is where the Internet comes in. Why should I run a disk server when there are so many people out there who can do so more reliably than I can? And most of us need to back up only a trifling about of data anyway. Disk servers are most economical in the terabyte range, and I would be lucky to have a gigabyte of data to back up, a mere one-thousandth of the capacity of even the lamest disk server.

Hence the rise of companies like Mozy, the on-line backup people. For about five dollars a month, you can have your PC backed up over the Internet, on to Mozy servers. If you ever need any of your old files, they will be right there for you. Businesses have to pay more, but, if your

business is data-centric, the cost is trivial compared to the benefits. Your data are backed up remotely, and safely, with very little investment of your time.

What You Must Still Do

Although you don't have to sit in front of tape drives waiting, there is still one task that you must pay attention to: specifying what to back up.

Most of the stuff on your hard drive is not worth backing up, because most of it is software, installed from disk or over the network with complex installations programs. These days, software is rarely just a single executable file. Backing up these files makes no sense, since you usually cannot restore them in a useful way. When these programs get messed up, you reinstall from the disk or over the network, the same way you originally installed the program.

The only files that need to be backed up are the personal files that you have added to the computer: your documents, pictures, music and videos. In the Windows world, these are all usually kept under a folder called "My Documents". In the Linux world, these will be under your home directory.

When setting up a backup, make sure that all the data you want to back up can be written to some directory under My Documents, and, if not, make sure that you add the appropriate directory to the list of what is backed up.

Cheap, or Just Frugal?

For those of us that don't have a data-centric business, but only the usual data-centric lives, is there a solution that is even cheaper? Of course there is, and it is called Google.

My data-centric life has involved many a lost computer file, and much wailing and gnashing of teeth as a result. When Google docs came out I saw the immediate benefit of having someone else manage all my data files. For me, Google has it all.

These days, Google will even let you store any old data file on their servers, up to a gigabyte for free. Well, that's very nice of them. By using Google docs for spreadsheets, word processing, and presentations, using the Google Picasa and You Tube for photos and videos, and the extra Google gigabyte for everything else, the enterprising computer user can get very nice on-line backups for free!

Privacy? Who Cares!

This solution sacrifices all notions of privacy to Google. Can they make any use of all these documents? I really don't know - but I note that they are very good at extracting information from large numbers of documents.

In my case, I can't see that it matters. Mostly what Google will find, looking at my large collection of on-line documents, will be a lot of articles for ComputerEdge. Since they are intended for publication anyway, I really don't mind of Google looks at them. Or, for that matter, my work documents, none of which involve anything secret.

If I did care more about my privacy, I could encrypt my data before sending it to Google, and be reasonably confident that no one outside the National Security Agency could read it. And if anyone at Google is reading this, how about some more storage? One gigabyte isn't much!

Staying on Top of Things

There is always the waiting. It is always there, always part of the computer experience. From the time that you wait for your computer to boot up until the time you wait for it to shut down, using a computer involves a lot of waiting.

If you've got a decent system (which means it's probably running Linux), and aren't doing anything too demanding, these waits might be almost impossible to notice. They may last no longer than the blink of an eye.

But if you are on a system with a little less power—perhaps a beloved ultra-portable—then you may wait a bit longer, a bit more often. And if you are a little more demanding, if you run a few compiler jobs, are editing 50 buffers with Emacs and have two dozen tabs open in Firefox, well, you too might start to notice the waiting.

What the Heck Is Going On?

Sometimes it's pretty obvious what you are waiting for, and sometimes it's not at all clear. Sometimes you can do something about it, and sometimes you really can't. The first step is to try and see what is going on—what exactly are you waiting for?

The best way to do that is with a program called top—just open a terminal and let it run.

The top program will take the whole window and will update itself every three seconds. When you are done with top, hit the q key to end it.

The Summary Section

The top five lines of output show a summary of what is going on with the system. The number of users logged on is usually one for most Linux systems, but remember that Linux also runs on vast multi-user machines, and top must be able to cope with that. The load average is an arcane way of measuring how many things are waiting on the CPU for the last five, 10 and 15 minutes. Interpreting these numbers is something I leave for the experts.

The next line tells you all about the tasks—that is, the individual programs running on your computer. A Linux computer running X Window will generally have a lot of tasks, but a few should be running. Don't worry about the zombie tasks; your computer is not going to try and eat your brain.

Next comes the summary of your CPU and how busy it is running user code (that is, programs you launch), system code (operating system stuff), nice code (which someone very pleasantly chose to mark as less urgent with the nice command), the time spent idle (usually the largest number), and the time spent waiting for I/O and serving hardware and software interrupts (which should all be near zero under normal conditions).

The memory is summarized next, with the total amount available, the amount in use, the amount free, and the number of buffers currently existing. Don't worry about the number of buffers, but the amount of memory in use and the amount left free are important numbers. When you have almost all your memory in use and very little free memory, it means you should buy more RAM.

Finally, the swap disk is summarized. The swap disk is there to handle overflow from RAM. If you start too many programs, and are trying to use more RAM than you really have, the computer will take something you haven't used for a while and write it to the swap disk, freeing up the RAM. Whenever you try and access that memory, the computer will sneak off and grab it off the swap disk.

In my case, there is no swap disk. This is a peculiarity of my system, the ASUS Eee.

The Task List

Under the Summary list is an ever-changing list of tasks, organized so that the most active are on the top. Each task (or process, as it is sometimes called) is assigned a number when it is started. The number has no particular meaning, it's just a way to uniquely identify any task. The first two columns of the task list show the process ID of each task, and the user who started it. Then comes the priority and niceness of the task, then the amount of memory the task takes up in memory, how much of the task is still resident in memory (as opposed to being sent to the swap disk), and how much of the tasks memory is also shared with one or more other tasks.

Next comes the most important two numbers: the percentage CPU and the memory that this task is using. These will tell you which task is causing you to wait. The final two columns show the cumulative CPU time used by the task, and the name of the task (the program that is running).

Interactive Commands

Top is one of those crazy Linux command-line programs that is just souped up to the max. There is practically nothing this command can't do, and its bewildering array of command-line choices and options will delight the Unix guru with hours to spend reading the man page and figuring out all the tricks that are possible with top.

Considering the program is targeted at system administrators, I guess it's not too surprising to find so many features hanging off the software. System admins can be downright obsessive about knowing what is going on in their system. These features can be used by hitting the right keys while top is running (so you will need a man page open in another window to try them out).

But every now and then, these little extras become just what you need. Recently I was wondering if I was taking full advantage of a dual-core system. How to really know? Turns out that top has an interactive command 1, which shows the usage of each CPU in the system. By seeing that they were both working hard, I was able to confirm that I was really getting the most out of the system.

Why You Wait

With a basic understanding of the top program, you are now able to answer the question posed at the beginning of this column: What are you waiting for when you wait for your computer? Just open top and look at the top one or two tasks, and you will have your answer.

The top program also allows you to see exactly how each program is using the computer—what it is costing you in memory and CPU to run each task on that machine.

Disk Space--the Final Frontier

Back in the bad old days, our disks were always full. How well I remember the 300-megabyte platter drive I had to use back in the '80s. It was about the size of a washing machine. This was, of course, part of a large computer, with its own dedicated computer room. After the personal computer came along, I was amazed when I saw my first external hard drive; it was about the size of two large hardcover books stacked on top of each other. And it held a whole 10 megabytes.

These days, you can get a 500-gigabyte drive for less than $100. We don't even bother cleaning our disks off—it's hard to imagine ever filling up 500GB. (Yet I know that in 10 years, I'll think of 500GB as a small amount of disk space!)

When disk space was tight, there were several tools we used to figure out which files to delete in order to free up some disk space. It's been many years since I've used them, because of the cheapness of giant disks, but with my little Asus Eee computer, there is no disk, only 4GB of static RAM.

And while 4GB seems like a lot (4,000 times larger than my first PC hard drive), I have already filled it after only a few weeks of owning the computer. Oh, deary me.

The Old Standby: df

In any kind of disk-space problem, the first command of the Linux guru will be the df command. It shows the amount of space on the file systems mounted on your computer.

When I run df on my computer, it looks a bit funny, because the Eee does some funny things with its disks. The df command lists each mounted file system, its total size (in KB, by default), the amount used and available, the percentage in use, and where the file system is mounted.

```
~ $ df
Filesystem           1K-blocks      Used    Available  Use% Mounted on
rootfs                 1454700      888792    492012    65%       /
/dev/sda1              1454700      888792    492012     65%      /
unionfs                1454700      888792    492012     65%      /
```

```
tmpfs                  254164        12       254152       1%       /dev/shm
tmpfs                  131072        72       131000      1%        /tmp
```

If you can' t figure out which filesystem applies to you, go to your home directory, and run the df command with a period as an argument. This will show the information for the current disk, the one that holds your home directory. And using the -h option gives the output in more readable form.

```
~ $ df -h .
Filesystem            Size  Used Avail Use% Mounted on
unionfs               1.4G  869M  481M  65% /
```

This tells me that I have only 492MB free on my computer. It sounds like a lot, and it is, but these days we're so greedy with disk space that it won't last long if I don't watch it carefully.

Yet I am seeing only 1.4GB total space, and I expect 4GB?

Some Eee Silliness

Something that is confusing about this picture is that the little Asus Eee that I am using is doing something a bit silly with its file systems. In order to be able to restore the original software on boot, the Eee keeps a read-only copy of all the software distributed with the system. This is typical in the PDA world of Palm, but not very usual with a computer system. With a computer like the Eee, there's really no need for this, as the computer can be restored from a USB device. I have found a way to remove this business and recover that disk space, but I haven't tried it yet.

Using du

The du command (which stands for "disk usage") will show you how much space is used in each subdirectory. Here's my accumulation of mail messages on my Eee. (Frightening—110MB after only a few months!)

```
~/Mail $ du -h
16K    ./friends
68M    ./rmiug
4.0K    ./drafts
2.7M    ./CLUE
12K    ./returned
24K    ./cygwin
188K    ./science
368K    ./school
600K    ./oracle
1.4M    ./other
11M    ./opera
9.3M    ./mingw
3.0M    ./spam
2.1M    ./archive/mail
2.1M    ./archive
12M    ./junk
16K    ./me
110M    .
~/Mail $
</pre>
```

For more information about the du command, try man du, but there's not much to this tool.

Using find to Catch the Big Fish

When cleaning up a disk, the find utility can also come in very handy. The find command has so many different arguments that reading the man page can be a bit scary! There are an astonishing number of ways to specify which files you want to find, and what you want to do with them once you find them.

Back in the old days, we used to use find to automatically delete large files after a few days of inactivity on the physics department computers. Today, that sounds like pretty rough treatment, but back then, disks were expensive and life was cheap.

For a simple use of find, just use the --size parameter. Then use something like +3M to tell it you want to find files larger than 3MB. (You can also use "G" for gigabyes, or "k" for kilobytes.)

```
~ $ find --size +3M
./.mozilla/firefox/6wi42rce.default/urlclassifier2.sqlite
./My Documents/My Videos/eeepc.wmv
./My Documents/My Music/amarok-tmp-0.170-1204660416186385&lid=751-usa&from=pls.part
./My Documents/My Music/Day & Night.wma
./My Documents/My Ebooks/manual.CHM
```

This is showing me my five largest files, and four of them look like stuff that came with the Eee that I probably don't need anyway.

Ultimately More Space Is the Answer

This little trip down memory lane (pun intentional) has reminded me what a pain it was to have to monitor disk space!

The ultimate answer is usually to just order some more storage right about the time you find yourself scanning for files to delete. Although you can usually get some useful space the first few times, it seems that eventually you end up filling the space with files you don't want to delete.

In my case, I've ordered a 2GB Micro-SD card for my Eee, which will give me extra storage, and I'm going to pull out the special Eee read-only filesystem, losing my restore capability by gaining back the disk space.

Hopefully that will keep me from reaching for these disk-monitoring commands for at least a few more months!

Closing the Linux Loophole

A robotic voice blaring, "Intruder Alert! Intruder Alert!" came from the Space Invaders video arcade game, way back in the last millennium, when computers were simple and a virus was just something that gave you the sniffles for a few days.

Now that we've developed advanced technology like the Internet and Russian computer crime, things have changed in many ways, but the notification of an intruder alert is something we need more than ever.

How Secure Is Secure Enough?

There is one security question every computer user needs to consider: How secure is secure enough?

For a home user who is reading e-mail, following the news, and banging out the occasional ComputorEdge article, not all that much security is needed.

At the other end of the spectrum, there are many systems that have a much higher need for security, for example, some of the computers used to develop military technology. In those systems, security is about as important as it can be.

Between these two extremes are commercial enterprises, universities, and research labs that work on non-classified projects. Each of these organizations will have different security needs.

Users must place themselves somewhere on this spectrum to decide how much time, money and effort should be devoted to securing their system.

More Secure Than Windows

Linux is justly famous as more secure than the operating system sold by Bill Gates, the world's richest geek. Although you might think that keeping your operating system code a secret would be more secure, it turns out to be the wrong thing to do.

System insecurities are, by definition, bugs in the operating system. The operating system with the least bugs (Linux) will, in general, be more secure than the operating system with the most bugs (Windows).

The fact that Linux code is available for all users to see is a powerful mechanism of code review. Countless eyes look at each line of code, each function, each module, for errors. Contrast this with a commercial operating system, where the team of programmers is much smaller—hundreds instead of hundreds of thousands.

Furthermore, each of those Micro-serfs has lots to do, and the corporation has a vested interest in fixing as few bugs as it can—and keeping defects as secret as possible.

With all those factors working against it, it's really no wonder that Windows can't keep up!

How to Practice Safe Computing

Once you have reached a decision about your security needs, it is time to devote some time and effort to tightening up your system. This involves careful reading of the documentation that came with your Linux distribution, and following all the good security advice that can be found there.

The most important aspect of safe computing is to keep your operating system software up-to-date. When a security hole is found, the software will be fixed, but that doesn't do you any good unless you update your system with the fixes!

The next step in securing a system is to turn off all the system services that are not needed. Services such as telnet, FTP, sendmail, and other old-time UNIX staples provide opportunities for hackers. If you don't need these services, turning them off is the easiest way to foil hackers who want to use them.

Setting up security is just the beginning. The system must be closely monitored to make sure that some hacker hasn't gotten around all the security you've set up.

How to Tell if Your System Has Been Cracked

Unfortunately, it can sometimes be hard to tell if your computer has been cracked. There are some obvious signs, such as suspicious entries in log files, or sudden increases in disk space and Internet activity, which can be tip-offs.

But these are the obvious traces of bad hackers; the more skilled hackers won't leave those details behind. If security is a very important issue, regular security audits by an experienced system administrator are needed.

Keeping up with the latest security news seems like a full-time job in itself, and this is just one more area where a motivated system administrator is worth his or her weight in gold.

How to Learn More

A good place to start is the Frequently Asked Questions (FAQ) list on the Linux Security Web site (www.linuxsecurity.com). Another source of basic information is the Linux Administrator's Security Guide (www.seifried.org/lasg).

A great place to keep on top of the latest threats is the CERT site (www.cert.org), a federally funded lab at Carnegie Mellon University devoted to finding and neutralizing computer security threats.

Stay Safe!

Even though Linux users have a built-in advantage over Windows users in the area of security, there's no reason to be complacent.

It's a dangerous world. Be careful out there.

Wi-Fi and Linux

I hate to admit it, but managing wireless networking is something that Windows and Mac OS sometimes are a little better at than Linux.

It's not that Linux doesn't have the capability the network anywhere that Windows or Mac OS does, it's just a little bit harder, and a little less obvious what to do. The graphical users interfaces are not always as intuitive as the Windows ones (yet - remember that Linux is always improving.)

The Graphical User Interface

All Linux distributions these days have some sort of graphical user interface which handles network configuration. The Asus EEE system I use, which has a specialized version of the Xandros distribution, has a utility called "Network Connections." It gives a very nice GUI which shows the wireless networks I've configured, and allows me to edit the details of each.

There's really nothing wrong with this GUI, except that there doesn't seem to be an easy way to get a list of all the available wireless networks. That is, if I am in an new environement, like a coffee shop, and I want to see if there are any wireless networks available, there seems to be no easy way to do that, except to start the wizard to create a new connection, and proceed through several windows of questions before getting the list of available networks.

Thing is, I need the list of available networks first, and I don't get it.

The Command Line to the Rescue

In the very old Western movies - the ones that were old even when I was a boy - the movie would always end with the U.S. Cavalry coming over the hill, with the bugle sounding "Charge!" (http://www.15thnewyorkcavalry.org/Media/charge.wav.)

That music plays in my head every time I am dissatisfied with a Linux GUI, and I open my bash shell to get a command line. I don't care what you are trying to do - in Linux, you can always do it from the command line. And in many cases the extra documentation and design of the command line tools help you gain a fundamental understanding of what is going on.

Even better, while GUI will vary from desktop to desktop, and may also depend on other installed applications on a machine, the command line is always the same. It's like having 100 cavalry troopers gallop over the hill, with pistols blazing, swords drawn, banners flying, and bugles bravely telling all your problems to get the heck out of the way.

When that happens, your problems suddenly don't seem so big after all. And that's what happens when you go to the Linux command line tools.

The Wireless Trooper: iwlist

The command to start with is iwlist. The iwlist command lists all the wireless networks that your wireless card can find. It gives me an output like this:

```
<pre>
yumyum:/home/user> iwlist ath0 scanning

ath0      Scan completed :
          Cell 01 - Address: 00:1B:11:58:1F:B5
                    ESSID:"michelle"
                    Mode:Master
                    Frequency:2.412 GHz (Channel 1)
                    Quality=18/94  Signal level=-77 dBm  Noise
level=-95 dBm
                    Encryption key:off
                    Bit Rates:1 Mb/s; 2 Mb/s; 5.5 Mb/s; 11 Mb/s; 6
Mb/s
                              9 Mb/s; 12 Mb/s; 18 Mb/s; 24 Mb/s; 36
Mb/s
                              48 Mb/s; 54 Mb/s
                    Extra:bcn_int=100
                    Extra:wme_ie=dd070050f202000100
          Cell 02 - Address: 00:1B:FC:CE:DC:35
                    ESSID:"Thunder Lake"
                    Mode:Master
                    Frequency:2.437 GHz (Channel 6)
                    Quality=17/94  Signal level=-78 dBm  Noise
level=-95 dBm
                    Encryption key:on
                    Bit Rates:1 Mb/s; 2 Mb/s; 5.5 Mb/s; 11 Mb/s; 18
Mb/s
                              24 Mb/s; 36 Mb/s; 54 Mb/s; 6 Mb/s; 9
Mb/s
                              12 Mb/s; 48 Mb/s
                    Extra:bcn_int=100
          Cell 03 - Address: 00:1B:2F:E0:12:14
                    ESSID:"NETGEAR"
                    Mode:Master
```

```
                    Frequency:2.437 GHz (Channel 6)
                    Quality=6/94  Signal level=-89 dBm  Noise level=-
95 dBm
                    Encryption key:on
                    Bit Rates:1 Mb/s; 2 Mb/s; 5.5 Mb/s; 11 Mb/s; 6
Mb/s
                              12 Mb/s; 24 Mb/s; 36 Mb/s; 9 Mb/s; 18
Mb/s
                              48 Mb/s; 54 Mb/s
                    Extra:bcn_int=100
                    Extra:ath_ie=dd0900037f0101001dff7f
          Cell 04 - Address: 00:0F:B3:5B:08:39
                    ESSID:"ACTIONTEC"
                    Mode:Master
                    Frequency:2.452 GHz (Channel 9)
                    Quality=38/94  Signal level=-57 dBm  Noise
level=-95 dBm
                    Encryption key:on
                    Bit Rates:1 Mb/s; 2 Mb/s; 5.5 Mb/s; 11 Mb/s; 22
Mb/s
                              6 Mb/s; 9 Mb/s; 12 Mb/s; 18 Mb/s; 24
Mb/s
                              36 Mb/s; 48 Mb/s; 54 Mb/s
                    Extra:bcn_int=200
```

</pre>

The thing to look for here is the name of the node (or the ESSID, as it's known in computerese), and the encryption key. For my home network (ACTIONTEC), the encryption is on, as it should be, and the wireless card must be configured with the proper key before it will work.

Meanwhile, my neighbor Michelle has an unencrypted node. This is bad in a home network, but usual for free wireless networks, like those in coffee shops.

(Note that the iwlist command is located in the /sbin directory in my distribution (as is the iwconfig command discussed below). That's not in my path unless I log in as root, and on some machines, it's not in root's path either until you explicitly add /sbin to the PATH.)

With the iwlist command I can find out what networks are available. Once I know, what do I do next?

The iwconfig Command

In the cavalry of Linux system administration tools, iwconfig is the trooper to help out with wireless networking configuration.

There are several important settings with wireless networking: the network name (called ESSID), the frequency, the mode, and the access point, as well as some other parameters that are less frequently used. The iwconfig command lets you set them all. When run without any command line options the iwconfig command will show current settings. By using the command line options, you can change parameters and configure for any wireless network.

The most common way to use the iwconfig command would be to set the name of a network. For example, if you are in a coffee shop with a free, unencrypted, wireless network called "coffeeshop", you can configure wireless like this:

```
<pre>
iwconfig ath0 essid coffeeshop
iwconfig ath0 key off
iwconfig ath0 ap any
</pre>
```

The Ups and Downs of Networking

Once you've got your wireless configured, use the ifup and the ifdown commands to bring the interface up and down.

In most cases, I use the command line to gain a better understanding of what is going on in the system. Often, with the increased knowledge gained at the command line level, I can go back to the GUI and quickly sort through the jumble of menus and buttons to find what I need.

Getting More Information

The trusty man command, like the knowing cavalry sergeant, can tell you everything you need to know. Just type "man iwconfig." I often open a separate bash window for man commands, so that I can leave the documentation open on my screen while I compose my command in a different bash window. If you are operating without any GUI at all, the same can be achieved with the screen command, or by using an editor like emacs.

Now that you have a better understanding of command line handling of wireless networking, perhaps you'll one day call in the cavalry!

Text Manipulation

When it comes to text editing on Linux, there are a number of popular choices, including vi and emacs. Never one to shy away from expressing a preference, I have always been a strong proponent of emacs. It is arcane and hard to get started with, and has its own unique syntax and way of looking at the world, but it is still the best editor and integrated development environment (IDE) for Linux systems, for C programming.

Editor Round Up

In the early stone age, proto-nerds performed many important calculations with primitive devices called "TRS-80s", "VAXes", and "mainframes." Even back in this primitive era, as caveman programmers sat under trees, doing software development with surprisingly sophisticated flint tools, the text editor had developed into several distinct species. Some were headed for extinction, a few have survived, or spawned successful progeny, and remain a part of the text editing ecology to this day.

Ed - A Purists Editor

When I was a little boy, I was taken with all the other New Jersey school kids to see Valley Forge, and other Revolutionary War sites. While the tour guides and teachers droned on and on about something or the other, we all had the same question, and it always went unanswered: How did they fit in those tiny uniforms and beds they had on display? The uniforms in the display cases looked children's Halloween costumes. The bunks were about four feet long.

Were those soldiers really that small?

Ed is like those old soldiers - it fits into such a tiny space that you wonder if it is for real. In the case of ed, the space is computing space, and ed uses so little memory and processor that you could probably run it on a wristwatch. It doesn't even both displaying any text - you're just supposed to know what is in your document, from a previous print-out. (After all, if you don't know what's in there in the first place, why are you editing it?)

In all fairness to the developers of ed, the so-called "glass teletype" hadn't been invented yet, so there was no way to display text except print it our on a teletype. (And that's how they knew you had a print out of the code somewhere.)

These days ed can still be found on just about every Linux workstation, though I doubt there are any users out there. It's genes live on in a descendant called sed, a utility for processing text files.

vi and the Twisted Mutants it Spawned

Eventually the bright idea of hooking up a TV to a computer resulted in the glass teletype, After several years of using ed to edit their programs, some programmers came to a startling conclusion: displaying the text on the screen would be useful for editing. Instead of having to remember where everything was in the file, you could just look at it on the screen!

The visual orientation of this editor dictated its name: vi.

Although vi was an improvement on ed, it is a far cry from today's modern editors. In fact, due to it's bizarre command structure and weird way of only editing one file at a time leave my soul filled with horror. It's not uncommon for new users to be driven completely mad by an attempt to use vi.

Despite its numerous problems, the vi editor continues to lurk in its harsh ecological niche. It has even spawned a nightmarish array of descendant, including vim. Like a species of tapeworm, it has even succeeded in parasitically embedding itself inside another editor, with vi emulation modes.

Truly is it said: you can't have "evil" without "vi."

Emacs - The God Particle of Software

The CERN eggheads are firing up the Large Hadron Collider in their attempt to find the so-called God Particle. (does it makes anyone else a little nervous, the way they are constantly assuring us that they're not going to destroy the planet?)

If they were looking for editors instead of particles, the one they would find would be emacs.

Widely acclaimed as the best integrated development environment in the history of computer programming, emacs is a programmers paradise and playground. It's fun, it's effective, it does everything you can think of, and twice as much that never even occurred to you (but is quite useful).

With emacs, pogrammers don't ever leave editor - all aspects of program development are done from the compiler, coding, compiling, debugging, and running. Emacs was the first free software product released by the Free Software Foundation, and remains a core part of Linux/GNU. To get started type "emacs" at the command prompt, or look for it on your menus.

Emacs is extremely adaptable. It has been ported to just about every platform, and works over a high-bandwidth connection with X Window, or a low-bandwidth connection via an ASCII-based interface.

Pico and Nano

Back in the day, there was a Unix email program called pine, and it came with an associated editor, pico, for composing email. Many became enamored of this editor, and started using it for code development. Due to free software license restrictions, pico eventually became extinct, but it's direct descendant, nano, lives on.

The original pico had appeal in the days of very limited hardware. It was better than vi (which almost everything is), and not as slow as emacs used to be on days when the computer was busy. But that hasn't been a problem for about 25 years, so it's hard to see why nano lives on, yet it certainly does.

Many new features have been added along the way, of course. Now nano boasts more features than almost any other editor of the olden days, though by today's bloated standards it is still quite svelte. This is an ASCII-only editor, those who need a fancy GUI need to move on to another choice. Nano has its own web page at http://www.nano-editor.org.

The GUI Accessories

If you are reading this on a Linux box, take a look in the menus, perhaps under something like "Accessories," and you will find a text editor. On KDE systems this is a program called kate, one GNOME systems, gedit.

In the world of commercial desktop computing, the free accessories which come with the operating system are minimally functional. After all, they aren't making any extra money on that software, so there's not much motivation for a commercial company to continue to improve and develop them. Linux, as is so often the case, is a different kettle of fish entirely. In the Linux world really great programmers will continue to develop tools, just for the bragging rights, resume bullets, and sense of pride that come with contributing to a free software project.

Given the usual dynamics of Linux software, it's not really surprising to find that these GUI accessories are full-powered text editors, with some interesting features. They can handle multiple files, code syntax highlighting, and spell checking. Unlike all of the text editors above, these editors work only with an X Window connection - they are graphical only.

Kate has a web page at http://kate-editor.org/; gedit's web page is at http://www.gnome.org/projects/gedit.

Eclipse - A Very Fancy IDE

Started by IBM in 2001, the open-source eclipse editor is an attempt to rebuild the IDE from scratch in Java. The result is impressive, and certainly equals or surpasses the Microsoft IDE, Visual Studio, which it much resembles. Although the editor itself is written in Java, it may be used to code in any language. The fact that it's built in Java has no impact on what type of files can be edited.

The Java base does have one implication: it is inherently slower than C/C++, the language used for most other editors. However this is not noticeable on high-end developer workstations.

Other Editing Critters

There are so many other species of text editor in Linux, it's a real testimony to the ecological richness the free software. Some other interesting specimens include jEdit, a Java IDE written in Java (http://www.jedit.org/), NEdit, a Linux clone of Windows text editors (http://www.nedit.org/), Eddie, a clone of the Macintosh development IDE

(http://www.el34.com/index.html), Scribes, a very focused text editor that tries to automate things (http://scribes.sourceforge.net/), Diakonos, which aims to be easier to configure than emacs, more powerful than pico or nano, and not as cryptic as vi (http://purepistos.net/diakonos).

So go out there and edit people! Perhaps the software you create will be the next great editor.

Who Needs a Fancy-Schmancy HTML Editor?

My 8-year-old niece asked me a difficult question. Upon being told that I was a computer programmer, she asked, "But what do you do all day?"

I tried to explain it to her, but from her incredulous look, I gathered that she didn't quite believe me. The adults around the room laughed, but I'm not sure any of them had a better understanding of the process. They were just more used to the ridiculous idea of paying a grown man to sit and play with a computer all day.

What Do Programmers Do All Day?

Programming is mostly about editing text files, and that is as true for the fancy graphical Web site as it is for the most boring, old application that doesn't even have a graphical user interface.

That is, instead of pushing pretty buttons, most programmers just type and type and type, without once touching the mouse.

I love movies like Jurassic Park, where they show a fat, sloppy, obnoxious programmer with his computer screen in the background. Such scenes look totally fake to me, not because I've never known fat, sloppy, obnoxious programmers, but because the screen in the background had a lot of fancy graphics—even a dancing hula girl at one point!

What geek ever does graphics? When I'm programming, the screen is just white text on a black background, and the mouse might as well be disconnected from the machine. And nothing ever dances or blinks. It just sits there, a bunch of letters on the screen.

Trying to Make Programming Pretty

These days, fancy editing environments are all the rage, and editors for Web developers are a prime example of the proliferation of the so-called Integrated Development Environment. This is an attempt to make programming as easy as balancing your checkbook, writing a letter, or turning on the door locks in a dinosaur theme park.

Programming, however, is a bit more complicated than that, and Web programming is perhaps the most complicated type of programming. It involves dozens of different software packages, all working together to take text and graphics files, prepared by the Web programmer, and convert them to what the users see in their Web browser. And although the Web is a very graphical environment, a surprising amount of Web programming has nothing to do with graphics—it's just text processing.

A Web page, for example, is really nothing but a simple text file, until it is served up in a browser. The browser understands the file and the computer user enough to make some decisions about what the resulting Web pages will look like.

Back to the Basics

When it comes to processing text files, there's no school like the old school. Text processing is something that any programmer from 1980s onward can tell you all about.

In olden times, programmers used various different tools to perform the tasks that make up programming. These tools are all embodied in the general UNIX principle: Do one thing, and do it well. Lots of programmers still abide by this.

Learning Curves and How to Flatten Them

Learning how to use all these tools is a significant burden on the new programmer, as well as a fertile field of mistakes and bugs. Thus, the idea of a graphical users interface that hides all the tools beneath windows, wizards, and nicely written help files.

It's just a pretty face on the same old tools, but it allows new programmers to get up to speed very quickly, and to use the tools in an intuitive way. Experienced programmers will also claim that the integration of all the tools into one application allows them to work faster and to accomplish tasks more easily.

The idea is to have one application that can handle all the needs of a typical Web programmer, just as Microsoft attempts to handle all document-preparation needs with its Word application, and all the financial needs of users with Microsoft Money (and what does Bill Gates know about, if not money?)

This explains why such programs have little appeal to old-timers such as myself. We have already climbed that learning curve. And no matter how pretty the face of the latest graphical tool, I wouldn't trade it for the integrated development environment that I use.

What Do People Who Are NOT Cranky Old Geeks Use?

In the world of Web programming, the Dreamweaver application is the industry-standard Web-development tool.

It presents the user with a very typical Microsoft Windows interface, from which the entire range of Web development tasks can be undertaken. Files can easily be uploaded and downloaded from the remote hosting site; and the text in the files can be edited in a nice HTML editor, which changes the color of the text in accordance with its HTML meaning. (For example, commented-out HTML is all one color, while tags are a different color from the rest of the file, etc.)

There's really no doubt that, for the average computer programmer in these modern times of graphical user interfaces, a program like Dreamweaver is the best way to do Web programming and HTML editing.

But it's not what I use.

Emacs: The Programmer's Secret Weapon

Back in the mists of time, back when the Watergate scandal was still in the news, and only a few months after President Nixon resigned from office, Richard Stallman, a crazy lunatic or a brilliant programming genius, or perhaps both, began work on a new editor called Emacs—which would, in fact, become the first truly integrated development environment.

The Emacs editor (www.gnu.org/software/emacs), which is still actively developed by Stallman, continues to lead the way in features for programmers, while remaining completely backward-compatible.

Emacs has long since acquired a graphical face for those who wish to use it. (Not me. If text was good enough for my grandfather, the first programmer in the family, then it is good enough for me!)

The Emacs editor provides the usual features of a programming environment, such as Dreamweaver. But the great thing about Emacs is that it works the same for every type of programming. Whether I am doing Web sites or old-fashioned FORTRAN programming, Emacs provides all the tools needed.

Emacs is smart enough to adjust itself to the work that you are doing.

Perhaps best of all, Emacs is free software. I have the source code and can compile it any time I want. It runs on every computer system I have ever programmed on, from old VAX systems, to the latest Windows release.

You can download it right now, if it's not already on your system.

And no one can make it backward-incompatible on you, not even Richard Stallman. No one corporation can hold all of your work in the palm of its sweaty, corporate hand.

* * *

I could have answered my niece's question by saying I work in Emacs all day, but I don't think she would have understood that either. You have to work with it to understand what a great tool it is.

Emacs: The Only Program You Need

Remember Mr. Spock? He used to look into that primitive computer thing and pull out answers that saved Kirk's butt again and again. It was always an amazing demonstration of computing prowess. It can finally be revealed what Spock was running on the Enterprise's computers: emacs.

This explains Spock's facility with computers! It wasn't that he was Vulcan, it was because he was an emacs user!

What Is Emacs?

Emacs is a text editor, but saying emacs is a text editor is like saying the Pacific Ocean is wet. It is more than an editor—it is a super-editor, an über-editor. It's the one that all the other editors wish they could be, but know they never will be. It is the top-banana, the tip of the spear, the big enchilada. It is not just a text editor. It is the text editor.

The primary audience for text editors is the computer programmer. Computer programs, though ephemeral, expensive and exasperating, are the core of all our coolest technology. And, when it comes right down to it, they are text files. Emacs is ultimately about software development. And it is the best software development environment, by far, ever to be developed.

To say emacs has a loyal following is to dramatically understate the case. I've used emacs for the last 25 years, nearly every day. If you took it away and made me use another text editor instead, I would use it to write emacs from scratch, rather than go on without it. That's the kind of user loyalty Microsoft would kill for. But they will never get it, because they can never produce any software a tenth as good.

Where Did It Come From?

Emacs was the beginning of the free software movement. It was the first application released by FSF (www.fsf.org) (Free Software Foundation), the first free software application that soared to a big, fat mind share.

Back in the olden times, before X Window provided a desktop interface for Linux, emacs just used the technology of the day—ASCII terminals. However, with version 20 of emacs, the X Window graphical user interface was fully supported by emacs.

Getting Started

To get started with emacs, type "emacs" at the command prompt. You will get an emacs window.

The most important thing to know is how to use the so-called "meta" key. The meta key is like the Control key (or "Ctrl," as it is abbreviated on the keyboard). As with the Control key, you hold down the meta key while hitting another key. For example to hit Ctrl-x, (or C-x, as you will see it called in emacs documentation), you hold down the Control key, and hit the "x" key. To do meta-x (or M-x), you hold down the meta key, and then hit the "x" key.

The only problem, as quick readers will already have perceived, is that there is no "meta" key on their keyboard. This key existed on an early ASCII terminal that emacs was developed on, but it didn't catch on, and so now there are no keyboards with a meta key.

No matter; with emacs there is always a way. On most computers, the "Alt" key will function as a meta key for emacs, so M-x can be achieved by holding down the Alt key and hitting the "x." In case the Alt key doesn't work, you can use the escape key ("Esc"), but it a slightly different way. Instead of holding down Esc and hitting the "x", first hit the Esc, and then hit the "x."

Awesome for Programmers

Now, why would programmers put up with all this business of meta and control keys? What are the features that emacs offers that make it worth the trouble? They are far too many to list.

The first are the editing modes. If you open a file named program.c in emacs, you will be in something called C-mode. This gives you all kinds of commands for programming in the C language, and turns on formatting that looks good for C code. If you open program.f, you will get Fortran mode. If you open program.java, you will get Java mode, etc. Almost every existing programming language has its corresponding mode. And here's the neat thing: The same commands work for every language.

For example, the M-x comment-region command will turn a region of code into comment, even though every language uses different characters for comments. M-x indent-region will cause a bunch of code to have good indenting for whatever programming language you are working in. And there are many, many more.

Starting a Life of Productive Programming

Emacs can be a bit confusing to get started with. It's a serious power tool, and requires some dedication and effort to use properly. But give it a try—you won't regret it!

Using the Shell

In the days before graphical user interfaces, the command line shell was the way we interacted with computers. On Windows, the command line shell has been relegated to the DOS command window, but on Linux, the shell is alive and well. Many advanced users prefer shell interaction over GUIs because of the power, ease, and flexibility of the command line.

Coming Out of Your Shell

One of the great things about Linux is the variety of ways in which you can interact with the software. In some operating systems (and I will point no fingers!), there is only one way to get the job done. Functional, but boring. On Linux, it is never this simple and always more exciting.

Last week, I described some of the different graphical user interfaces that Linux users can enjoy, but the graphics in Linux are just a layer on top of a more basic command-line interface.

What's a Command-Line Interface?

Predating today's modern world of fancy graphics, the command-line interface hails from the old days—when men were men, and computers were the size of minivans. In those days, the so-called "smart" terminal was all the rage because it could achieve such amazing visual effects, like blinking text, different-colored text and even bold text. Programmers of that era were very close to the hardware. They often used the specs from the hardware manufacturer, and each type of machine had its very own operating system.

Back in the 1970s, a programmer named Ken Thompson was working on the development of a new operating system—one that, for the first time, was not written for any particular computing hardware, but rather was generalized, so that it could work on any hardware. As a metaphor, Ken imagined the hardware system as the yolk, with the system software as the egg whites, but there was something else needed—something between the egg and the user. Something to wrap around all of the system's functionality, and be all that the user needed to touch. That something is the shell. In 1971, Ken Thompson released the first version of his eponymous shell, and the Thompson shell was the basis for all succeeding UNIX command-line interfaces.

Pipes and Redirects: the Plumbing of UNIX Systems

One thing that the shell must do is allow the user to run programs on the machine. In fact, this is the main purpose of the shell. If you have a program called "trek," then the user must be able to type "trek" somewhere, to launch the program—in this case, a classic Star Trek game.

But the Thompson shell did more than this. It allowed users to send a file into a command, just as if all the lines in the file had been typed on the keyboard. It allowed the output of a command to be saved or appended to a file. And it allowed the output of one command to become the input of another.

With these capabilities, the shell turned into a powerful tool for assembling complex command sequences. Simple tools, used in complex combinations, can do sophisticated data processing. This is the basis for the success of the UNIX operating system.

Meeting the Families

In the UNIX world, there are two families of command-line interface, both derived from the original Thompson shell. One is called the Bourne shell (sh), and the other, the C shell (csh). From these two early shells, a whole coral reef of free software has been built, and many variants exist that improve on these two shells. Some of the choices now include the TENEX C shell (tcsh), the Bourne Again shell (bash), the Korn shell (ksh), and the Z-shell (zsh). Each of these variants of the shell offers its own set of features, its own way of helping users automate their common tasks and string together different UNIX tools to do one complex job.

In spite of all these alternatives, there are still some diehards out there who stick with the original shells out of some sort of stubborn tribute to the past. We all can respect the past, but that doesn't mean we walk around with wigs and tri-corner hats. Sometimes the past has been improved upon, and that's the case with UNIX shells. (I'm a bash man, myself.)

Now You're Speaking My Language!

After typing in a complex series of commands, you long for a way to automate the tedium of doing so again. And this leads, quite naturally, to needing a way to generalize and pass parameters to the commands, so they can be adapted for various circumstances.

The shells that came after the Thompson shell started to include the features of procedural programming languages—variables, branches and loops. A variable is a named area of memory that can store a value that changes (i.e., varies). The branch is an if-then decision. If some condition is true, then execute some code; otherwise, execute some different code. The loop is a way of repeating code as many times as needed.

With variables, branches and loops, procedural programming becomes possible. This allows the automation of complicated and tedious tasks, and allows the programmer to encode ways for the computer to do the "right thing" under various circumstances.

There are many existing examples of shell programming, and many existing systems that are based, in part, on sophisticated shell script programming. Professional programmers will prefer to use more feature-rich languages (like Java) for really complicated programs, or languages that make more efficient use of the computer for better performance (like C). But any programming task can also be performed with the various shell languages, and sometimes it's quicker and more convenient to do so.

Programs for the shell are usually called scripts, and shell scripts can quickly become indispensable to the computer power user.

The DOS Shell: A Hastily Hacked Toy

One of my college buddies has a great term for the kind of watered-down math classes that business majors take. He called it "let's pretend math." The same term can be applied to the Windows version of the shell, the DOS shell. Reportedly, it was written by Bill Gates on a flight to meet the IBM engineers who would be using it on the new (at the time) IBM PC. I don't know if the story is true, but I have little trouble believing it. Any UNIX user who encounters the emasculated features of the DOS shell will have no trouble believing that it was written on an airplane, with all the distractions and limitations that might imply. The DOS shell is a "let's pretend" shell, with little of the power and elegance of the UNIX shells, which are so shamelessly copied. The DOS shell is an inferior knock-off.

The Shell Today

These days, just about everything on a Linux box can be done through whichever GUI you are using, but that doesn't make the command line useless—it is a tool for experts. It allows you to run commands with a powerful and terse syntax, and provides a wide variety of tools to help you customize your interaction with the computer.

The shell is still there, just as it always has been, waiting to provide a powerful interface to the user who is willing to take the time to get to know it.

Fun With Bash Prompts

Every command-line interface must have some sort of prompt string—something that is printed out on the terminal to indicate that it is your turn to type—to tell the computer what to do. You then issue a command, and when it is complete, the prompt again appears to let you know that the computer is ready for more input.

When I start my bash shell, I see: ~> .

This is my directory ("~" means home directory), and then an angle-bracket and a space indicates the end of the prompt. (The space is important; without the space, the command line looks very crowded and ugly.)

As with most everything in Linux, this can be customized by the user. You change your prompt by setting the environment variable PS1, and there are many neat things you can do to make the prompt more useful and interesting.

Setting an Environment Variable

To set an environment variable in bash, use the export command. Once you set the PS1 variable, the command line immediately changes.

```
~> export PS1='what is your command human? '
what is your command human?
```

It's important that you don't put spaces around your equal sign (=) when using export. There must be no spaces between the variable you are setting, the equal sign, and the value you are setting it to. Also note the use of the single-quote character around the string value. Without those quotes, the shell would get confused by the spaces in the string, and would set PS1 to "what," and then try to process the rest of the string as additional arguments to export.

There are three quotes on your keyboard: the single quote, the back-tick (a single quote facing the other way), and the double quote. On a U.S. keyboard, the single quote and double-quote are on the same key, near the Enter key. The back-tick is on the upper-left, with the tilde. Each of these sets of quotes means something different, so make sure you are using the correct one.

To play around with prompts, you can set them from the command line, but to change your default prompt, you are going to have to set the PS1 variable in your .bashrc file.

By setting PS1, you can have any string as your prompt. But there is so much more!

Built-In Fields

The bash shell has a bunch of built-in fields for the prompt—special character strings that you can put into PS1 to change the look of the prompt. For example, if you want the time in your prompt, use '\D{}' in PS1.

```
what is your command human? export PS1="\D{}> "
07:54:11 AM>
```

There are dozens of built-in fields, such as the date and time field above (which can be formated by putting the correct string between the curly-braces).

Some useful fields are the hostname (\h or \H), the user name (\u), and the current working directory (\w or \W). There are many ways to get the date and time, or one or the other, in addition to the '\D{}' shown above.

Living Color

Setting the prompt also allows you to change the colors of the text, even the color of the background. It's a bit of black magic, but add the string '\[\033[1;31m\]' at the beginning of your prompt, and the string '\[\033[0m\]' at the end. The first blob of funny characters tells the shell to use light red as the text color, and the other blob tells the shell to turn off any color changes and go back to the default color. If you don't include this part, not only what you type, but also the computer's response, will be in light red.

Of course, light red is not the only color available—the complete list of color codes can be found in the bash man page. They range from black (0;30) to white (1;37), with many in between, such as cyan (0;36), brown (0;33) and light purple (1;35).

If you add a 10 to the color number, the background color is changed instead of the text color. For example, including '\[\033[1;45m\]' turns the background color to light purple. The '1;45' is the light purple color number (1;35), with 10 added to the second number.

I don't know who came up with this crazy way of setting colors in the prompt, because it could hardly be more confusing. But using both features together allows you to change both background and text color. For example, setting your prompt to '\[\033[1;45m\]\[\033[1;37m\]\d \[\033[0m\]' will give you the date in white, against a purple background.

In that funny-looking string, there are four parts. The first part, '\[\033[1;45m\]', turns on the purple background. The second part, '\[\033[1;37m\]', turns on the white text. The third part, '\d ', shows the date, followed by a space, and the last part, '\[\033[0m\]', turns off all the fancy color stuff so that the command line is shown with default color and background.

Prompt History Lesson

Back in the '70s, everyone was delighted with the invention of the first glass teletypes. They replaced the paper teletype—an IBM Selectric typewriter, modified to print out the computer's response underneath whatever you typed on the typewriter. The paper teletype used boxes and boxes of computer-fold paper, and sounded like a machine gun firing. (It sounds primitive now, but compared to punch cards, it was very high-tech.)

The glass teletype, as it was called, was the first cathode-ray tube monitor. The CRT monitor was a technology that had a long run, but is now almost completely obsolete, due to the LCD screen. At the height of its development (i.e., just as it's going obsolete), the CRT delivers super-sharp pictures in amazing color.

The first glass teletypes also had color: green. They displayed everything in the same ugly green text on the screen. But it wasn't long before the so-called "smart" terminal was developed, one that could let the user experience things like colored text, blinking text, etc. This was another technology that was approaching perfection and obsolescence, as the graphical user interface was introduced and the command-line interface became something that appears in just one window on a screen full of fancy gadgets and applications.

These days, no one makes a "smart" text terminal; they just emulate them on your general-purpose computer. Similarly, few rely fully on the command-line interface, but that doesn't mean you can't take advantage of the perfection of this technology to get your screen to look more fun.

Combining UNIX Commands

"Do one thing, and do it well." This is a statement that inspired early UNIX programmers. UNIX consists of a wide variety of tools, each extremely focused. But when you combine them, you can produce sophisticated results. The fundamental difference between Windows and UNIX is that Window is designed to be easy to use, while UNIX is designed for pure power.

But these powerful results wouldn't be possible without some way for these separate programs to share and communicate data.

Three Streams of Communication

Just about every UNIX tool can handle three streams of communication. One, stdin, is a way for you to provide information to the program. Another, stdout, is where the program sends its normal output. Finally, stderr is a way for the tool to send error information.

Redirection

Suppose you want to save the output of a command to a file. This is useful if you want to send a list of files through email. The greater-than sign (>) handles this in UNIX. Take a look at the following command:

```
ls -l > listing.txt
```

This will create a file called "listing.txt." The file will contain everything you would see on screen if you typed "ls -l". The greater-than sign (>) is called the redirection operator in geek-speak. It tells UNIX to send the output of a command to a file. In other words, whatever the ls program sends to stdout will be written into a file. If the file already exists, it will be overwritten. That is, the original contents will be destroyed. The prevent this, use two greater-than signs, which will append output to the file:

```
lp -l >> listing.txt
```

The redirection operator works in both directions:

```
lpr < listing.txt
```

This tells the computer to send listing.txt into stdin of the lpr program. It's not used as often, because programs that take input (like lpr) usually let you specify the filename on the command line. So the file can also be printed like this:

```
lpr listing.txt.
```

Piping

Sometimes you need to send the output of one command into another. You could redirect the output to a file, and then use that file as the input for the next command.

For examle, to get a sorted list of files, try:

```
ls -1 > listing.txt
sort < listing.txt
```

There is a way to do this in one step: the pipe operator. It does for data what a pipe does for water. It's the vertical line (sometimes found with a break in the middle) found above the "\" key on most keyboards.

For your sorted list of files, try:

```
ls | sort
```

This will give you a sorted list of files in the current directory. The pipe takes the stdout from one

command and sends it to the stdin of another.

Piping with Grep

The grep command searches through one or more files for a text regular expression. For example:

```
grep "howdy" *.txt
```

This command will search through all .txt files in the current directory for the string "howdy."

Often, I use two grep commands connected with a pipe. This allows me to refine my search. If the grep command above returned hundreds of hits, I might refine the search:

```
grep "howdy" *.txt | grep -v "partner"
```

The first grep will generate a list of lines containing the word "howdy." The second grep will act on that list and find all the lines that contain "partner."

The grep command can be told to show lines that don't match the criteria. To show all the lines of a file that don't contain "howdy," try:

```
grep -v "howdy" *.txt
```

To find the lines that contain "howdy" but not "partner":

```
grep "howdy" *.txt | grep -v "partner"
```

The Find Command

The find command goes beyond merely finding files by name. It can also find files larger (or smaller) than a certain size.

The real power of the find command is to provide input to the grep command (using xargs command along the way). This is regularly done by programmers. Take a look at:

```
find /usr/home/ed  -name "*.cpp"|xargs grep "variable"
```

This command will find all C++ source files under my home directory, and all subdirectories, and search them all for the string "variable."

More or Less

In the grand old days before the GUI, there were no slide bars on the sides of windows. Indeed, there weren't even windows! How then did early computer users examine long documents? They used something called "more." The more command will display any document one page at a time. For example:

```
more my_long_file.txt
```

As the years went by, the more command was improved in various ways, and released as a new tool called less. The less command works just the same as the more command:

```
less my_long_file.txt
```

More or less are used with commands that have a long output. A directory might have hundreds of files, in which case the ls -1 command will scroll right by, without giving you a chance to read it. Try using a pipe to more:

```
ls -1 | more
```

Now the output will be presented one page at a time.

More Text Manipulation

UNIX has a tremendous selection of tools to work with text- far too many to mention here. Any modern Web site developer or programmer can benefit from these tools, and the combinations that are possible with pipes. Text tools commonly used with pipes include the sort command; the uniq command, which can sort and eliminate duplicates in a list; and cat, which spits out the contents of a text file.

The tr command allows lots of fun manipulation of text, including changing case from upper to lower, change one character for another, or removing every instance of a character in the output.

Some Combinations

Once you get the hand of using pipes, it's amazing how much you can do with them. Three or more pipes can be used in a command. One common application is to search the output of grep with another grep. For example:

```
find .-name ".cpp|xargs grep "somestring"|grep "otherstring"
```

The above command will search all C++ files in and under the current directory for the string "somestring." It will then search the output of that for the string "otherstring."

To find the ten most popular words in a document, try:

```
cat file.txt|tr -c 'a-zA-Z' '\n' <file|grep.|sort|uniq -c|sort -nr|
head -10
```

Or, to count the number of occurences of words in text:

```
cat.file.txt| tr"\n" ''|sed's/[^a-zA-Z]+//g'|tr'"\n"|sort|uniq-c|
sort|tac
```

Once you develop a useful command, you can save it as a shell script. In this way you can develop your own set of useful UNIX command-line tools.

The command line is alive and will in the world of UNIX computing; use pipes and redirection operators to get the most of it!

How to Become a Linux Guru: Command-Line Tricks

Some of the new graphical tools for system administration are really terrific. They save me a lot of time when it comes to doing something that I've never done before. But when repeating a task over and over again, give me the command line every time. It's not just that I can type a lot

faster than I can move the mouse, it's also that so much of Linux is hardwired into my brain that I don't even have to stop and think about the command I want. It just flows.

When working from the command line, there are some tricks I frequently use that speed things up—though they add a little syntactic confusion.

To be a true Linux guru, take the time to learn a few simple command-line rules before giving up on the terse but powerful syntax of the Unix shell.

Variables and Shells

The first thing to be aware of is that a plethora of different shells are available. Each shell is slightly different, so it's hard to write about all of them at once. For the purposes of this article, I will discuss the popular bash shell.

Within your shell environment, the variables that are set change the way things work. Print a shell variable with the echo command. For example, to see your path, type:

```
echo $PATH
```

It's useful to have your EDITOR variable set to the name of your favorite text editor. I do this in my ~/.bashrc file (cshrc users should try ~/.cshrc). This is a file that is run every time I start the shell.

One trick is that there are two families of shells, and they have different (and incompatible) ways to set a variable. In my bash profile script, I can put:

```
export EDITOR=emacsclient
```

If I were a csh user (or a user of any of the csh family of shells), I would instead have something like this:

```
setenv EDITOR emacsclient
```

Quotes and How to Use Them

One of the most confusing thing for the newbie is the use of quotes. If you look carefully at your keyboard, you will see that there are actually three different kinds of quote marks: ", ', and `. In the Linux bash shell, each of these quotes does something different.

The single quote, ', does the least of any of the three quotes. It groups strings that would otherwise be separate because of the spaces embedded in the string. For example, to find the words "to be" in a file, you could use:

```
grep 'to be' file.txt
```

The double-quotes are like the single quotes, except that the string in quotes is searched, and any environment variables that can be found are expanded. So if you have an environment variable $VAR, and it is set to 'be,' then the following grep command will search for the string 'to be'.

```
grep "to $VAR" file.txt.
```

Finally, there is the single back quote, which does something completely different. Whatever is inside the single back quotes will be run as if it were typed into the command line, and the result will be inserted into your command. For example:

```
grep `whoami` file.txt
```

This will search for the string returned by the whoami command. The date command is frequently useful when trying to construct a unique name.

```
find . > out_`date +%Y%m%d`
```

And and Or

When executing commands, I frequently need to perform several commands, in order. If there are any problems, I want to stop processing immediately, not keep trying to execute the commands.

For this, the && (and) and || (or) operators can be very helpful. For example:

```
./configure && make install
```

This will run "make install" only if the configure command executes correctly. If there is a problem with the configure, the make command will not be run. Parentheses may be used to group the operations as well.

Usually the and operator is all you need, but occasionally you need to use the or as well:

```
(./configure && make install) || echo "failure!"
```

The Pipe and the Redirect

Frequently, the output of one command needs to become the input of another; this can be achieved with the pipe operator: |. Yes, that's the tall vertical bar, located somewhere around the backspace key on most American keyboards.

For a trivial example, this command will search the list of files and directories returned by the ls command for a string "pictures."

```
ls | grep pictures
```

The file redirects can be used to send the output of this command to a file on disk, instead of to the screen. This command puts the output of the ls command in a file called output.tmp:

```
ls > output.tmp
```

Running this command twice will cause the file output.tmp to be overwritten the second time. To append instead of overwriting, use the double greater-than sign:

```
ls >> output.tmp
```

It's not always clear, but there are two different sets of output being displayed on the screen, the so-called "standard output" and the "standard error" streams. (Usually abbreviated stdout and stderr.) The idea is that programs send normal output to stdout, and error messages to stderr. If

you do nothing, both of these are dumped to the screen, and you see any error messages mixed in with the command output.

If you automate some script, and attempt to capture its output with redirection, you will get the stdout stream, but not the stderr stream. This won't matter if everything works OK, but when something goes wrong, you will miss the error messages. One final trick is to redirect the error as well as the output stream. Do this in bash with the &> operator. For example:

```
ls &> output.tmp
```

will put the output of ls into the file, and also any error messages that were printed during its execution.

Using find and grep

Two commands that you can't live without are find and grep.

Find will find a file or directory by name, size, age and a host of other factors. For example, this command will find all files in my home directory, and all subdirectories, which have been modified within the last three days, and are greater than one kilobyte in size.

```
find ~ -mtime 3 -size 1k
```

The default action of find is to list the filename (including path) to stdout, but you can also tell it to perform various actions on the file. This is often used to delete files over a certain size that have not been accessed within a certain amount of time.

Grep will search one or more files for a text pattern. It will take a regular expression and search a file for anything that matches the regular expression. Regular expressions can become very complex very quickly, and are powerful ways to pack a lot of functionality into just a few keystrokes.

The xargs Command to the Rescue

Sometimes you want to pipe one command to another, but you want the second command in the chain to run on each line of input from the first, instead of once, for all lines of input. It's a subtle difference, but important.

If you want to find all the files that contain a certain string, you might be tempted to pipe the output of a find to grep. But that won't work, because you want to run grep on each file in the list, not on the list itself. The xargs command is made for this. Try something like:

```
find . -name something | xargs grep "some string"
```

This Old Clock

Automation is a good thing. So when you have a command that you run every day (except when you forget), use the cron utility to have the computer do it for you. Cron is nothing more than a list of commands that are run on a regular schedule. If you have your EDITOR shell variable set, run:

```
crontab -e
```

Your editor will be started, and your currently running cron jobs will be displayed (if you have none, you will see an empty buffer). Type in something like:

```
0 0 * * * ls ~ > ~/mydir.out
```

Learning More

When you have 20 minutes some day, look at the man page for bash or tcsh, or whatever shell you use. You will find that it contains a complete programming language, and numerous ways to automate or ease common tasks that fall short of the effort required to program.

On a Linux system, I find that the info pages often contain more useful documentation than the man pages. If you use emacs, try:

```
M-x info
```

If you don't use emacs (why not?), try the standalone info system with:

```
info
```

In It for the Long Haul

One great thing about computers is that there's always something new to learn, and one great thing about Linux is that you can be sure these tools and techniques will not go away—they will work in 20 years the same way they work now.

Unlike certain other operating systems, that makes Linux worthy of a little study!

Programming Tools

One of the great things about the ComputorEdge columns was the chance to explore new ideas and new ways of writing about programming and technology. I decided to try and teach the basics of Linux programming without ever getting too heavy for the ComputorEdge reader. I don’t know if I actually got anyone to try programming for the first time with these articles, but they are still relevant and still show the path to a career as a Linux software developer.

Software Development From 30,000 Feet

The Linux software-development tools are the most important part of Linux to understand. Without them, there would be no future for Linux—and no past, either.

What Is Software?

Software is the set of instructions that the computer hardware follows in manipulating the hundreds of millions of tiny, transistor-based switches that make up the microprocessor. The switches are just tiny machines that can be turned on and off. Any pattern or larger meaning in the position of the switches is for us to determine. And it is these patterns that allow us to create word processors, spreadsheets and games like Pong.

The programmer is the person who develops the set of instructions. It is a painstaking task, since the computer will do exactly what it is told—no matter how stupid. The instruction sets have to be nearly perfect to be useful, and it's this quest for perfection that occupies the programmers' time.

The instructions, or source code, exist as files on the computer disk. The files are processed by compilers or interpreters, and the computer carries out the instructions. You can't touch the software, or see it (except using other software), but without it the computer would just be a hunk of junk. An old saying about the difference between hardware and software will help you understand: Hardware is the part you can kick; software is the part you can only curse at.

The Development Tools at the Heart of Linux

When a new microprocessor is developed by the good people at Intel, AMD or some other chip designer, how does that end up as a functioning computer? A new microprocessor needs to be able to run existing software to be worth anything.

A computer language called C is used to get existing software on new platforms. The C compiler is a program that turns C source code into executables for the computer, and it is one of the first pieces of software to be ported to a new platform (using some assembly-language programming—far more primitive than C.) Once C is available, other tools (written in C) can be built. Using those tools and the C compiler, any application can be ported to the new platform.

Since everything in Linux is built from these tools, and since the software is freely available, you can pick and choose your tools, starting with a brand-new microprocessor and C compiler, and end up with a core set of functioning tools, which would allow you to build any other Linux software. Without its software-development tools, Linux literally couldn't be what it is today. The Linux development tools are used for much more than just porting, however.

Creating More Shrink-Wrapped Software

The term "shrink-wrapped" takes us back to the days when software was distributed in shrink-wrapped boxes in the computer store. That was before the Internet. They still sell those shrink-wrapped boxes because I see them in my local computer store. I believe the box now includes an old version of the software and the pass codes needed to download the latest copy, which is what everyone does.

In the Linux world, I have never seen shrink-wrapped software, though presumably there is some somewhere. I get all my Linux software for free, so I wouldn't know how it is sold.

The distribution channel may have changed, but the term shrink-wrapped remains. Shrink-wrapped software is a product. It's developed by one group of developers, and used by hundreds, thousands or even millions of users. The companies, universities and private individuals who

work to develop code for the Linux platform each have their own reasons for doing so, but for all of them, software-development tools lie at the center of their product-development process.

Creating More In-House Software

The amount of shrink-wrapped software that is written is tiny compared to the amount of software written for the in-house use of a business, university, government agency or other organization. This can be anything from specialized payroll processing to special reports for the CEO. All of this business activity relies on the development tools of the local computing platform.

Creating Web Sites

The Web has been adopted faster than any previous invention in human history. Cavemen sat around and debated the pros and cons of using those newfangled metal arrow tips longer than our civilization took to turn all our banking, reservations and business information over to this brand-new computer tool. Does that make us smarter now—or dumber?

All this has created a new field of software development: Web site development. Even a relatively simple Web site involves lots of software and at least a little programming. Any ambitious Web site is going to involve a lot of programming. The vast majority of serious Web sites are built on the ultra-reliable LAMP (Linux, Apache, MySQL and PHP) stack, and the L in LAMP is for Linux. Consequently, most Web site development is done with Linux development tools.

Creating Embedded Software

They are everywhere: in your car, your kitchen, even your bedroom. I refer, of course, to computers, which are now so cheap that they are found in many very inexpensive devices, and so useful that it almost doesn't make sense to build anything complicated without one. I read some time ago that there were 14 microprocessors for each person in America, and that number is just trending one way. Get used to computers in everything that costs more than $20.

Those embedded computers need software, and much of this software is written on Linux systems. The propagation of Linux in the embedded market is inevitable and unstoppable, so this trend will only increase.

The Power of Free Software

The magic (or at least magical) power of free software is that it can adapt to any platform. Like The Blob, the eponymous star of the 1958 Steve McQueen film, it just absorbs everything that it comes up against. The reason is the very freedom that defines free software. Since no one owns it, no one has to pay anyone to use it. If you have a new hardware platform, you need to get some software on it, and Linux is free, portable, and the experience is already there—so why do anything else?

The software-development tools of Linux are developed, maintained and released as free software by the Free Software Foundation. In the next few weeks, I'll be looking more closely at these development tools, and I'll attempt to explain where they each fit in the Linux development package. Without these tools, there would be no Linux.

Developing Software for Linux Platforms

Fall is the time when high school seniors start to think about college choices. Some lucky few (like me), have to decide only which engineering college they will go to. Others, more perplexed, are trying to decide what profession, if any, they should pick. They are aware that their entire future hinges on this decision; in the casual way of 17-year olds, they don't let that bother them much.

At times like these, my wife frequently suggests information technology as a possible career choice. I am more hesitant to do so. I've discovered over the years that it's no favor to turn someone into a software developer. If it's not something they choose from the start, with their own eyes open, then should they really be doing this kind of work? Are good programmers born, or made?

I really don't know, but it's not my intention to do another "Sex in the City" parody (see "Linux and the City").

Judge a Craftsman by the Quality of His Tools

Fortunately, for aspiring programmers, the very best tools are also completely free. Linux provides a complete programming suite, from editor to run-time environment. The Linux tool chain is widely used throughout industry for the heavy lifting of software development, and a good programmer, trained in Linux tools, need never worry about running out of interesting and profitable employment.

To start, pick a text editor. My favorite choice is emacs (see "Emacs, the Only Program You Need"). Whatever text editor you pick, you will spend a lot of time there. So choose one you like, and spend a little time getting to know it. Any editor you choose should include the ability to compile, run and even debug code within the editor.

Compiled vs. Scripted Languages

There are many different programming languages, each with its own proponents. All those languages are available to Linux programmers, and most Linux programmers are skilled in a number of languages.

For the purposes of discussion, we'll divide languages into compiled languages and scripting languages. Compiled languages are the ones you use to make a user application, most of the time. Scripting languages are the languages you use to make some kind of "glue code," which runs various other programs in the right order, with the correct arguments.

That is, compiled languages usually are used to developed very focused programs that try to do one thing well. Scripting languages are used when you need to coordinate the efforts of a bunch of different compiled programs. (This is a loose distinction, with many exceptions and gray areas.)

Newcomers note: Scripting languages are easier to start with!

Start Easy with Python

I think the easiest language for the absolute beginner will be Python. It's nice, sensible and fun. In the '60s, the BASIC language was developed to teach people how to program, but that wouldn't have been necessary if Python had been invented.

Python has a simple and spare syntax, and its fully featured built-in library makes it a win for quickly written code that still makes sense to other programmers (With most scripting languages, you might as well try to read Martian as to read someone else's code.) Part of this is the unusual use of tabbing and white space in loops. Well-written code automatically looks good on the screen or the printout.

Although Python is a great learner's language, it also has many serious users and uses. It's easy to embed the Python interpreter within another application, so that very complex apps can use Python scripts. It's also used in some famous Web site hosts, such as You Tube.

Heavy-Duty Scripting with Perl

Although Python is loved by Web programmers, it is Perl that runs most of the back ends on the Web. Perl has been around a long time, and has gone through many significant changes in its history.

Perl looks a little more cryptic than Python, in part due to the Perl motto: There's more than one way to do it. Perl programmers can take many different approaches to the same goal, depending on how they feel that day. This kind of programming whimsy is what makes Perl fun, but also a little harder for newbies.

In both Python and Perl, object-oriented programming is possible. Object-oriented programming is just a way to organize really large programs (thousands of lines of code or more). Although it can be used on smaller projects, it doesn't really come into its own unless it has been used to organize a lot of code. In spite of the ample object-oriented features of Python and Perl, most serious object-oriented programming is done in Java.

Object-Oriented Compiled Programs with Java

Once the scripting world has been explored, it's time to move into the world of compiled code, with Java. (Many will argue that Java does not really compile code—it just pre-interprets some Java into pseudo-code, and then interprets the pseudo-code. I will not let such arcane objections stop me from classifying Java as a compiled language!)

Java is the most popular programming language out there. When I scan the job Web sites, Java is what I see. Also, it is the language of choice for computer science courses at universities around the world. Java is considered the serious contender for object-oriented programming in both academia and industry. It's not just on the desktop, either—many mobile-phone handsets and personal digital assistants are programmed in Java.

There are various flavors of Java out there, and most Linux boxes will already have Java. If you don't have it, get the Java installation download from Sun Microsystems, the inventor of Java.

C, the Granddaddy of Them All!

Only the most hard-core, the most adventurous and the most reckless programmers will start with learning C, the linga-franca of electronic computing. There is literally no platform that does not run C. And for many types of programming, include some hyper-cool stuff like robots and supercomputers, C is the language of choice.

The Linux tool for C is the GNU Compiler Collection, or GCC, and you can read more about it in a previous issue of ComputorEdge, in an article called "Development Tools: Say Hello to the GNU Compiler Collection."

Professional Software Development in Three Easy Steps

In summary, here are three easy steps to becoming a professional programmer:

1. Learn interpreted languages Python and Perl.
2. Learn compiled languages Java and C.
3. Obtain engineering or computer science degree from an accredited four-year university.

Easy, isn't it?

Building Software From Source

Mostly, when you are getting or upgrading software on your Linux computer, you should use whatever package-management system comes with your distribution. My distribution includes the Synaptic Manager (www.nongnu.org/synaptic).

Why Build From Source?

Sometimes you need to build from source. Not all software is built for your repositories, and sometimes, even with software available in the repository, you need a more recent version than the one available, maybe even a beta release of the next version, to get a feature or bug-fix that's important to you.

For example, on my system, I took a look at the tar command. As with all GNU utilities, running it with the --version option will tell you the version.

```
~/downloads> tar --version
tar (GNU tar) 1.16
```

Opening my Synaptic package manager, I see that my repository has a slightly later version, but still nowhere near the version 1.20 offered as tarballs on the gtar Web site (www.gnu.org/software/tar). I got one of the tarballs and used tar to unpack the distribution.

When you download a package from your package-management software, the executable files are downloaded to your machine. They were built by the package maintainer, who compiled the original program. When using a package-management system, nothing is compiled on your system. All the work is done somewhere else.

When you build from source, your computer has to do the heavy lifting. But the problems are not all on your side of the equation; the software developers face a serious problem too.

The Problem of Portability

One of the great things about Linux is its portability: It runs on every computer out there. Linux runs on supercomputers. It also runs on tiny embedded platforms; it runs on mobile phones, in laptops, and it is quickly ported to almost every new processor developed.

Back in the very early days of Unix, operating systems were written in assembly language and worked on one type of computer. An operating system was specific for a hardware platform, and that was that. When someone came up with a new hardware platform, they started again from scratch to develop the operating system.

In the early '70s, Unix programmers broke the paradigm by rewriting almost all of Unix in C, a higher-level programming language, and leaving only a core piece of assembly language programming. For the first time, an operating system could grow and improve as it moved from platform to platform, and developers could be confident of a complete, high-quality development environment without having to start from scratch each time. All they needed was a C compiler, and all of the tools that make up the operating system would become available. The rest is history.

But there are still differences from one machine to another. This is a concern when writing software for Linux (or other Unix) platforms. Since the operating system can run on such a wide range of platforms, getting software to work well can become a real challenge.

The problem comes from the dozens or hundreds of tiny differences between machines. One uses big-endian, the other little-endian storage. One uses 64-bits for addresses; another uses 32. The list is endless and endlessly frustrating. But when new hardware is being developed, newness is part of the deal. After all, if a processor didn't do something new, there wouldn't be much point to developing it in the first place.

How, then, can the poor software package manage to build itself on a new platform, when so many things can vary from platform to platform? The answer is the configure script.

Ask and You Shall Receive

The solution adopted by the Unix community is imperfect and not always very elegant, but it gets the job done. It is the configure script.

Each software package comes with a shell script called configure. Before building the source code, you run configure, and it asks your system an awful lot of questions. With the answers, the package will know how to build itself on your system.

After unpacking my tar installation, I was able to run the configure script. The output shows the questions that the script is asking the platform and the answers it is getting back.

```
~/downloads/tar-1.20> ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking how to create a ustar tar archive... gnutar
checking for gcc... gcc
```

```
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
checking for ranlib... ranlib
checking for bison... no
checking for byacc... no
checking whether gcc and cc understand -c and -o together... yes
checking how to run the C preprocessor... gcc -E
checking for grep that handles long lines and -e... /bin/grep
checking for egrep... /bin/grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
etc.
```

Making Sense of it All

The answers are not just written to the screen in the output shown above. They are also written to a header file that is available to all of the code in the package. This header file serves as a master list of information for the build to use in compiling the package on your platform.

The configure script itself is a giant Bourne shell script, which does not really bear looking at. (It's constructed by other tools, and to the natural ugliness of code that can only rely on the

lowest common denominator of the Bourne shell feature it adds the obfuscation of code constructed by a program instead of by a human.)

The configure script offers some standard options, and some that are unique to the package being built. The —help option to configure can tell you what is available. Some of the output for the configure for the tar package is shown below.

```
~/downloads/tar-1.20> ./configure -help

`configure' configures GNU tar 1.20 to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as
VAR=VALUE. See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:
  -h, --help              display this help and exit
      --help=short        display options specific to this package
      --help=recursive    display the short help of all the included
packages
  -V, --version           display version information and exit
  -q, --quiet, --silent   do not print `checking...' messages
      --cache-file=FILE   cache test results in FILE [disabled]
etc.
```

Another useful option is --prefix, which allows you to specify where the package should be installed. If the prefix option is unused, the software is installed under /usr/local.

Finish the Job

After the configure script has done its magic, it's possible to build the package with the make command. This will call all the tools needed to build the package, in the correct order, and with the correct options.

Those who like to live dangerously will just issue a make install command, and build and install the software in one step. The more cautious will do a make check install, to run any tests that came with the package. If the tests fail, the install doesn't happen.

The make command can do what it does only because the configure script has done much of the hard work in advance. Without configure scripts, Linux software could not be portable.

An Amazing Side Effect

Usually, when you hear about a side effect, you get worried. (Like if you're taking a medication and hear it has a side effect of EBD—Exploding Brain Disorder.) In the case of configure scripts, there is a side effect that you will be happy to know about.

By asking all those questions, the configure script is able to adapt to many different platforms, even ones that didn't exist when the package was released—even ones that the package developers never heard of. Each platform can have its own unique set of answers to the questions that configure asks, and the package can still cope with it.

It's this side effect that makes the configure script such a powerful tool for portability. It allows software to build on a new system, taking account of the unique differences of that particular platform. That's the power of configure.

Say Hello to the GNU Compiler Collection

Last week's Linux Link column described the important role of the GNU/Linux development tool chain. Without it, there would be no Linux. This week we zoom into the heart of software development.

The core component of the development tools is GCC, the GNU Compiler Collection (gcc.gnu.org). This formidable compiler package effortlessly handles C and C++ on almost every

known computer platform, but also provides FORTRAN, Pascal, Ada, Java, PL/1, Modula-2 and Modula-3, and more programming languages that you've never heard of.

The GNU Compiler Collection is not just a Linux thing; it's also the compiler for the Macintosh platforms, and is widely used for embedded development as well, including those fancy mobile devices. It can run on one computer and produce executables for another, different computer. This cross-compilation, as it is known, is what allows engineers to use personal computers to develop software for mobile phones, calculators and any other embedded application.

Strap in tight, and we'll hop in and give this tool a spin!

Getting It

Most Linux systems will already include GCC. Open your favorite shell and call GCC with the —version option to see whether it is there, and if it is, how old it is.

```
~> gcc --version
gcc (GCC) 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)
Copyright (C) 2006 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There
is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
```

If it's not on your system, try to use your package management system to get it.

As a last resort, you may have to build it from source code. If so, be warned that it takes a while, and is a little more complex than the average software built under Linux. (In particular, read the install documentation to tell the GCC to configure exactly which programming languages you want it to handle.)

The version of GCC is not that important for C programmers; C hasn't changed in a long time, and any version of GCC from 3.0 up works very well. With the other languages there is more change over time, and you want to make sure you have an adequate version of GCC for the language in which you want to program.

The Basics

The core functionality of GCC is to take the source code developed by the programmers, plus other library code, and produce an executable program—something you can run on the command line. As a sample program, I'm using a little prime number-finding algorithm, the rather famous Sieve of Eratosthenes. The code is appended below. Get your favorite text editor out and cut and paste the code. Save it as a file called find_primes.c.

Then turn this C source code into an executable with this command:

```
gcc find_primes.c
```

If this command succeeds, it will do so with no fanfare whatsoever. In fact, you might wonder if anything happened. That's just the laconic way of things in Linux. The above command will produce an executable called a.out. It can be run like this:

```
/home/user $ ./a.out

reached 10 primes.
2
3
5
7
11
13
17
19
23
29
```

If you change the numbers in the C code next to MAX_NUM and NUM_PRIMES, at the top of the file, you can change how many primes are found. On my little Asus EEE, Yum-Yum, it took 12 minutes and 11 seconds to compute the first hundred thousand prime numbers. (You must recompile the program with the GCC each time you change it.)

Command-Line Options

There are many command-line options with GCC, and the documentation in the info pages is excellent. Every GCC programmer should sit down at some point and just read the documentation from end to end.

The -o option allows you to name your binary executable something other than a.out. The -g option will include debugging information in the executable, so that a source code debugger can be used. The -Wall option tells GCC to give all sorts of interesting warning messages about your code. The -v option puts GCC into verbose mode—really helpful if you are having link problems.

Here's the same build, but with a more realistically complex command line:

```
gcc -Wall -g -o find_primes find_primes.c
```

The Three Steps of C Program Compilation

A C program is compiled in a three-step process.

First, the C pre-processor is run on the source code. The pre-processor is what takes the string NUM_PRIMES, and substitutes the number 10 wherever it is found. The pre-processor also handles the #include statements at the top of the file, by including the contents of the system headers stdio.h and stdlib.h.

Second, the C compiles the find_primes.c file into the find_primes.o object file. The object file is like a piece of an executable, but without enough to run by itself. In this case, there is only one source code file, find_primes.c. But in a real-world software project, there could be hundreds or thousands of code files, all of which have to be compiled separately, before being combined in the final step.

Third and finally, the linker is invoked. It takes all of the object files and links them together in a way that adds up to an executable program that the computer knows how to run.

Rebuilding the above program with the -v option, we can see a lot more information about what is happening. The output tells us the computer that is running the program, and the configure options that were used when GCC was built for this platform.

```
~> gcc -v -Wall -g -o find_primes find_primes.c


Using built-in specs.
Target: i486-linux-gnu
Configured with: ../src/configure -v
   --enable-languages=c,c++,fortran,objc,obj-c++,treelang
   --prefix=/usr --enable-shared --with-system-zlib
--libexecdir=/usr/lib
   --without-included-gettext
   --enable-threads=posix --enable-nls --program-suffix=-4.1
--enable-__cxa_atexit
   --enable-clocale=gnu --enable-libstdcxx-debug --enable-mpfr
--with-tune=i686
   --enable-checking=release i486-linux-gnu
Thread model: posix
gcc version 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)
 /usr/lib/gcc/i486-linux-gnu/4.1.2/cc1 -quiet -v find_primes.c -quiet
   -dumpbase find_primes.c
   -mtune=i686 -auxbase find_primes -g -Wall -version -o
/tmp/ccvZtEQN.s
ignoring nonexistent directory "/usr/local/include/i486-linux-gnu"
ignoring nonexistent directory
   "/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../i486-linux-
gnu/include"
ignoring nonexistent directory "/usr/include/i486-linux-gnu"
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /usr/lib/gcc/i486-linux-gnu/4.1.2/include
 /usr/include
End of search list.
GNU C version 4.1.2 20061115 (prerelease) (Debian 4.1.1-21) (i486-
linux-gnu)
```

```
     compiled by GNU C version 4.1.2 20061115 (prerelease) (Debian
4.1.1-21).
 GGC heuristics: --param ggc-min-expand=63 --param ggc-min-
heapsize=63541
 Compiler executable checksum: 60ce5af9fc7eaad04f44a3c8fd96f438
     as -V -Qy -o /tmp/cccwq8CL.o /tmp/ccvZtEQN.s
 GNU assembler version 2.17 (i486-linux-gnu) using BFD version 2.17
Debian GNU/Linux
  /usr/lib/gcc/i486-linux-gnu/4.1.2/collect2 --eh-frame-hdr -m
elf_i386 -dynamic-linker
    /lib/ld-linux.so.2
    -o find_primes /usr/lib/gcc/i486-linux-
gnu/4.1.2/../../../../lib/crt1.o
    /usr/lib/gcc/i486-linux
    -gnu/4.1.2/../../../../lib/crti.o /usr/lib/gcc/i486-linux-
gnu/4.1.2/crtbegin.o
    -L/usr/lib/gcc/i486-linux-gnu/4.1.2
    -L/usr/lib/gcc/i486-linux-gnu/4.1.2
    -L/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../lib -L/lib/../lib
    -L/usr/lib/../lib /tmp/cccwq8CL.o -lgcc --as-needed -lgcc_s --no-
as-needed -lc -lgcc
    --as-needed
    -lgcc_s --no-as-needed /usr/lib/gcc/i486-linux-gnu/4.1.2/crtend.o
    /usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../lib/crtn.o
```

Using a Debugger

Even in a small program like find_primes.c there are many mistakes that might be made. When programming, any mistakes are likely to result in that dreaded pest of the modern world: the software bug. Finding and squashing such bugs is part of the programmer's daily job. Figuring out what has gone wrong can be notoriously difficult, but there is a tool that helps: the debugger.

The GNU debugger is called GDB, and it can be run on our prime number program like this:

```
gdb ./find_primes
```

The debugger allows you to step through the program, one line at a time if you like. You can stop anywhere and look at any of the values in the program. The debugger deserves its own column some week, because it is so fun and useful that it really makes programming a lot easier.

Java Language Support: Why?

You might wonder why GNU provides support for Java. After all, Sun is already providing Java and all its associated technologies as freeware. But this was not always the case, and though they always gave away Java for free, the code itself was not released as freeware until just last year.

Now that the Sun Java code is all freeware, there's no reason not to just use that for Java development. However, the GCC Java compiler will create Java virtual machine code for embedded platforms, and that may be easier (and more free) than anything else readily available, on small processors that just can't run the Sun JVM.

Other Languages

To use the GNU Compiler Collection with other programming languages, you need to use the correct front-end program. For example, G++ is the C++ compiler. Under the covers it calls GCC with the correct options to convert C++ code into an executable. Other front ends include gfortran for FORTRAN, gpc for Pascal, gcj for Java, etc. To program in one of these languages, you write your code and call the appropriate front end, with the usual GCC options, and the executable will be built. (Unless you have a compile error!)

The Future of the GNU Compiler Collection

The GCC project is one of the most active, and most successful projects in free software history. Major releases of GCC come every few months, and the software is known to be stable, efficient, robust and reliable.

In classic free software form, the GNU Compiler Collection is organized to make the addition of new languages easy. One day, you may just want to invent a new computer programming language. When you do, GCC will be there for you!

Appendix: find_primes.c Source Code

```
/* Find prime numbers
   James Hartnett, 4/26/08
*/
#include<stdio.h>
#include <stdlib.h>
#define MAX_NUM 1000
#define NUM_PRIMES 10
int
main()
{
   int *prime;
   int i, j;
   int n = 1;
   if (!(prime = malloc(NUM_PRIMES * sizeof(int))))
   {
      printf("no memory!n");
      return 1;
   }
   prime[0]= 2;
   for (i = 3; i < MAX_NUM; i++)
   {
      for (j = 0; j < n; j++)
         if ((float)i/prime[j] == (int)i/prime[j])
            break;
      if (j == n)
      {
         prime[n++] = i;
         if (n == NUM_PRIMES)
         {
            printf("reached %d primes.n", n);
            break;
         }
```

```
        }
    }

    for (i = 0; i < n; i++)
        printf("%dn", prime[i]);
    free(prime);
    return 0;
}
```

POSIX: The Right Software for the Job—Everywhere

Failure is an orphan, and Linux has many, many parents, which may be one testimony to its success and the explanation for its rich genetic history as well.

Unix is not any one particular operating system, but rather a family of operating systems based on common ideas and tools, but with substantial variation from one system to another. This resulted in a rich set of features and a fertile environment for invention and development of new ideas. It also made porting software about as fun as taking a bucket of peanuts away from a herd of drunk elephants.

The Importance of Porting

Successful software inevitably invites porting; that is, taking the software from one machine and getting it to run on another. Take, for example, software from a research lab at a university. The program was developed over the years by grad students working 80 hours a week and sleeping in front of their computers at night. Perhaps the program predicts what will happen in electric circuits, or the flow of air around the wing of a supersonic jet that's being designed, or the flow of currents in the deep ocean. Whatever it does, it starts with some input data, processes it, and provides some new information to the scientist—something never known before, perhaps. This is such a powerful tool that there is no field of scientific endeavor that does not make extensive use of computer models.

The computer program is just a complex set of instructions. It is the formal expression of some subset of the domain knowledge of the programmers, married to the computational speed of the computer and all the other software on it. It is like a little piece of brains that can actually make useful predictions of this big mess we call reality. Even after the original programmers are long gone, this little part of their brain will continue to think on whatever machine on which they run.

Pretty amazing when you think of it that way, isn't it?

Out With the Old, In With the New

When the lab gets money to buy a new computer system, naturally they want to take this important piece of software with them. But there's a problem: The new computer is not the same as the old computer (and if it was, they wouldn't buy it anyway). New computers will be faster and more powerful than their predecessors, but they also might be from a different manufacturer, like IBM instead of HP, and run a different flavor of Unix.

What happens to the program? Generally the files are copied and recompiled, and frequently everything just works. It does so because of a standard called POSIX.

Beam Me Up, Scotty

POSIX is a bit like the universal translator in Star Trek. As Kirk explained, there are certain universal ideas and concepts that are found in all alien languages. With that little gadget Kirk, Spock and McCoy could talk to almost any alien, and with POSIX software can be written to run on almost every computer. POSIX, in fact, is that set of universal ideas and concepts that can be found on any operating system.

The Two Halves of POSIX

There are two pieces of POSIX: the API and the tools.

The API is an application programming interface, or really, a set of interfaces, to various services offered by the operating system. What this means in practice is that the builders of the computer system provide software on the system to translate what POSIX says and what the computer understands.

For example, the C function to open a file, fopen, looks the same to the programmer, no matter what kind of computer is involved. This is quite an achievement (though so commonplace we don't even notice it). Different computer systems can have radically different ways of storing data, yet the same fopen call will work everywhere. Every builder of computer operating systems takes the time to write the fopen function (and hundreds more like it) into their operating system. In the latest POSIX standard, there are more than 1,700 functions.

The POSIX tools are even more interesting. Rather than just the fragments of software offered by the system libraries, the tools are stand-alone applications that can be run from the command line. They are so useful that they are standardized by POSIX so that programmers can use them and be confident they will still work the same way when ported.

Some tools are provided as part of the various shells available. For example, the cp and mv commands, to copy and move files, are part of the command-line program, and are also specified by POSIX so that they will work the same from one system to another. Other tools are stand-alone programs, even programming languages like the awk tool, one of the many text-processing programming languages available with POSIX systems. With these tools you can write programs that will work the same way no matter what system they run on.

Getting POSIXed

Unless you are using a Windows machine, your machine is already POSIX or POSIX-like. Even Windows users need not despair; there are several ways to get POSIX tools to run under Windows, including Cygwin (http://www.cygwin.com), MinGW (http://www.mingw.org) and even one with some sort of upgrade package from Microsoft, though I have never used it. (As if I would trust any part of a critical application to the tender mercies of Bill Gates!)

POSIX can be experienced as a programmer or as a user of tools, and I enjoy using both. POSIX is another way in which software can spread from machine to machine. It's of critical importance to the world of free software that the software be as platform-independent as possible, to make it easy to use new advances in computer hardware and operating systems.

POSIX makes it possible to write software today that will work on the world's fastest computer in 2018 (which will certainly provide a POSIX layer). Writing POSIX-based software ensures that your work can live on long after your computer is gone and, if you're a good enough programmer, even after you are gone.

Make It Happen

In 1977 a fellow named Stuart Feldman at Bell Labs released a new computer program, destined for greatness. The name of his program tells you all you need to know about what it is for: He called it "make."

Putting Together Software

Early computers, like the kind available in the '60s and '70s, had very severe hardware limitations. There was so little available memory to work with—every kilobyte, nay, every byte, was carefully used. Programming was the task of managing functionality in such small units that they could fit into the hardware. Forget about things like word processing and spreadsheets. Think instead of giant, primitive calculators.

But by the late '70s, computers had changed. The release of the first microcomputers in the late '70s marked the beginning of a new hardware era—an era of plenty. The decade had seen the development of processors from primitive 4-bit designs, suitable for calculators, to the legendary Motorola 68000, a 32-bit chip later used in the first Apple Macintosh. Computers stopped being specialized calculators and became generalized information processors.

With the profound changes in hardware, changes in software development permitted a new generation of programmers to stop worrying about the hardware. With the new, very large-scale integration techniques for chip manufacturers, programmers didn't have to care about every byte and every clock cycle. They started to really stretch themselves and the new machines, and this led to more complex software. The software included such groundbreaking inventions as the spreadsheet, the word processor and the database.

These programs were much more complicated than previous software, with many more code files. Feeding every command to the compiler and linker became a significant task. Instead of the one-line GCC command that can be used to compile a single code file into an executable, the more complex program is built in a series of steps, which must be done in the correct order. Dozens or hundreds of commands need to be issued.

Keeping track of all this was a complex task for the poor, simple programmer. Furthermore, it was repetitive and boring. In other words, it was a task that called out for some software automation. And this is what Stuart Feldman achieved with the make utility.

Backward-Chaining Logic

The concept of make is very simple: You tell it what you want, and it figures out what it needs to do to get that.

Of course, it can't do this on its own—make needs the programmer to create a control file that describes what can be built (the targets) and how they should be built (the rules). The programmer can also specify dependencies between files, specifying which source files are used to create an executable. With all this information, make can decide on the minimal set of steps to get your executable built.

Since the operating systems keep track of the current time and the last modification time for all files, it's possible for make to decide which of your files should be recompiled (because you have changed the source code since the last compile), and which should be left alone.

The specification file is called a Makefile (note the upper-case M). Creating and maintaining them can be a real hassle, but not as much as building software without them.

An Example

The following Makefile builds a program called find_primes. The program is built from two different source code files, find_primes.c and calc.c. (This is the program from last week's column, but broken into two separate files to illustrate the use of make.)

This is a very simple example—a real project might have dozens or hundreds of source files.

```
# Makefile for find_primes example.
# James Hartnett 5/2/08

CFLAGS=-g -Wall

all: find_primes

find_primes: find_primes.o calc.o
    gcc -o $@ $^

find_primes.o: find_primes.c
    gcc $(CFLAGS) -c -o find_primes.o find_primes.c

calc.o: calc.c
    gcc $(CFLAGS) -c -o calc.o calc.c

clean:
    rm -f find_primes *.o
```

Lines that begin with the hash (#) are comments. The CFLAGS= line illustrates the use of environment variables in Makefiles. The rest of the file contains five targets, their dependencies, and the rules to make the target from the dependency.

The first target, all, is a special target—use this target to list all the things you would like this Makefile to make. In this case it lists find_primes as a dependency. Dependencies are sub-targets that must be built before the target is attempted to be built by executing the rule, if any.

The second target is find_primes, the executable we are interested in. It depends on the two object files, find_primes.o and calc.o. The rule for this target is the command using GCC to build find_primes from the object files.

The next two targets show how to build the object files find_primes.o and calc.o from the source files, find_primes.c and calc.c.

The final target, clean, has no dependencies. Its rule is an rm command, which removes the executable and also the object file. This commonly used target allows us to type make clean in order to delete all created files and start fresh.

With this Makefile, the developer can modify any of the source files, and then issue the "make all" command to have the executable built. Only those code files that have changed will be recompiled, and make will figure out all the steps needed to get your program built.

Built-in Rules, Special Symbols and Other Arcana

It all seems reasonably simple, but only because the example does not really represent how a Makefile looks in the wild. Note that certain strings are repeated in several places in our sample Makefile. For example, the string "find_primes" is repeated in numerous places in the file. Programmers hate that. Not only is it boring to read, but it increases the chance that the string might be mistyped in one place or another, leading to a bug. Far better to have each string in just one place.

Furthermore, make makes a lot of assumptions. And since make and the C programming language grew up together, the assumptions made by make are quite good when you are programming C code, as we are in this example. You don't have to tell make how to turn C

source code into object files. Make knows more about it than you do, and can be trusted to handle those details.

Make has some special symbols defined, which help with the repeating string problem. They make the Makefile a little hard to understand for the uninitiated, but that's the way the cookie crumbles over the keyboard of life.

Here's the same Makefile, taking advantage of some of the special symbols and other shortcuts offered by make. I suspect there are more I could take advantage of, if I spent a few hours more with the GNU make manual (www.gnu.org/software/make/manual/html_node/index.html).

```
# Makefile for find_primes example.
# James Hartnett 5/2/08

CFLAGS=-g -Wall

all: find_primes

find_primes: find_primes.o calc.o
    gcc -o $@ $^

clean:
    rm -f find_primes *.o
```

Brevity Is the Soul of Confusion

You may praise the terseness of Unix on many occasions, but none of them will be while debugging Makefile problems. The backward-chaining of the logic, the built-in rules, the use of special symbols, all cause make to be a real challenge. The verbose option (-v) can be used to resolve any question about what make is doing, but the output is so very verbose that following what is going on is still about as fun as picking diseased fleas off a cranky mountain lion.

But when it's all you have, it's all you have.

The Future of Make

Although make has enjoyed a spectacular success story, its shortcomings have become the stuff of legend. The complexity of make has contributed to an aura of black magic from the very beginning. Make has earned its place in the pantheon of software tools, but it has also earned a lot of curses over the years.

For very complex projects, the creation and maintenance of Makefiles becomes a full-time job. Makefiles that work on one machine usually will not work well on a different machine, and having portable Makefiles becomes a problem for teams that want to distribute their source code (as free software programmers tend to do). In the end, babysitting the build system becomes a significant task.

This, of course, leads to even further automation. In a future column, I'll go over the successor tool, automake, which builds portable Makefiles and has a much simpler syntax. However, automake does not replace make, it simply uses it in a portable way. And make is still just right for small projects with only a few code files. For all of these reasons, Stuart Feldman's tool lives on in the hearts and minds of Linux programmers.

Code File: find_primes.c

```
/* Find prime numbers
   James Hartnett, 5/2/08
*/
#include <stdio.h>
#include <stdlib.h>

#define MAX_NUM 1000
#define NUM_PRIMES 10

int calc(int max_num, int num_primes, int *n, int *prime);

int
main()
{
   int *prime;
```

```
   int i;
   int n = 1;

   if (!(prime = malloc(NUM_PRIMES * sizeof(int))))
   {
      printf("no memory!n");
      return 1;
   }

   if (calc(MAX_NUM, NUM_PRIMES, &n, prime))
      return 1;

   for (i = 0; i < n; i++)
      printf("%dn", prime[i]);

   free(prime);
   return 0;
}
```

Code File: calc.c

```
#include <stdio.h>
#include <stdlib.h>

int
calc(int max_num, int num_primes, int *n, int *prime)
{
   int i, j;

   prime[0]= 2;
   for (i = 3; i < max_num; i++)
   {
      for (j = 0; j < *n; j++)
```

```
            if ((float)i/prime[j] == (int)i/prime[j])
               break;
         if (j == *n)
         {
            prime[*n] = i;
        *n += 1;
            if (*n == num_primes)
            {
               printf("reached %d primes.n", *n);
               break;
            }
         }
      }
      return 0;
}
```

A Perl Beyond Price

It seems to take a free software product about 20 years to reach full maturity. Most free software projects are begun by one avid (and slightly crazy) programmer. But if the product meets a genuine need in the community, a free software project will begin to attract other programmers, first as users, then as collaborators.

Such is the case with a programming language named Perl. Started in 1996 by a programmer named Larry Wall, it was originally a program for helping to make sense out of the myriad text files that fill the life of a Unix system administrator.

From those humble beginnings, Perl has gone on to become one of the most popular languages in computer history, with hundreds of thousands of users all over the world.

Compiling vs. Interpreting

At heart, every program, in every programming language, is a simple text file (or set of text files). Into this text file the programmer pours his work, trying to get every variable name, loop index, and semi-colon correct.

Once the programmer thinks the program is ready to run, he faces the task of getting the computer to understand these text files and turn them into the kind of instructions that the computer can actually use.

In some languages, this step is done just once—the programmer feeds his text files into another program, the compiler, and some runnable, binary files are constructed that can be fed right into the computer to run—no further processing needed. This is the case for the compiled languages, such as C, C++, Fortran, and many others.

With other languages, such as csh, Perl and Java, the program is not compiled once; it is processed (or interpreted) each time you run it. Instead of having some binary files that can be understood by the computer, in interpreted languages the text files are reinterpreted each time.

Using interpreted languages seems less efficient, because the computer has to figure out your source code every single time you run the program, instead of just once. In the olden times, this was considered a serious drawback, but these days, with so many CPU cycles going to waste anyway, we really don't care about efficiency anymore.

Another disadvantage of the interpreted languages is that the source code is required to run the program. With a compiled program, you just need the binary executable files, but with an interpreted program, you need all the source code. This is a problem if you want to sell proprietary software, and for that reason commercial software is almost always written in a compiled language. Perhaps because of these limitations, interpreted languages seem to have gotten a lot less attention from language developers, and most of the interpreted languages in the late '90s suffered from poor syntax and poor resources for organizing large bodies of code—until Perl.

There's More than One Way to Do It in Perl

One of the biggest differences between Perl and other languages is the number of ways that a task can be accomplished. In most languages, the idea is that there should be just one correct way to do each programming task. If there's more than one way to get things done, the theory goes, programmer confusion will ensue, with some programmers doing things one way, and some the other way.

In Perl, this is turned on its head. Instead of there being just one way to do things, there are many. It is up to the programmer to choose which way is most convenient. It's true that this can easily result in poorly organized, confusing code. But it also allows the programmer to work very quickly, in the programming style that suits him or her best.

Object-Oriented Powerhouse

Perl functions very well as a procedural scripting language, like a sort of super-shell. It's just like programming in bash or csh, except everything is a little (or a lot) more convenient.

But Perl really begins to shine when its powerful, object-oriented features are used. With object-oriented programming, much larger programs can be written, without sacrificing organization of the code.

As with C++ (and unlike Java), Perl's object-oriented features do not intrude into the language. It's easy to write procedurally until you get a few pages of code, and then switch to object-oriented programming without throwing away all your existing code.

A Little Help from Your Friends

One of the advantages of object-oriented programming is that, if done correctly, it allows for the reuse of existing code. The idea is to gradually build up a library of objects that represent your problem space. In time, this growing body of code allows you to program more quickly.

In practice, re-usability is difficult to achieve in the real world. But in the Perl world, you don't have to try as hard. There exists the amazing code repository called CPAN, the Comprehensive_Perl_Archive_Network (www.cpan.org/). This archive of freely available Perl code provides code modules and scripts for almost every imaginable task, from statistics to neural nets.

Serious Perl projects start with a visit to CPAN to see what can be gleaned from this free software candy store, before spending time reinventing the wheel.

Learning Perl

Perl has the most extensive set of man pages of any Unix tool. Start with "man perlintro" to get started. The beginner's object-oriented tutorial is essential for those seeking to use the object-oriented features of Perl, but make sure you have a good understanding of procedural programming first.

A Little Code Sample

It's impossible to come up with a Perl example that is representative of Perl programming, because it can look so very different, depending on the task and the programmer. This chameleon-like language suites all programmers, because it allows every programmer to code in something approaching his or her "native tongue," whether that be C, Java, or BASIC.

Here's an example of some Perl code, to give an idea of what it can look like.

```
 use List::Util 'shuffle';
 foreach $n ("n", "av", "j", "v", "p") {
     open(FILE, "/home/ed/visionator/.vis_".$n) &&
                          (@{"$n"} = shuffle(split(",", <FILE>))) ||
die "no ".$n;
 }
 print "We will ".pop(@av)." and ".pop(@av)." ".pop(@v)." ".pop(@n)."
".
     "by ".pop(@p)." and utilizing our ".pop(@j).",
                         ".pop(@j)." and ".pop(@j)."
".pop(@n).".*bckslsh*n";
```

This is a complete Perl program, which, using some ASCII files, creates a vision statement for any organization.

```
 bash-3.2$ perl vis.pl
 We will methodically and cooperatively encourage
                       workshops by continuing evolution
                        and utilizing our flexible, organizational and
                        scalable challenges.
 bash-3.2$ perl vis.pl
 We will passively and proactively brainstorm action
                      items by establishing managerial policies
```

```
                    and utilizing our technical, diverse and narrow
foci.
 bash-3.2$ perl vis.pl
 We will cooperatively and reactively clarify metrics
                    by balancing supervisory roles and utilizing our
valid,
                    scalable and flexible strategies.
```

Although many organizations devote hours or days to the development of a vision statement, this program can generate thousands of visions an hour, saving staff time for more valuable pursuits, such as learning Perl!

Bash vs. Perl Cage Match

On a recent work project, I was dealing with about 500 lines of shell script. Now, 500 lines is not a large program. A usual Linux C program might have tens or even hundreds of thousands of lines of code. Microsoft Word has more than a million (but then, it's famously bloated).

So 500 lines should be easy to deal with. But shell scripting is just a pain, even in Bash, which is the highest development of the shell programming art. Bash contains all the best ideas from the original Bourne Shell (hence BASh: the Bourne Again Shell). I don't mean any criticism for Bash when I say that a 500-line Bash script can be a pain to deal with. Shell scripts are meant for simple tasks—the short and sweet sort of thing that can almost always be done in one line, if you can just figure out what that one line is.

But for complicated tasks, Bash scripting gets old. Programming Bash reminds me of the old definition of hardware/software: hardware is the part you can kick; software is the part you can only curse at.

Therefore, I started to switch these scripts to Perl. There are several important differences between Perl and Bash scripts, so obviously it's time for a Bash vs. Perl cage match. There's not much doubt which language will prevail—it's like watching Bourne the computer nerd fighting Perl the world kickboxing champion. Stay tuned for some old-fashioned butt-kicking.

Command-Line Options

Command-line options are those quirky little hyphens and letters you use on the command line. For example, ls is the command to list the files in the current directory, and the -l command-line option causes it to list more information about each file (the "l" stands for long).

Command-line options are very useful. They allow you to write general-purpose scripts, which can solve the same problem on numerous different data sets. They also allow the user to control exactly what your script does, and how it does it.

In Bash, command-line options are handled by the getopts function.

```
while getopts "va:h" opt
do
  case "$opt" in
    v) verbose=1;;
    a) args=${OPTARG};;
  esac
done
```

The equivalent Perl:

```
use Getopt::Long;
my %opt;
GetOptions(*bckslsh*%opt, 'verbose', 'args=s');
```

It's not much different for this tiny example, but the Perl GetOption call scales well—adding more options is easier. Also, the Perl version gives fancier options, with better error handling.

Some might claim this is an unfair comparison, because it uses an extra code module for Perl, but leaves Bash naked, without a helper function to bless himself with. But as the Getopt::Long module comes included with Perl, it's fair game in a language comparison.

This round ends with Perl ahead on points, and Bash still in there for the rest of the fight.

Using Other Tools

Something you always need to do is use the other tools on the machine. Bash is all about calling other utilities to get the job done. For example, to get the current working directory into a variable, you can call the pwd command, like this:

```
dir=`pwd`
```

With Perl, the equivalent is more clumsy:

```
$dir=`pwd`;
chomp($dir);
```

In this case, Bash is clearly better than Perl!

This round ends as Bash lives up to its name and nails Perl with a solid body blow.

Documentation

As with most mature Linux technologies, Bash is well documented. The man page for Bash (try "man Bash") is complete and well-written. There are also several good tutorials available on the Web (just Google "Bash tutorial").

But the documentation of Perl is simply outstanding. Not only do the Perl man pages comprise an entire book about Perl (start with "man perlintro"), the wealth of online information is unparalleled. I believe Perl is the best-documented computer programming language I have ever seen. Start with the CPAN site: www.cpan.org.

In the documentation round, Bash throws some good leather, but is completely overwhelmed by Perl's head-kicks and knee smashes. Looks like Bash is in real trouble out there!

Data

Back in the '80s, programmers developed a concept called "data-driven design." The idea was that your programs would be more flexible if they used some sort of data store to help control their behavior. That is, instead of having their behavior completely hard-coded, it depends on some of the input data as well.

For example, if an application allows you to set an environment variable to control its behavior or edit a data file before running it, that is data-driven design in action.

Data representation is crucial in programming languages. The more closely you match your data representation to your real problem, the simpler software development is.

Bash has a very simple data representation. It can handle scalars (that is, single values) and one-dimensional arrays (groups of values), and that's about it.

Perl has superb data structures. In addition to what Bash offers, Perl has lists, which are much like one-dimensional arrays in Bash. Perl also offers associative data arrays, which contain a set of key-value pairs. With associative arrays, very close mappings between the real-world data and the program data are possible.

Finally, Perl offers references, a rich pointer syntax that allows for complex, indirect data storage.

In this round, Bash is down on the mat bleeding. It has no hope against Perl's data body blows.

Object-Oriented Programming

The problem with programs is that they tend to get bigger and bigger, and more and more complicated, until no one can figure out how to change them without breaking them. To combat this problem, object-oriented programming was developed. It's really just a way of organizing your code so that you can find what you want, even in a million-line program.

Bash has no object-oriented programming constructs at all. They can be faked, but that's not the same as support within the language. Bash is like an old-style boxer facing a modern kick-boxer.

For a boxer, only the hands matter. A kick-boxer can punch just as hard—but can also kick 10 times harder. Ouch.

Perl has some great object-oriented moves, with the complete set of classes, inheritance and all the rest.

In this round, Bash is against the ropes, absorbing the punishment, and Perl delivers kick after kick to the head and body.

Calling the Fight

At this point the doctor is going to call the fight. Bash is just not in the same league as Perl, and perhaps it's a bit unfair to pit them against each other. But they are often used for the same sort of task, so perhaps this kind of bloodletting was inevitable.

The Software Mixed Martial Arts Commission might not allow these sorts of cage matches in the future, so perhaps it's almost time for the long-awaited C/Java bout.

Got the Time?

If you're sitting on a hot stove, according to Einstein, a minute would feel like an hour. If you're talking to a pretty girl, an hour can seem like a minute. That, the great Professor asserted, is relativity.

As Einstein knew, time is no laughing matter, and presents some interesting and subtle problems to the software engineer. Even a simple question can have a pretty complex answer: how long does a program take to run?

The motivation for this question is quite obvious. When building some engineered system, the computational components must keep pace with the overall system in the real world. In other words, we would like to be sure that the program that pops the chute out during a space shuttle landing takes less time to run than the space shuttle takes to careen off the end of the runway.

How Does the Computer Know?

Once you set the time on the computer, it keeps track of the passing of the days and years. A small battery on the motherboard allows it to keep time even with the power off. This is called the "real time clock," and it is read by the operating system when the computer is booted.

From that point on the real time clock is ignored, and the operating system keeps track of the time itself, in memory. As with any time piece, the basis of keep the time is a periodic event. For a pendulum clock it is the time the pendulum takes to swing back and forth; a quartz watch has it's little piece of quartz inside, pulsing at 60 times a second. Your motherboard has a little circuitry somewhere which fires off an interrupt to the main processor 250 times a second.

This unit of time is known as a "jiffy" and may vary from system to system, based on your motherboard, your kernel, and how the system is configured. The main processor counts in jiffies and updates a buffer which stores the number of seconds since a fixed date and time. In the case of Linux systems, that date and time is January 1, 1970; this date and time is known as "the epoch" for Unix systems.

The number of seconds since the epoch is stored as a 32-bit signed integer. Astute readers will wonder how long we can keep counting in seconds from 1970 before we fill up the 32-bit value. After all, 1970 was a long time ago!

Computers will run out of space to store more time on January 19, 2038. On that day, all over the world, millions (or billions?) of Linux computers will fail horribly, unless something is done to address the problem. One proposed solution is to double the storage space for time. Critics claim that this is just a short-term solution, since the same problem is going to occur again. Supporters point out that a 64-bit time would add 290 billion years, and perhaps by then we can figure something else out.

To find out the date and time, use the date command, which gives output like this:

```
<pre>
~> date
Fri Sep  5 06:14:10 MDT 2008
</pre>
```

The date command lets you control the format of the output, so that you can get the date and/or time in whatever way you want. To see the actual number of seconds since the epoch, use the + %s option, like this:

```
<pre>
```

```
~> date +%s
1220616811
</pre>
```

But How Long Did My Program Take?

So one way to time how long your program takes is to use the date command before and after, and subtract the difference. But this is unsatisfactory for two reasons: firstly because subtracting dates and time and showing the output in a meaningful way is a pain, and secondly because there is a better way, the time command.

The time command is there to help you answer the question: how long did my program take to run?

To use it, just run your program from the command line, and use the word "time" first, like this:

```
<pre>
~> time ./prime &> /dev/null

real    0m0.621s
user    0m0.410s
sys     0m0.000s

</pre>
```

(This is timing a program that calculates the first 10,000 prime numbers.)

The time command actually gives you three times, the real, the user, and the sys time.

The real time is what is also called "wall-clock time." This is the most obvious answer to the question of how long your program took to run. In this case, it took .621 seconds.

What's Going On Here?

But Linux is a multi-tasking operating system, and other things are happening at the same time that my program is running. For example, every time I type a key in the word processor, the computer puts aside everything else it is working on and processes my keystroke. Therefore a wall-clock time is not the best way to measure how long my program is taking.

The user and sys times address this problem. They tell how much time the processor spent processing user and system code, respectively. User code is the program your running (in this case the program is called prime.c), and system time is the time spent by the program calling system functions.

To put it another way: the operating system offers a bunch of services to the programmer, and these are known as "system calls." The sys time is the amount of time spent by the CPU handling these calls.

I this case, my program makes so few system calls that the time for sys comes out as zero. The time for user, .410 seconds, illustrates the difference between wall-clock time and program execution time. The wall-clock time is a third again as large as the time spent actually executing my program.

Using the time command you can answer questions about how long your program is taking to execute. You may not program the chutes for the space shuttle, but you can still find out how long your code takes!

Graphics, Sound and Games on Linux

How to Become a Linux Guru: The Graphics

Back when I was a lad, computer graphics consisted of what you could put together with the extended ASCII character set that came with the original IBM PC. That consisted of a bunch of little blocks and symbols that looked like something you would find on an Egyptian tomb.

IBM, in addition to all the numbers, punctuation marks and letters, added a bunch of funny hieroglyphics to its ASCII table. By printing these symbols in the correct places, you could create very primitive graphics on your ASCII terminal. Back in the day, this was considered pretty hot. I still remember how excited I was when I got bar graphs in my database applications using this technique.

Compared to the graphics we have now, it was all very primitive. These days, we can do a lot better.

The Usual Linux Problem: An Embarrassment of Riches

As usual with Linux, the biggest problem faced by the new user is the wealth of choices they are offered. In Windows, there is one graphics system—Windows—and everyone uses it.

In Linux, there are many different flavors of graphics, and you get to pick which one you like the best. The good news is that they all play together well, so no matter which one you pick, you can still use software written for other components and not even notice the difference.

The Windows Manager

In Linux, the graphics cake comes in two layers: the windows manager and the desktop.

The windows manager is what draws and maintains the windows on your screen, handles your mouse movements, maximizes and minimizes windows, etc. Different windows managers are available, and they draw windows that look slightly different.

In the early days of Linux, this was the layer of software that the user generally had to figure out how to install and operate.

But it's kind of a pain for the usual user to do much mucking around with the window-management layer, and it could take a bit of effort to get a set of software tools working well with your window manager.

What happened next was as inevitable as it was useful: Some groups of Linux users got together and decided to package all this up in a simple way, and thus was born the Linux desktop environment.

Keep It On the Desktop

The desktop environment software layer is all about aggregating the software in the windows manager layer, and adding some extra tools. These tools have become a standard part of the computer desktop metaphor. They include things like the useful Control Panel along the bottom

of the screen. It includes a file manager, which allows you to navigate around your hard drive, opening, moving, copying or deleting files and directories. The desktop environment also includes system administration tools to allow you to manage your system.

The desktop developers also ensure that a rich suite of software works with their desktop, for example a word processor, spreadsheet and presentation editor.

The idea of the desktop layer is that it gives you a basic set of useful software. The desktop is what users directly interact with, and it is here that budding Linux gurus will devote their attention.

The two main flavors of Linux desktop at the moment are called GNOME and KDE. They each have a slightly different look and feel, but they both will be familiar enough to any Windows user. Those who are using Linux boxes from some corporate vendors (like IBM and HP) will get the CDE desktop. Another choice for the slightly less conformist would be Xfce.

The best news is that all the different graphics' pieces play nicely together. It doesn't matter to the user whether a program was developed for KDE or GNOME. It will work on both.

The System Admin Reality

With Linux, there is a defined and public layer at which the desktop software works. Above that layer, the user sees dialog boxes and mouse cursors, and other aspects of the modern GUI. Below that layer, everything is text editing and command-line interface. The fancy system-admin tools and file-manager software are only putting a facade on this existing functionality.

As a result, there is always the possibility of bypassing the pretty graphic tools and doing your system administration on the command line. This helps, because if you are getting confused by all the tricky operations of your graphics tools, you can go back to the basics and figure out what is going on.

It also doesn't matter which set of graphical tools you use to manipulate the system. You can switch from GNOME to KDE tools on the same system, and they will inter-operate OK because they both are dealing with the same underlying layer.

Which One Is for You?

Which is the best desktop package? That's hard to say.

I have used both for years (on two different computers), and I can't really say that I prefer one to the other. For those who are very visually oriented there could be some preference, but I can go from one to the other without even noticing. In both cases, everything just works, and everything I expect to find on a computer is there. For me, it doesn't matter whether I use KDE or GNOME.

With Linux, you are offered a lot of choices, but the choice of desktop environment is pretty easy: Just take whatever one you have handy. If you don't like something about it, or just want to try something new, switch to a new desktop environment and give it a try.

Underneath it all, the same text-based code that has operated reliably for a decade or more will actually be running. The desktop is just a pretty set of dials on this mighty software engine.

X Marks the Spot

When you first look at a Linux system, you see some very familiar stuff. There's the desktop, with various objects on it. There's a taskbar, with some information and buttons, and there's the ubiquitous arrow of the mouse pointer—sitting there in the middle of the screen, but edited out of view by my well-trained brain, until I need to find it.

Something you might notice about the Linux desktop: It looks just like Windows!

This is not a coincidence, but neither is it a case of Linux copying the work of the microserfs of Redmond. The reason that the two graphical environments look the same is that they are both derived from a common source—the first graphical user interface work at Xerox, and the immediate derivative, the Apple Macintosh. It was at Xerox's Palo Alto Research Center that these concepts were developed, and they have come to fruition in several different ways since then.

The Beginnings of X

In 1983 at the MIT Laboratory for Computer Science, some professors were having a problem. They needed to provide a computing environment to the students, but they had a wide variety of computers, each with its own different way of doing things. They decided to create a graphical

system that would run on all these machines, but that would look the same on every machine. They called it X Window, or X.

In other words, they deliberately designed a cross-platform graphical user interface; there had been other graphical interfaces, but X was the first that was written for cross-platform use, right from the beginning.

This placed some extra demands on the programmers and the hardware, and was a little more work up front for everyone. But everyone agreed it was an elegant design.

The Windows Way

In the fast-paced and ruthlessly competitive personal computer industry of the '80s, elegant design was felt to be a luxury. In the Windows operating system, there is no separation between the program and the user interface. You run a program, like a spreadsheet, a window opens, and some code is running. The code is not separate from the window—it's all one package. Everything is jumbled into one lump of code and shipped.

Forget elegant.

The Ethernet and Other Crazy Ideas

The X Window project was also the first graphical user interface that took full advantage of the high-speed Ethernet network (also developed at Xerox PARC). With X Window, the user interface and the back-end program can be running on different machines. The user interface is exactly the same in both cases—windows with menus, the mouse, etc. But what I see on my screen is just the graphical interface; the core of the program can be running elsewhere over the network.

Client, Server (or Is That Server, Client?)

Sometimes X Window seems needlessly confusing. One common cause of confusion is the non-obvious use of the terms "client" and "server" in the X Window system. In X Window, the client is the back-end program, and the server is the X Window GUI program (i.e., the program that is actually drawing the screen for you and handling your mouse movements).

The important thing to remember is that X Window separates your back-end program from the program that is the user interface. If you open emacs (everyone's favorite text editor) on a Linux system, an emacs window will pop up, and it will look just like it does in Windows. But actually, two programs were started—one that is the core of emacs, and another that draws the emacs window and responds to your mouse commands, etc.

When you select something from the menu, or when something happens in emacs to change your window, the two programs communicate. If you select a command from the menu, the emacs window will send that to the emacs program; if that causes the window to change, then the core emacs program will send all those changes back to the X Window GUI program.

Since the two programs are running on the same computer, this looks just like what happens in Windows.

But if you have access to another Linux machine, use the ssh command to connect to it. (Make sure to use the -Y option). Start emacs on the remote machine.

Once again, an emacs window pops up. This window looks and feels like the local emacs window, but it is running on the remote machine. Your local machine is just acting as an X Window terminal to display the windows, handle the mouse commands, etc. When you use the keyboard or mouse to enter commands, the X Window program sends the information to the emacs running on the remote machine.

Why Windows Prospered

If the design of X Window is so danged elegant, how come everybody is running Windows?

Part of the reason is that the X Window designers didn't compromise their design, even though hardware was so marginal that getting X Window to run at all required a top-end workstation. These days we just fling around gigabytes of RAM as if it were nothing, but back when X Window came out, RAM was measured in megabytes, and it was far more expensive than it is now.

There is also, of course, the Microsoft marketing department—no slick corporate marketing department for X Window ever screened television ads saying that you should switch to X Window.

But X Window's design superiority just can't be denied. Perhaps this is the year that Linux starts to take over the desktop and, if so, it will be in part because of the amazing capabilities of X Window.

Sounds Bad for Linux

How nice it is when I get to write about some Linux feature that works better than the same feature on Windows!

Nothing is nicer to a Linux fan than discussing the merits of OpenOffice, or the joy of using gcc, the GNU compiler collection, to compile software. Let's talk about networking, or file systems, or editors, and I can drone on for hours.

In so many cases, Linux just works better.

Unfortunately, sound is not one of those cases.

A New Gadget

More than a year ago, my lovely and thoughtful wife gave me one of those newfangled MP3 players. If you haven't heard of these, you must spend all your time in a cave (as I do).

Oddly, perhaps, programmers like me are just not that good with gadgets. (Well, after spending eight or 10 hours trying to figure out how to get some complicated software working correctly, would you want to figure out a new gadget?)

My history of gadgets is one of unused devices, complex features unexplored, and VCRs with blinking "00:00" instead of the correct time. I've never managed to record a TV show by timer, or call my answering machine remotely, or hook up my TV to my stereo speakers.

This MP3 player sat on my desk for a year before I decided to give it a try.

Computers! Do They Ever Work Right?

The first step was to get an MP3 file and try to play it on my computer. Then, using my desktop Linux box, I could shop for music, store my collection, and then use the MP3 player to listen to it.

When I tried to play an MP3 file on my Linux box, I was shocked to find that it didn't work. There was an MP3 player, but when I tried to play an MP3 file, I got an error message telling me that the decoder was missing.

In fact, nothing I did could make it work. I could play CDs just fine, but not MP3s. I popped in a set of Mozart opera arias while I tried to get things working. While the music was fine, my system administration was not, and I got nowhere quickly.

(This is why I don't mess with gadgets in the first place!)

Fortunately, I have a good friend who has forgotten more about system administration than I will ever know. He came along and sat down at my system for a little while, humming his hum of concentration. I did what I am good at in such situations—I got a cup of coffee.

When I came back, a funky European dance beat was blaring through my speakers. (Along with Mozart, that's what I like to listen to.)

Obviously, my system admin friend had gotten the MP3s to play. When I asked him how, he started to list the drivers and packages that he had to install.

Yikes! This is exactly what new Linux users hate. With that other operating system, the one owned by a billionaire, everything like this just works. When I tried to play my MP3 file on a Windows box, the music just started without further ado.

The Ultimate Terror—Lawyers

The fault lies not in our stars, but in our lawyers. There are some licensing issues with MP3 that caused Fedora (my Linux distro) to drop MP3 support from its release.

It all comes down to software patents.

Software patents are the worst idea ever.

They are deceptions perpetrated on those who are ignorant of software. Patent lawyers don't know anything about software, so you can take any obvious software idea, phrase it in a new way, and file a patent for it, even though it is a well-known algorithm, or an obvious application of existing principles to a new area of engineering.

What do the poor patent clerks know? They leave it for the courts to figure out.

Whether or not lawyers can understand software is open to discussion, but everyone agrees that getting them to try is very, very expensive. No one wants to go to court because they violated a software patent, because who wants to spend more time talking about the law to a bunch of well-dressed people who charge $300 an hour?

This give patents some power, even if they are ridiculous.

I don't know if the MP3 patent is ridiculous or not. Since music is just a data file (as far as the computer is concerned), the compression technique is not terribly important. MP3 is an excellent technique, but there are others, and some may be just as good or even better.

It comes down to marketing. Everyone knows about MP3. Everyone's players and mobile phones handle MP3. So, everyone wants MP3, and the MP3 patent holders can make money by selling the right to use their software idea.

All of this has scared Fedora away from including MP3 support with its distribution, and who can blame the company? Software engineers are more frightened of lawyers than lawyers are of software engineering. After all, engineering has to make sense, whereas that is not at all required in the law.

A Principled Stand for Free Software

You have to admire the principled stand of Fedora and other free software distributors who respect the MP3 patent. They have taken the stand that they will release only code with which you, the user, have complete freedom. (Free as in "freedom of speech," not as in "free beer.")

However this puts the user at some inconvenience when a popular commercial format is not supported. I think, over time, free software will evolve a better format, based on some of the existing free formats. The most obvious candidate is Ogg Vorbis format. Whether, or when, this will be adopted by the marketplace is less obvious.

Where to Go to Get MP3 on Linux

If you don't hanker to learn all about installing packages on Linux, and don't have a handy friend who can help, perhaps you can try following the instructions on the unofficial Fedora frequently asked questions list (www.fedorafaq.org/#mp3).

Now that I can get my MP3s to play on my computer, the next step is to get them on and off my actual MP3 player!

Maybe next year.

Captain Kirk Walks Among Us

A co-worker of mine asserts that everyone falls into one of three personality types: Captain Kirk, Mr. Spock, or Dr. McCoy.

Dr. McCoys are interested in helping everybody, Mr. Spocks are driven to understand everything, and Captain Kirks are driven to drive hard for personal glory.

According to my co-worker's theory, only the Kirks among us are interested in computer games. Those who want to help people and those who yearn to understand the universe don't waste their time on such frivolity. ("Dammit, Jim, I'm a doctor, not a gamer!")

Only action-driven glory hounds can sit in front of a computer monitor for eight hours of constant virtual action. People who enjoy computer games enough to spend this kind of time on them clearly have some strange personality problems.

I say charge up the photon torpedoes, and let's head for the neutral zone! Because, while the names and the faces may change, we all know that the basis of most computer games is destruction, and new aliens are just targets we haven't met yet. Things aren't any different in the Linux world.

No Gravity

No Gravity, a freeware game, will make your inner Captain Kirk happy. A space shoot-'em-up in the classic mold, it allows you to pilot your little fighter from system to system, blasting up other spaceships along the way. As addictive as it is easy to use, it can suck away a Saturday morning faster than any other Linux game I've tried.

While learning this game, you'll destroy dozens of enemy fighter ships—and a few on your own side—before you figure out how to tell friend from foe on the radar screen.

The graphics are excellent, and gameplay with the mouse is outstanding. In no time, you will be using the mouse to navigate a ship of variable acceleration through a moving system of friendly ships, enemy fighters, mines, navigation beacons, and asteroids. Best of all, if anything gets in your way, relief is just a mouse click away, firing an array of interesting-looking pulses at whatever your space fighter is aimed at. A few well-placed shots will make enemy fighters flame, smoke, and eventually explode in a most satisfying manner.

The game itself consists of a sequence of missions to complete in order to move on to the next level of play. Eventually, I'll get to the end of one of the scenarios and find out what happens when you win. At the moment I'm much more familiar with explosions, flames, and spinning out of control, which are all associated with being destroyed.

Go to http://www.realtech-vr.com/nogravity to download the game (for Linux, Mac or Windows) and the game data file. (You need both the game and the data file.)

Freeciv

As a turn-based strategy game, Freeciv doesn't try to compete with the adrenaline-releasing shooter games. Its challenge is more like that of a game of chess, but with hundreds of different types of pieces, and a game board as big as the Earth itself.

Freeciv gives your creative side full play, as you carefully design your civilization, gathering resources, establishing cities, and developing an infrastructure. Finally, when your internal civilization is perfect, it's time to go destroy everyone else in the game!

In Freeciv, you start out with a few settlers in an undiscovered world of forests, mountains, oceans, rivers, and other terrain features. Settlers can found cities, and cities can grow to produce more settlers, as well as military units of various types, ranging from spear phalanx to jet fighter.

Freeciv is based on the Civilization games, most notably on Civ 2. But Freeciv goes far beyond Civ 2, with more detailed graphics and a better user interface.

My favorite unit in Freeciv is the nuclear missile, which can destroy an enemy city with a fun little mushroom cloud. But usually these are defeated by SDI defenses—unless you've sent a spy in to sabotage them first.

Rome wasn't built in a day, and your empire won't be either. I've had Freeciv games that I've played over weeks, with more than 24 hours accumulated. And many is the civilization of mine that has gone down in a blaze of fire, with the quick movement of enemy units along my railroad lines—once they break into the heart of your empire, there's nothing to stop them!

For all things Freeciv, see the Web site athttp://www.freeciv.org. Windows users can also get together on the Net and play each other.

McCoys and Spocks

It's true that the world of computing wouldn't be what it is today without the McCoys and the Spocks, trying to help and trying to understand.

The Kirks have their share of contributions, too, and it should be noted that games have always been one of the driving forces behind hardware and software development.

So arm the torpedoes and activate the warp drive—it's time to blast some aliens.

Office Tools

Getting Out the Word

Back in the old days, it used to be important to know how to spell.

How well I recall hearing from my teachers how I would suffer for my lack of interest in spelling. "When you grow up, James, you will need to write letters, and your poor spelling will be a real handicap."

I ignored all the good advice. And I turned out to be right, and they turned out to be wrong. They never would have believed me if I had told them that not only would I manage to succeed

without good spelling, but that I could actually earn money with my writing. It was the invention of the word processing program that put all those warnings in the trash bin of life, and that allows me to survive and thrive, although I still can't spell for crap. As a result, I have always felt sorry for those who could spell a word only one way.

Early Word Processing

In the early days of computers, there was no word processing—there was just text editing. The difference is that word processing allows you to do more than just manipulate a file of letters strung out in order. It also allows you to do fancy things like titles, bullet points, superscripts and other enhancements, which turn a plain series of letters into a formatted document.

Way back in the early '80s, the way to do real word processing was with a very expensive typesetting machine. These were about the size of a large desk and had a keyboard, primitive monitor and a large printer. To get your document, you would write it up by hand (or, if you did it a lot, on a typewriter), and turn it over to the typesetting specialist, who would re-type it into the typesetting machine, along with the special codes that were needed to control formatting, titles, etc. The typesetter would then print out a copy for you to check—called the galley proofs. You would read and correct these and send them back to the typesetter, who would enter your changes and run off another set of galley proofs for you. Eventually you would be satisfied with the results (or just too fed up to continue), and a final copy of the document would be printed for you.

Does all that sound like a huge pain? Because it certainly was. Some of the problems with this primitive word processing included the fact that the machine and its operator were very expensive, and also usually busy, so turnaround was slow. Preparing a document could take a week, even after all the text was written. Furthermore, the whole procedure was possible only if you were lucky enough to work at a place that could afford one.

The Early Word Processors

The early word processors would be called text editors today because they didn't even try to show you what you were getting. You might make some text bold-faced, and it would come out of the printer that way, but on the screen it looked the same as all the other text. The "What You See Is What You Get," or WYSIWYG (pronounced wis-eee-wig) was not available until better graphical monitors and operating systems were developed in the 1980s.

WYSIWYG is a great idea, and a key feature of what we now recognize as a word processor. The first WYSIWYG editor was the (still) famous Microsoft Word, at that time a newcomer to the word processing market, trying to displace the much more popular WordStar program.

They Grow Up So Fast

It didn't take long for WYSIWYG word processors to catch on, and new features started appearing as fast as dollars in the Gates checking account. Spelling and grammar checking, integration with spreadsheets, databases, diagramming software, more fonts and text-justification options all became standard parts of the word processing program.

Eventually, as with most computer programs, the word processor developed ways to handle repetitive tasks with a simple form of internal programming, first with macros, then with dedicated scripting languages. As the concept of the word processor developed, so did the various software packages that met these needs. As invariably happens, the word processor has become a commodity product. There are many available, for a variety of prices, but they all do more or less the same thing. In the corporate world, Microsoft Word reigned.

Getting Away From Microsoft

Word has been a real moneymaker for Bill Gates, and still is. In fact, it used to cost so much money that it was cheaper in 1999 for workstation developer and manufacturer Sun Microsystems to buy a whole company than to buy a copy of Microsoft Word for each of its employees. The company was a German software outfit that had developed a knock-off of Microsoft Word, which was part of their suite of software called StarOffice. Sun turned around and immediately released the source into the free software community, and OpenOffice (www.openoffice.org) was born. There's still a commercial version available from Sun for corporate users, which includes some extra tools and features, but the free Open Office suite already has everything I need.

Running With the Pack

These days, the OpenOffice word processor, called Writer, is almost indistinguishable from Microsoft Word (except that it's free). It offers the full gamut of WYSIWYG features, and is what I am using to write this article. It can even read and write documents in Microsoft formats so that you can use OpenOffice Writer interchangeably with Microsoft Word. The latest releases of Writer even support the Visual Basic for Applications scripting language. This means even complex Word documents with significant scripting can be handled by OpenOffice.

It's Everywhere You Need To Be

Another great feature of the OpenOffice word processor is its ubiquity. It's available (always free) on just about every computing platform out there: Linux, Macintosh and even Windows. It works the same on all platforms, so that you don't even notice what kind of machine you are using.

Some Cool Stuff

One of the reasons I so enjoy writing about Linux is that it allows me to learn so many interesting things about software that I work with almost every day. Like most word processor users, I use a small subset of all the fancy bells and whistles that come with the software. Naturally, OpenOffice Writer offers automatic spell checking and a thesaurus (a.k.a. synonym finder, wordbook, in case you were wondering). It has a presumptuous (but often annoyingly accurate) auto-correct feature. It has a built-in media player, so that you can embed media into your documents. It can handle tables, frames, outlines and bullets in a million different ways. It can keep track of bibliographies and mailing lists. And all that without the free extension packs you can find on the Web!

Breaking a Bad Habit

When I show Windows users the software available on Linux, I am often asked, in amazement, "If all this free software is just as good, then why does anyone pay Microsoft?" I usually point out that the free software is frequently much better in quality and usefulness than proprietary software, and that lots of people don't pay a penny to Microsoft. But I still end up getting suspicious looks. I get the feeling that they think I am trying to pull something on them. After all, would all those Microsoft users really be paying all that cash if they didn't have to?

People get into bad habits, and it can be hard to change. Maybe I should market some kind of skin patch for people trying to give up Microsoft. If you prepare any documents on your computer, then check out OpenOffice Writer and start to kick the habit.

Getting Down to Business

The entire personal computer market was built on the spreadsheet.

Back when the original IBM PC was released, the big question was: Why would anyone want a personal computer?

It sounds like a crazy question now, but other than the hardcore geek market, what reason was there for building a tiny computer for just one person? The only people who could use a computer for anything useful were the computer geeks who worked on the Big Iron, the mainframe computer. They despised the emerging world of the personal computer. The early personal computer market was for hobbyists, not serious users.

Then a program called VisiCalc was released, and the spreadsheet was born. Businesses suddenly discovered the power of small computers, and the rest is history.

One of the Great Inventions of the Renaissance

Back in the last millennium, people like Galileo and Thomas Jefferson were wandering around, causing important dates to occur so that future schoolchildren would have something to memorize. Businesses were doing something called double-entry bookkeeping by hand to keep track of their money.

Here's how it worked: You kept a big book, called a journal, which had a bunch of specially lined sheets of paper. There were horizontal lines, each representing one expense. There were also a bunch of vertical columns, about 10 or 12. There was one for rent, one for supplies, one for payroll, and others for whatever else the common expenses of the business were (including the catch-all "misc" column).

Every time any money was spent, the amount was recorded twice, once next to the comment about the expense and once in the appropriate column for that expense—hence the term "double-entry."

When the journal page was full, each column was totaled, and the totals for all categories of expenses had to match the total expenses for that page. If the two numbers were not the same, then you made an arithmetic mistake, and you had to keep redoing your math until you found it.

The double-entry bookkeeping system, developed in Renaissance Italy, is what allowed modern business to develop and flourish. It was so successful that it was used by every business in the Western world, right up until the invention of the spreadsheet.

In fact, before I discovered how much easier it is to earn cash by computer programming, I worked as a bookkeeper for a bakery in Philadelphia. Each week I would go in and pay the bills, writing checks to the suppliers of flour, fruit and sugar. I would pay the rent, the insurance and the payroll.

Each number was duly recorded in the journal in two places. Each filled page was cross-totaled, and the numbers checked again and again (and sometimes, again and again) before everything added up correctly and the owner knew exactly what was spent. But what a pain!

Even though I was being paid a pretty good hourly wage for the bookkeeping, I always groaned when the cross-totals didn't match. It meant at least a half hour of repeating all the calculations on that page until the mistake was found. It was tedious, it was painstaking, and it was not nearly as fun as hanging out with my buddies and girlfriends—not even as fun as doing my usual engineering homework. It was drudgery, pure and simple. And the more you hated math, the worse it was. Some business owners hated math so much that they hired people who liked it just to do the bookkeeping. Other business owners just suffered. Even those who enjoyed math (as I do) found bookkeeping to be boring.

Computers to the Rescue

There are lots of things computers are not good at. They can't hold a conversation, cook a good meal, or appreciate a great work of art. But computers are good at adding. And they are good at drudgery. They never get bored and make a mistake. All of these things make them great bookkeepers. But getting the computer to do what you want is not always easy for the non-specialist.

With the invention of the spreadsheet, a tool was created that didn't require special engineering talent. Any business owner can immediately understand it, since it is so similar to the double-entry bookkeeping journal. The spreadsheet makes the bookkeeping business obsolete. With a spreadsheet, you don't even need double-entries. Just enter it once, and trust the computer to do the addition correctly.

(But don't worry about the poor bookkeeper losing his job. You don't have to be smarter than a computer to survive, just smarter than the person who owns the computer. And every business owner who doesn't have to pay a bookkeeper still has to pay for someone to operate the computer! But they pay less, and it gets done quicker, and so everyone is happier.)

Linux Left Out?

In the early days there was VisiCalc, but it was a program called Lotus 123 that created the massive spreadsheet market, which still thrives to this day. Eventually, like a bad rash, Microsoft spread over everything, and now Excel rules the spreadsheet roost.

In the early days of Linux spreadsheets were lacking, and it was one of the few areas in which Windows users could claim superiority. But, as it always does in the end, free software caught up with multiple spreadsheet projects.

Gnumeric vs. Open Office

These days, there are two very good Linux spreadsheet programs: Open Office Calc (www.openoffice.org) and Gnumeric (www.gnome.org/projects/gnumeric).

Open Office is a complete knock-off of Microsoft Office. It includes a word processor, a presentation manager, a database front end and a spreadsheet.

The spreadsheet is a clone of Excel, and only the most advanced users would be able to spot a difference between them. Although the Open Office applications have their own formats for their files, they can also use the Microsoft formats, making them drop-in replacements for the Microsoft applications. Someone using Open Office can create and edit files in Microsoft formats, and Microsoft-using friends and co-workers need never even know that you are not sending money to Bill Gates to do your spreadsheets.

Open Office Calc Spreadsheet

Gnumeric is an older and more stand-alone application. Not part of any office suite, it nonetheless completely replaces Excel on the user's desktop.

Which One to Use?

Once again, the theme for the new Linux user is one of choice. Unlike other, proprietary operating systems, Linux offers more than one spreadsheet application. So which one is a new user to choose?

As a somewhat casual spreadsheet user, I can only say that I have never found anything that either program could not do just as well as MS Excel. Both are perfectly adequate replacements —both can open, edit and create spreadsheets in MS Excel format. Both can handle graphs and charts, statistical analysis and complex extensions and customizations. Both offer excellent compatibility with Excel spreadsheets.

Use whichever one you find on your Linux machine. Whichever you choose will work just fine!

The Adventure Continues

Some time around 2010, ComputorEdge stopped their print edition, and became an on-line magazine only (though still with local editions, including one for Colorado). Due to financial limitations Jack was unable to continue to pay for these articles, and so I stopped writing them. In the years I had been writing them I learned a lot, grew a lot as a programmer and perhaps as a writer, and had a lot of fun. I miss writing them!

But the computer industry continues to grow and change. New technologies come on to the already crowded landscape with exciting and ever-increasing frequency. Computer software is changing our lives more and more as each day passes. There is always more to learn, and more to do.