Linux Journal Issue #9/January 1995

Linux Journal
New Year's Resolutions
In This Issue:
Virginia Power Goes Online with Linux
Linus Predicts the Future
Hooking Up To the Internet
Kernel Korner: Block Device Drivers
Donate that old 386SX to the kid next door who wants to try Linux
Delete the DOS partition from my hard drive
Give my old Linux CD to a friend
Send a virtual beer to Linus
Subscribe to Linux Journal!!

Archive Index

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

A Conversation with Linus Torvalds

Belinda Frazier

Issue #9, January 1995

Our associate publisher talks with Linus via e-mail to get and update on Linux projects.

Linux Journal: You recently went on an international tour, speaking in Belgium, Australia, Singapore, and other places. Could you describe some of the questions or events you found interesting?

Linus Torvalds: Hmm. I got a few interesting questions. In Australia, for example, I had two quite separate persons ask me whether the windows emulator would be extended to run OS/2 programs as well. I couldn't answer them (although I think it's very unlikely to be a high priority), but I found the fact that somebody even asked interesting, as I haven't seen an OS/2 program.

Anyway, the most interesting parts of Australia weren't computers at all, but the small and furry (and sometimes feathered) animals there. I got bitten by a penguin in Canberra (Killer Penguins Strike Again), but it was a very small and timid one. And I naturally saw all the normal Australian animals like wallabies, koalas, etc.

LJ: Were you in a zoo or on the coast when you were bitten?

Linus: It was at a zoo in Canberra. The wild fairy penguins seem to be much too shy to approach at all closely. I don't remember what the island with all the penguins close to Melbourne was called (might have been Shark Island), but reportedly people going there just get to see a lot of penguins; the penguins are so shy that you won't get very close.

LJ: Where did you find the best beer?

[Editor's note: This question about beer needs an explanation for those new to Linux. Linus thanked the “Oxford Beer Trolls” for sending him some virtual beer in his release notes. Also, available on the Internet is a photo of Linus with a beer bottle in front of him; the photo is captioned “Linus Torvalds—Creator of Linux”. Jokes about virtual beer and virtual breweries have blossomed among Linux users.]

Linus: The Australian beer was okay, although I happen to prefer Guinness, not lagers. There was one interesting stout in Singapore called ABC stout (or something equally exciting), but I still think I should probably go to Ireland some day.

LJ: Did you hear any good jokes you could share with us?

Linus: I heard one ... but I don't think that one is suitable for a family magazine like *LJ*.

LJ: How has your perception of the Linux user base changed?

Linus: I don't think my perceptions have changed all that much. The user base is much more “user” these days and less “hacker”, but that's not some revelation that I got during my trips abroad.

Some of them make a mean barbecue, and some of them say “G'day mate”.

LJ: The last time we talked via e-mail was last January and much has happened during these past eight months. What Linux projects are you working on right now?

Linus: Uhh.. Getting ready for 1.2, I guess. It's already late, but I'd like to have that over and done with. Various problems there, of course—mainly the floppy driver and the TCP problem with the new ciscos.

And the alpha port. Watch this space, but don't hold your breath or you'll go blue and mottled in the face.

LJ: Those of your projections fulfilled or close to fulfillment include: i386 SYSV binary compatibility and windows emulation (halfway there we're told). What projects or work related to Unix, if any, have you been surprised to find not yet fulfilled or close to fruition?

Linus: Me? Surprised about projects not fulfilled? You must be joking. I'm more surprised about the various things that have been fulfilled (the Linux system itself being one of the things I'm surprised by).

Of course, there are a few projects that haven't come to anything yet, but for which I didn't really have high hopes (but I'd be more than happy to be proven wrong). Like a nice WYSIWYG word processor (yes, I use LaTeX occasionally, but no, I'm not crazy enough to think it's the answer).

LJ: We heard you were doing a 64-bit port for the Alpha. How is this progressing?

Linus: Right now I just have a bootloader and am testing the Alpha console code (essentially the same as the BIOS on the PC compatibles, but much more complex and not as well documented). So I have a simple program which boots the system and explores what's going on (the Alpha is a fun chip, I can tell you). The port by Jim Paradis is much further along, and even gets you a shell prompt (but not much else). I'll certainly leverage on that, but the travels have been limiting my time in front of the computer.

LJ: I heard there were two efforts going on for Linux being ported to the PowerPC and the Mac, and that one effort was put on hold because of lack of information from Apple. Do you think the effort is stalled, or do you know if there is still real progress being made?

Linus: I have no idea on the PowerPC port. I have only seen the occasional reports (the latest one indeed saying that they had no knowledge about the IO interfaces). Apple isn't known for disclosing technical information and IBM doesn't seem to have any PowerPC machine out yet (except for the RT which doesn't follow PReP). I don't know what will happen with the PowerPC (with regard to Linux or anything else for that matter). I saw a report about IBM now also considering the Pentium again.

LJ: What is PReP or a PReP machine?

Linus: PReP stands for “PowerPC Reference Platform”--essentially a unified external interface to the PowerPC chip, defining the external bus and the BIOS interface. It's an IBM standard, but even IBM doesn't have any machines out there that follow that standard yet. IBM does have machines with the PowerPC chipset, but those are in their RT line of Unix computers, and have their own bus architecture around the chip (essentially the same one that the POWER series of processors had which were the predecessors of the PowerPC chip).

LJ: Because you know the kernel better than anyone else, how do you feel about the port? Do you think it will be an easy port? Do you think it will run as well on a PowerPC as on the Intel architecture?

Linus: Oh, the PowerPC chip itself shouldn't be the problem. The memory management of the chip is rather strange (and ugly, imho), but that can be considered an extended TLB and the PowerPC port could well use the same memory management architectures, etc., as the current i386 version. The port should obviously run quite quickly on the chip.

The surrounding hardware (and thus the device drivers) will prove to be more problematic unless something comes up (e.g., IBM finally releases a PReP machine and actually gives enough technical documentation on it).

LJ: What is TLB?

Linus: TLB: Translation Lookaside Buffer. It's essentially a small cache inside the processor that caches the page tables, so that the processor doesn't need to look up the virtual-physical mapping in the page tables each time it does a memory access.

The i386 has a TLB with 18 entries (don't quote me on that, but it's something of that order), that it uses to cache the 2-level page tables that define the virtual-memory layout. When a TLB cache miss occurs, the i386 will then (in hardware) look up the virtual mapping in the page tables, and fill in the TLB.

The PowerPC uses a slightly different approach—it won't do a page table lookup when it misses its TLB. Instead, it will look up a new TLB entry from a hash table that has been filled in by the operating system. The operating system can use whatever page table it wants to generate that hash table.

As a final example, let's take the Alpha: it has only a TLB and does any TLB miss lookup in software (the PAL-code). So you can chose your own way of implementing the page tables. (You could do a hash-table plus a physical page table like the PowerPC, or you could go to the page tables directly, like the i386.)

LJ: When do you think we will see Linux on PowerPCs?

Linus: I'll pass on that one. I think both the Alpha and the MIPS ports will be there before the PowerPC, if only because the hardware and the documentation already exist.

LJ: What are the new features you see as needed for Linux?

Linus: The main new feature needed by the average user would probably be the ability to run windows binaries; I hope the Wine project really works out. From a kernel view, the memory management needs some tuning, and the buffer cache needs to be reorganized to allow indexing by inode and offset instead of the current device- centered view. And threading is already something of an issue.

LJ: “What's In a Name?” Do you think Linux would have taken off as fast if you'd named the operating system what you first considered, Freakix? Do you think someone would have published Freakix Journal?

Linus: Actually, it was just “Freax”. And I think Linux turned out to be a much better name, even though I at first thought it would sound too egoistical.

LJ: Anything else you'd like to say?

Linus: So this is where you expect me to do all the interesting revelations, is it? Foiled again.

LJ: Do you have any new hopes for Linux?

Linus: I think my “plan” says something like “World domination. Fast.” But we'll see.

LJ: Thanks for the interview, Linus. We appreciate your taking the time to answer our questions.

Live Chat

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Connecting Your Linux Box to the Internet

Russell Ochocki

Issue #9, January 1995

Want a faster connection to the Internet? In this article, you'll learn how to get your Linux box connected.

Ever wonder what it would be like to have your Linux box connected to the backbone of the Internet? Well, maybe not the backbone—but how about a direct connection to one of the many Internet service providers? Lightning-fast response time. You could set up your very own ftp site, or perhaps a gopher or World Wide Web server. Wouldn't it be nice to have Mosaic draw heavily graphical World Wide Web pages in seconds instead of minutes at 14.4 kbps?

What does all this cost? You'd better sit down first. Comfy? Good. Assuming you already have a Linux box, startup costs are in the range of $3,000-$5,000. Monthly costs range anywhere from $500-$1,000. This generally places such a connection out of reach for most people.

However, there is currently a rapidly-expanding market for small Internet service providers, who need a faster connection to the Internet than 14.4 kbps to be competitive—and some of whom are using Linux. There are also more and more businesses becoming interested in having a presence on the Internet, and many are using Linux boxes to provide this.

In this article, I'll take you on a tour of what's required to get a direct connection to the Internet. First, I'll touch on the various software packages you'll need to understand how to manage an Internet-connected machine. Next, I'll introduce the concepts behind a direct Internet connection; I'll describe the hardware required and the different configurations available. Finally, I'll discuss how to select an Internet service provider.

Get Some Experience First

You've got a lot to learn. No, seriously. And, with the cost of a direct Internet connection, the last place you want to learn this stuff is online. If you have the time to do it right, I'd suggest connecting in several stages, learning in pieces as you go along.

Let's assume you are fairly skilled at managing a Linux box. This is a skill you can learn even if your machine has no connections to the outside world. In fact, having no connections makes learning easier; it will allow you to focus on the tasks needed to maintain a Linux box. Once you are comfortable doing this, take one small step towards the Internet: Obtain a UUCP connection to your machine.

A UUCP connection is a good way to get your feet wet connecting to the outside world—it will teach you how to manage a news and mail feed. Most of the Linux distributions come with all the news and mail tools you'll need. The Mail-HOWTO guide and the News-HOWTO guide will help you configure things correctly. News and mail are the two most common services on both directly connected and indirectly connected Internet machines. These are important services that people using your machine will expect to work. Spend the time to learn how each package works. Make sure you understand how news and mail are configured. Learn what log files are produced and where they are located. When you connect your machine to the Internet, it is inevitable that you will encounter problems with news and mail. You can save yourself a lot of time and trouble by learning how these work now, rather than later.

The next step above a UUCP connection is a dial-up IP connection. Dial-up IP places your machine on the Internet like a dedicated connection, whenever you dial in. This will give you some experience running TCP/IP. You can also try running your very own ftp site, a gopher server, or even an HTTP server for WWW clients. If you expect to be running any of these services when you get your dedicated connection, start experimenting with them now. Learn how to configure them. Learn what log files are produced and where to find them. Check out the comp.info-systems.www newsgroup.

Security becomes a very important issue once you place your machine on the Internet. Any data residing on an Internet-connected machine can potentially be read by anyone on the Internet, unless your security prevents it. Now, even if you think you don't have anything on your machine that you don't mind others reading, don't think you can just brush security concerns aside. There are many documented bugs in Unix packages that allow hackers to gain access to an existing account, or even root access.

To get up to speed on security, start by reading the comp.security .unix newsgroup. It has an excellent FAQ on what to watch out for. Also check the newsgroup comp.os.linux.announce. You will find Linux-specific security holes posted here. The best method to determine just how secure your machine is, is to have someone try and break in. If you know someone who is very knowledgeable about Unix, that person would be an excellent candidate for the job. If you don't know or trust someone enough to do this, just get a few average computer people to try to break in. You'd be surprised at how many holes even the average user can discover.

What Type of Dedicated Connection Should I Get?

At this point, if you have followed my advice, you've managed a UUCP news and mail feed. You've worked with dial-up IP. Maybe you've even tried running a gopher, ftp, or HTTP server. And, you have learned a lot about Unix security. If you haven't, do it now!

A dedicated connection means your machine is connected to the Internet 24 hours a day. This speeds up services like news and mail. Mail between two Internet-connected machines happens literally in seconds. The frequency of Usenet news updates is controlled by each site. Hourly—or even more frequent—news updates are commonplace. You also get some services that are only available to Internet connected machines such as telnet, ftp, gopher, and World Wide Web.

In order to put your machine on the Internet, you will need a dedicated line between you and your service provider. A dedicated line is a telephone line that is open 24 hours a day. What do I mean by open 24 hours a day? Say you call a friend and talk for a few minutes. Then, you walk away from the phone for a while. When you have something else to tell your friend, you pick up the phone and tell him. You don't have to dial his number again because you've never hung up. This service is billed at a fixed, monthly rate; there is no charge for usage. The phone company connects the dedicated line to the destination phone number. Only the phone company may change the destination.

You will have to decide how fast a connection you will need. The minimum speed is 56 kbps, which is perfect for a small business. If you plan on transferring audio in real-time, you will need a 1.54 Mbps line, commonly known as a T1 line. If you plan to transfer video in real-time, you'll need a T3 line which transfers data at the rate of 45 Mbps. Watch out for bottlenecks—buying a T1 line in the hopes of talking with a remote site across the country at T1 speeds is pointless if any of the other lines the data will pass through are running at 56 kbps.

Dedicated lines come in several different flavours. Analog lines can handle speeds up to 28.8 kbps. This is the same grade as your typical home phone line. You probably don't want one of these. Digital lines handle speeds of 56 kbps right up to T3 (45 Mbps) speeds. The cost of a digital line depends on the distance between you and your service provider. An alternative to digital dedicated lines is frame relay. Frame relay is the new technology on the block. Frame relay charges are based on speed, not distance; this may offer significant savings over a digital line. Not all service providers support frame relay. Check with your service provider. For the purposes of this article, I will assume you are going to go with a digital line at 56 kbps. This is the most common Internet connection.

With a dedicated connection, your Linux box is available 24 hours a day to access the Internet. But beware, the reverse is also true. The Internet can access your Linux box 24 hours a day. Keep your machine secure or you could suffer a lot of damage from system crackers. In order to prevent this, consider reading Cheswick and Bellovin's Firewalls and Internet Security, reviewed in issue 6 of *Linux Journal*.

The Hardware

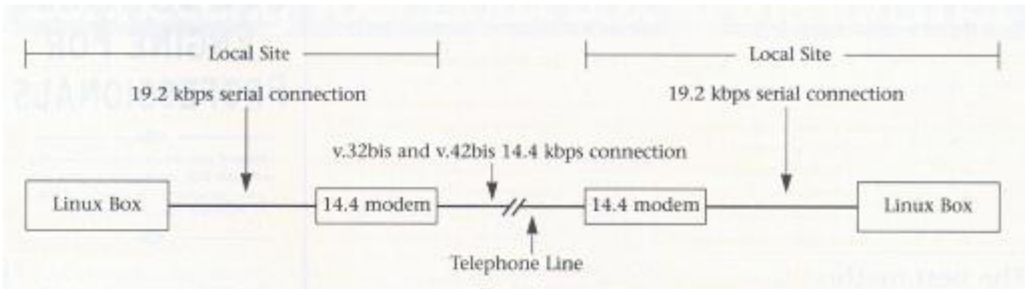
Local Site
19.2 kbps serial connection
Local Site
19.2 kbps serial connection
v.32bis and v.42bis 14.4 kbps connection
Linux Box
14.4 modem
14.4 modem
Linux Box
Telephone Line

Figure 1
14.4 kbps Connection

Before I describe a 56 kbps connection, let's review a connection with which you are probably more familiar: a regular 14.4 kbps modem connection. (See Figure 1 above.) A 14.4 kbps connection will require a serial port in each machine, a modem at each machine and, of course, a telephone line. The two modems communicate at 14.4 kbps using the v.32bis protocol. The serial connection between each modem and the Linux box can be set at 19.2, 38.4, or 57.8 kbps; data compression is the reason the serial connection runs faster than the modem. The modem connection is 14.4 kbps compressed with the v. 42bis compression protocol; the serial connection is uncompressed. In order for the serial line to keep up with the modem connection, it must pass more bits per second than the modem. Now that you know where all the protocols fit into the picture in a 14.4 kbps connection, let's tackle a 56 kbps connection.

Take a look at Figure 2 (opposite). A 56 kbps connection may be too fast for your serial port, so Ethernet offers an alternative. Ethernet signals cannot be transferred over the telephone lines, so you must use a protocol specifically designed for telephone lines, v.35. What you end up with is Ethernet coming out of the Linux box, being converted to v.35 signals, and being transferred over the telephone lines to your Internet service provider. You need to install an Ethernet card in your Linux box and configure the kernel to support TCP/IP —see the NET-2-HOWTO document for the details. To convert Ethernet signals to v.35 signals you will need a router. Finally, to send the v.35 signals over the phone lines, you will need a 56 kbps CSU/DSU (also known as a digital modem).

The router with CSU/DSU is the most common configuration for dedicated connections to the Internet. Vendors are now selling hardware which combines the router and CSU/DSU into one box. The single box is cheaper, but not as flexible in case of future growth. For example, if you want to change from a 56 kbps to a 128 kbps line, you can use the same router with a 128 bkps CSU/DSU. If you go with the single box, you'll have to replace the entire unit. Take into account your plans for the future and pick the option that suits them.

It will soon also be possible to buy a v.35 CSU/DSU card that plugs directly into your Linux box. That is, it is possible to buy the card now, but the driver is still being developed as this is written. When the driver is available, this option will cost less than an Ethernet card, router, and external CSU/DSU, be a little less flexible, require that the Linux box it is attached to act as a router, and be ideal for many situations where the Linux box is being used as a firewall. On the other hand, it is a poor solution for sites with more than a few dial-in lines.

Choosing an Internet Service Provider

The Internet service provider business is booming right now. New companies come online each month. Service providers come in all sizes—from large, cross-country providers serving an entire country, to medium-sized, regional providers serving several nearby cities, to small providers serving only a single city.

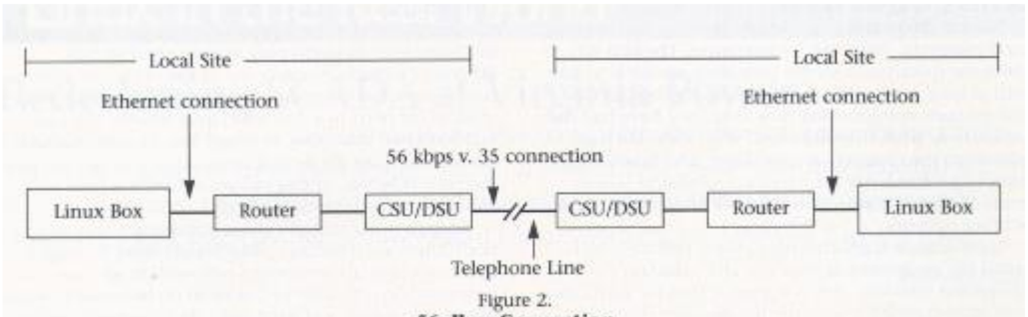

Figure 2.
56 dbps Connection

Finding the larger providers and most medium-sized ones is easy. There are several lists of service providers available; here are a couple I have found useful:

- DLIST: A list of Internet service providers that sell direct connections. Most, if not all, of the large national providers are listed here. You can get a copy of this list by sending e-mail with an empty message body to dlist@ora.com. If you have any problems getting this list, send e-mail to mj@ora.com.
- PDIAL: A list of Internet service providers that offer dial-up accounts. The PDIAL list contains many more providers than the DLIST. The additional providers tend to be the small to medium-sized ones. Most of the companies listed will provide dial-up accounts only; however, if you find a provider close to you, call or e-mail them and ask if they sell dedicated connections—some of them might. To receive the PDIAL list, send e-mail to info-deli-server@netcom.com with "Send PDIAL" in the body of the note.

Finding the smaller providers that service only your city can be a little trickier. Check your city's computer paper. Check local user groups. Check with local computer stores to see if they know of anyone providing Internet access in the area. You'd be surprised how many consulting firms, out-sourcing companies, and even computer stores are using the unused time on their computer equipment to provide Internet access.

Most providers have an e-mail address you can send mail to for more information; select a few and e-mail them. Describe your site to them and the type of connection you are looking for. For example, if you have an office LAN you want to connect to the Internet, tell them. See what they suggest and how much it will cost.

Now, you have to be careful when it comes to costs. Some providers charge a setup fee. Others require you to sign a six- or twelve-month agreement with them. Some charge only for the Internet feed; you must pay the phone company directly for the dedicated line. Others combine the two charges and you pay the service provider only. Some require you to purchase a CSU/DSU at their site. Others include this charge as part of your setup fee. In the end, you'll have to decide on whether you want the flexibility of a month-to-month lease, or the extra savings of a long-term commitment.

Narrow the possible candidates down to two or three good prospects. Then, ask for references. The best way to check the quality of a service provider's service is to talk with at least three other companies using their service and get their opinions. Ask how long they have had the connection, what they like most, what they like least, how often the connection goes down, and how long it takes to get fixed. Ask for references similar to you in terms of type of connection, number of users, and type of office network.

Maintenance is another issue. Some providers will install the equipment at your site and maintain the equipment remotely; this is a good option for small sites with little experience with the hardware involved. For the more daring, you can install your own equipment at your site. You might have the odd interruption as you learn how things work, but if you don't mind this possibility, the knowledge you gain will be helpful to you later.

You still need to purchase a dedicated line from your site to your service provider's site. Some service providers take care of this for you; others require you to arrange this with the telephone company. Arranging for a dedicated line is not difficult—you could do it yourself. The advantage of having the service provider handle this is you only have one number to call if something goes wrong with the connection. If you are dealing with two companies, each may tell you the problem lies with the other's equipment and will ask you to contact the other first. If you have ever run into this vendor-roulette before, you'll know how frustrating it can be. Neither side wishes to investigate the problem until the other side has investigated things first.

What To Do with Your Connection?

If you have an existing TCP/IP network that you are connecting to the Internet, you may want to set up your Linux box as a firewall between your network and the Internet. This is done to prevent unauthorized users from accessing services you want only your local users to access. See the comp.security.unix FAQ for more information about firewalls. If you plan on connecting dial-in lines to your Linux box, I have some suggestions on the machine size and configuration. These suggestions may not work for every situation, but they will give you a starting point from which to work.

For a small operation with 1-4 dial-in modems, a 486DX33 with 16MB of RAM and a four port serial card would be a good starter system. For a medium sized operation with 4-16 dial-in modems, a 486DX33 with 32MB of RAM and a 16 port serial card might be reasonable configuration to start with. Keep in mind that the CPU speed and amount of memory needed will ultimately depend on what your users will be doing on your system. If they only read news and send e-mail, this might be more than enough. If things start slowing down, add more memory.

For a large operation with 16+ dial-in modems, try two 486DX66s with 16MB each. Put a large hard drive on one machine and NFS mount it on the other machine. With so many modems, you don't want to overburden your Linux box with serial ports. Instead, you can get a terminal server which is a piece of hardware that manages modems. Your modems plug into one end and an Ethernet connection comes out of the other. Another feature of terminal servers is they allow you to attach each modem (or port) to a different Linux box. In this case you have two machines, so assign half to each machine. If your machines are getting overloaded, you can increase memory as before, or you can get a third machine and redistribute the ports accordingly.

Keep in mind that the above systems are only suggestions; there are too many variables involved to suggest that any of the above systems will work in all cases. Your configuration will ultimately depend on how many people use your machine and what tasks they normally perform.

Looking Further

If you want more information about connecting to the Internet, I recommend the following books:

- *Connecting To the Internet*, O'Reilly & Associates, ISBN 1-56592-061-9. Covers all aspects of connecting and offers a general overview of how data travels through the Internet, the different types of available hardware, how to choose an Internet service provider, and the trade-offs of dial-up IP vs. 56 kbps connections.
- *Canadian Internet Handbook 1994 Edition*, Prentice Hall Canada, ISBN 0-13-304395-9. If you live in Canada, this is an excellent source of information on how the Internet flows through Canada; includes a list of service providers by province.

Various USENET newsgroups are also an excellent source of information. Check out the following:

- comp.security.unix - Unix security issues.
- comp.unix.admin - Administering Unix boxes in general.
- comp.os.linux.announce - Important announcements about Linux.
- comp.os.linux.admin - Administering a Linux box.

If you have a security question, ask it first in comp .security.unix. For the most part, security is the same on all flavours of Unix—it is rarely Linux-specific. If your question happens to be one of those rare cases, there are many Linux-literate readers of this newsgroup who can help you out.

Connecting your machine or network to the Internet is a huge undertaking. But if you take the time to learn how things work, you will be able to tackle this task with ease. Good luck on your connection adventure.

Russell Ochocki, B.C.Sc. (Hons), is a computer programer/analyst for a major Canadian financial corporation. He has been using Linux for over one and a half years. He can be reached on the Internet at rdo@kynes.bison.mb.ca.

Russell Ochocki, (rdo@kynes.bison.mb.ca) B.C.Sc. (Hons), is a computer programer/analyst for a major Canadian financial corporation. He has been using Linux for over one and a half years.

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Remote Network Commands

Jens Hartmann

Issue #9, January 1995

Jens Hartman shows how to use rlogin, rcp, and rsh to transfer and manipulate data on different computers from across the network.

Normally, being connected to some kind of network does not mean that you are able to directly access all resources provided by that network. Some devices, like tape drives and printers, are connected to special computers and are only accessible on these machines. Others, such as disk drives, can be accessed easily, when the system administrators allow it.

Gaining direct access to resources becomes more complicated the larger the network gets. This is partly due to security reasons, and partly due to the simple fact that the more people that have to be convinced you need to mount a disk, the less your chance of success. From this follows the rule that the more resources available, the harder it will usually be to connect to them.

A nice and easy way around this dilemma is using the remote commands **rlogin**, **rcp** and **rsh**. These commands allow access to any account that is owned by you on any computer in the network without the use of a password. **rcp** and **rlogin** can be compared with ftp and telnet, whereas **rsh** offers the possibility to combine commands on different machines in one shell pipeline.

Configuration is extremely easy—in fact, your host network is probably configured correctly already—and you have instant access to these capabilities. In this article, I present these commands, explain the local setup, give some examples to give you a start, show some options, and demonstrate the complexity that can be reached. Most of my examples can easily be replicated on your networked machine.

Setup:

Under normal conditions (your network is up, you can telnet into other machines, and you can be reached by other machines), the only thing you need to do is create a file called **.rhosts** in your home directory which is readable and writable only by you (mode 600). This file should contain the full hostnames of each of the machines you want to log in from, and the user name on that machine, like this:

```
apple.groucho.edu           fred
orange.groucho.edu               sam
```

The **.rhosts** file specifies the machines and users that are allowed to login to the user **on the machine where the .rhosts file resides**. If I am logged in as sam on the machine banana.groucho.edu and I have the above .rhosts file in my home directory, then the user sam from orange.groucho.edu and the user fred from the machine apple.groucho.edu have remote access to my account.

Now, I log into apple.groucho.edu (username fred) from my account on banana.groucho.edu. From apple.groucho.edu I run the following command: **rlogin banana.groucho.edu -l sam**. Once you are logged in, shell commands will work as normal.

If you are asked to enter your password, do not enter a password, but instead quickly switch back to your original login on banana.groucho.edu and type **ps -a**. In the process list your rlogin request should appear with the name of the machine it came from as an argument. When this is different from the name you entered in the **.rhosts** file, you will need to enter the new name. Sometimes a machine uses different lines or a common server for such communication, although its name doesn't change. If there is still no connection, you should ask the system administrator. Some machines simply don't allow any rlogin commands.

In order to respond to any rlogin request, your Linux machine's **inetd.conf** should have the following two lines:

```
shell   stream  tcp     nowait  root /usr/sbin/tcpd  /usr/sbin/in.rshd
login   stream  tcp     nowait  root /usr/sbin/tcpd  /usr/sbin/in.rlogind
```

When you are a member of a domain and share usernames, you might want to include the hosts you frequently connect to in **/etc/hosts.equiv**. In this case your **.rhosts** file may contain only the nickname (which is commonly just the machine name without the domain information) together with the username. The above example **.rhosts** file on the machine named banana.groucho.edu would then look like:

```
apple.groucho.edu                fred
orange                           sam
```

RCP:

The rcp command copies files or directories from one machine to another. It is used like the cp command. For instance, I can copy a file named test.dat from the remote machine banana to my local machine orange. (For this example to work the two machines must share usernames.)

```
rcp banana:test.dat .
```

or

```
rcp banana.groucho.edu:test.dat .
```

The file **test.dat** is situated in my home directory on banana and is copied to my current directory on orange.

If I want to copy my Mail directory and its contents to orange into the directory **Mail.banana** (again, from orange):

```
rcp -r banana:Mail Mail.banana
```

To preserve the time stamp I would type:

```
rcp -r -p banana:Mail Mail.banana
```

Making a remote copy from the machine apple where I have a different account, apple:

```
rcp fred@apple:test.dat test
```

Of course, things also work the other way around. Here is a remote copy to apple:

```
rcp test.dat fred@apple:test.dat
```

The last interesting thing would be a copy from apple to banana, while you're logged into banana. Unfortunately, this works on every other machine except my Linux machines:

```
rcp fred@apple:test.dat banana:test.dat
```

You see that **rcp** is a bit handier than ftp.

```
rlogin:
```

With **rlogin** you perform a remote login to another machine. It can be used instead of telnet:

```
rlogin orange
```

or

```
rlogin -l fred apple
```

or

```
rlogin  apple -l fred
```

(for some versions of Unix)

I integrate every machine in my window-menu with an rlogin. This makes login very efficient. As an example, here are two descriptions—one for windows manager **fvwm** and one for **olvwm**--to add a menu and a shortcut key for **rlogin** to orange. The **xhost** can be omitted, but it is useful for other things. For **fvwm**:

```
Popup "Rlogin"
Title "Rlogin"
Exec "banana F1" xhost +banana;\
exec xterm -fn fixed -T banana -sb -e rlogin banana & EndPopup
Key F1  A N  Exec "banana" xhost +banana;\
exec xterm -fn fixed -T banana -sb -e rlogin banana &
```

For **olvwm**:

```
"Login" MENU
"Rlogin" TITLE PIN
"banana"  xhost +banana; exec xterm -T banana\
-sb -e rlogin banana
"Login" END
```

rsh:

The **rsh** command is the most powerful remote command, as it can be integrated into a pipe. This enables you to execute complex command sequences between different machines. Without any command, **rsh** rlogins to the other machine. When I am logged into orange:

```
rsh banana
logs me remotely to banana, while:
rsh -l fred apple
```

does the same on apple, where the username is different from the one for my current shell on orange.

Both **stdout** and **stderr** from the remote machine are piped to the local machine. After establishing the connection, neither **/etc/profile** nor **.bash_profile** (for bash) nor **.login** (for csh and tcsh) are scanned. This can be confusing in the beginning, as not everything you defined as variables and aliases for a login shell are present. This is different from rlogin, which gives you a real “login shell”. The most common problem is that you use a command that cannot be found because the path variable has not been set correctly. In these cases, you could set your path in your shell initialization file, such as **.bashrc** or **.cshrc** (there is no such file for **/bin/sh**, however). A more general solution is to simply use fully qualified pathnames for commands.

The easiest use of **rsh** is a simple command like:

```
rsh banana ls
```

You should get a listing from your home directory on banana. If you want to use options that start with “-”, the syntax would be:

```
rsh banana `ls -al'
```

Everything in between the quotes is executed on banana.

The listing was sent to your local stdout, your screen, just as if ls had been executed locally (on orange). Below are some examples of creating a tar archive on banana from a directory called bin, with the output going to stdout and the archive file being placed on orange:

```
rsh banana `tar cf - bin' | dd of=archive.tar
```

or

```
rsh banana `tar cf - bin' > archive.tar
```

Of course this will work for different usernames or in the other direction too:

```
rsh -l fred apple `tar cf - bin' > archive.tar       tar cf - bin | rsh -l
fred apple dd of=archive.tar
```

The **dd** command is very handy here, because it copies stdin to the file specified by **of**. If I were to use > to redirect the output, I would end up on orange again, but I want the file to be written to apple instead, which **dd** does correctly.

rsh Wizardry:

Now for some examples of what happens on which machine. Start with something simple:

```
        ps -aux | grep root
```

This shows all processes root owns on your local machine. The next command will show all processes on banana which are owned by root.

```
          rsh banana `ps -aux' | grep root
```

In this case, the **ps** command is executed on banana, but **grep** is executed locally (on orange).

The next command does the same thing, except that the grep command is executed on banana, as is the ps command.

```
  rsh banana `ps -aux | grep root'
```

or

```
  rsh banana `ps -aux' | rsh banana `grep root'
```

The next example shows how to split **stderr** and **stdout** (this works only for sh or bash, not for csh or tcsh!):

```
   rsh banana `dd if=bin.tar 2>fehler' > test.dat
```

Here we have dd that writes its error output to the file **fehler** on banana, but which transmits its standard output to orange into the file **test.dat**. The secret here is the use of quotes. Because **2>fehler** is inside the quotes it is executed on banana. Things can get very tricky. Not only can you make full use of shell commands, you also can run them on different machines:

```
  rsh banana `tar xf bin.tar `rsh banana `tar tf \ bin.tar' | grep gj2.c`'
```

Here I have a tar archive **bin.tar** on banana. In it there is a file called **usr/local/src/gj2.c**. First, there is a command expansion in between **` ... `**. This expansion returns all the filenames from the archive which contain the string “gj2.c”. First, **tar tf** returns the list of files in the archive and then **grep** (running on orange) performs the pattern match. This yields **usr/local/src/gj2.c** and **usr/local/src/gj2.c~**. Now the first tar (**tar xf bin.tar**) extracts these files on banana.

Imagine that you have a tape drive attached to banana and another to orange. You want to make a copy of a tape. Dumping the contents of the tape to the disk drive and copying it back to another tape would be a solution, but would require enough free disk space to hold the entire contents of the tape. Another solution would look like:

```
  rsh banana dd if=/dev/rst0 ibs=1024 | dd \
  of=/dev/rmt0 obs=1024
```

Here **/dev/rst0** is a SCSI tape drive on banana and **/dev/rmt0**a SCSI tape drive on orange. Now you want to process your data with a special program called **demux**. After processing, your data has shrunk considerably to about 200 megabytes. As we operate on binary data, porting our program to banana would be very time consuming (orange is still my Linux machine). On the other hand, orange doesn't have 200 megabytes of free space. We do the following:

```
rsh banana dd if=/dev/rst0 ibs=1024 | demux | \
rsh banana dd of=file.dat
```

Now we read from banana, process on orange and write back to banana.

In the next illustration you want to access a printer connected to banana. We have a PostScript file, **test.ps**, that we want to send on a printer called p_a4:

```
dd if=test.ps | rsh banana `lpr -Pp_a4'
```

You might want to have a look at the file before you print it:

```
dd if=test.ps | rsh banana `xv - -display orange:0'
```

This will only work under X-windows. On orange, you would have given a command like **xhost +banana** first.

Problems:

Apart from the fact that some of the above examples probably will not run under some configurations, I have encountered some strange behavior. Consider we have, as in the previous examples, two identical files: **bin.tar** on banana and **bin.tar** on orange. Now I try on orange:

```
ls | grep bin
```

The response is **bin.tar**. The same goes for:

```
rsh banana ls | grep bin
```

but the next one chokes:

```
ls | rsh banana grep bin
```

Piping files into a remote shell with the command **dd**, though, has never choked.

Conclusion:

With the help of these commands, getting connected to other machines is made considerably easier. The **rcp** and **rlogin** commands can be almost fully substituted for commands such as **ftp** and **telnet** on your local network. Not only can you access your accounts, you might also allow other trusted users to access your accounts by means of the **.rhosts** file.

Finally, the **rsh** command enables you to generate data streams through several different machines, accessing local disk and tape drives (or anything else you are allowed to access).

Jens Hartmann (hartmann@dkrz.d400.de) is a geophysicist at the University of Hamburg, where he uses Linux for his work.

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Linux Development Grant Fund

LJ Staff

Issue #9, January 1995

Linux International has announced the formation of the Linux Development Grant Fund, an international fund designed to both promote development for Linux by awarding grants to Linux developers and to give Linux users a way to support Linux development in an organized and efficient fashion. Everybody wins as Linux International creates a grant fund to both promote development and to offer users a way to support that development.

Linux International has announced the formation of the Linux Development Grant Fund, an international fund designed to both promote development for Linux by awarding grants to Linux developers and to give Linux users a way to support Linux development in an organized and efficient fashion.

Everybody wins as Linux International creates a grant fund to both promote development and to offer Linux users a way to support that development.

by *Linux Journal* Staff

Linux International is a worldwide, non-profit organization devoted to promoting Linux development and growth in the international marketplace. The organization has branches in several countries on most continents and, because of this structure, it is able to efficiently collect donations and distribute monies to individual developers with less overhead than if the money came from individual personal contributions. All 100 percent of the monies donated to the fund will be given out in the form of grants; Linux International will not retain any portion of the funds for administrative expenses.

By collecting the funds and then converting many donations at once, a smaller portion of the funds will be lost to currency conversion fees than if the donations were made separately to developers. This is especially important for smaller donations given from one person to someone else with a different local currency; currency conversion carries a fixed rate of about $7 to $15 (sometimes higher), and it can be difficult to do without a cooperative bank.

How will developers be selected to receive grants? Anyone developing free software for Linux with a specific need for funds to further development (for instance, to purchase hardware or documentation) may submit a request. Developers and potential developers can receive information on submitting grant proposals by sending e-mail to grant-submissions-info@li.org. If you do not have e-mail access, send paper mail or a fax to Linux International at the address below.

Who decides who will receive a grant? The Grant Fund is controlled by a board of three members appointed in a more-or-less democratic manner. The board members will each serve one-year terms. The first board members are well-known in the Linux community: Matt Welsh, Ian Murdock, and Michael K. Johnson.

All of the grants awarded will be announced in the comp.os.linux.announce newsgroup. Additionally, a list of all the donors, except for those who choose to remain anonymous, will be published periodically. (A single check-mark on the donation form is sufficient for donors who choose to remain anonymous.)

Donations may be made by credit card, international money order or check and may be sent by paper mail, e-mail or fax. If you wish to send your credit card number via e-mail, you will probably wish to encrypt it with PGP to avoid fraud. Linux International's PGP public key is available by fingering donations@li.org.

Donations to the Grant Fund can be made in almost any currency. However, to avoid excess currency conversion costs, US dollars, Deutsch marks, Pounds Sterling, or Australian dollars are preferred. However, do not send cash through the mail—it is not likely to arrive.

E-mail donations may be sent to donations@li.org, fax donations to +61 9 331 2443 in Australia or (203) 454-2582 in the US, and paper mail donations to Linux Development Grant Fund, c/o Linux International, P.O. Box 80, Hamilton Hill, WA, Australia, 6163.

A form for donating may be requested by sending e-mail to donations-info@li.org; one will be sent to you by return mail.

Because of very complex national laws determining charitable organizations, donations to the Grant Fund are not tax exempt at this time. However, businesses may count donations as business expenses in many countries; consult your local tax experts for details.

If you have any comments or questions about the fund, you may send them to donation-comments@li.org.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Linux System Administration

Mark Komarinski

Issue #9, January 1995

Got that sinking feeling that often follows an overzealous rm? Our system doctor has prescription.

There was recently a bit of traffic on the Usenet newsgroups about the need for (or lack of) an undelete command for Linux. If you were to type **rm * tmp** instead of **rm *tmp** and such a command were available, you could quickly recover your files.

The main problem with this idea from a filesystem standpoint involves the differences between the way DOS handles its filesystems and the way Linux handles its filesystems.

Let's look at how DOS handles its filesystems. When DOS writes a file to a hard drive (or a floppy drive) it begins by finding the first block that is marked “free” in the File Allocation Table (FAT). Data is written to that block, the next free block is searched for and written to, and so on until the file has been completely written. The problem with this approach is that the file can be in blocks that are scattered all over the drive. This scattering is known as *fragmentation* and can seriously degrade your filesystem's performance, because now the hard drive has to look all over the place for file fragments. When files are deleted, the space is marked “free” in the FAT and the blocks can be used by another file.

The good thing about this is that, if you delete a file that is out near the end of your drive, the data in those blocks may not be overwritten for months. In this case, it is likely that you will be able to get your data back for a reasonable amount of time afterwards.

Linux (actually, the second extended filesystem that is almost universally used under Linux) is slightly smarter in its approach to fragmentation. It uses several techniques to reduce fragmentation, involving segmenting the filesystem into independently-managed groups, temporarily reserving large chunks of contiguous space for files, and starting the search for new blocks to be added to a file from the current end of the file, rather than from the start of the filesystem. This greatly decreases fragmentation and makes file access much faster. The only case in which significant fragmentation occurs is when large files are written to an almost-full filesystem, because the filesystem is probably left with lots of free spaces too small to tuck files into nicely.

Because of this policy for finding empty blocks for files, when a file is deleted, the (probably large) contiguous space it occupied becomes a likely place for new files to be written. Also, because Linux is a multi-user, multitasking operating system, there is often more file-creating activity going on than under DOS, which means that those empty spaces where files used to be are more likely to be used for new files. “Undeleteability” has been traded off for a very fast filesystem that normally *never* needs to be defragmented.

The easiest answer to the problem is to put something in the filesystem that says a file was just deleted, but there are four problems with this approach:

1. You would need to write a new filesystem or modify a current one (i.e. hack the kernel).
2. How long should a file be marked “deleted”?
3. What happens when a hard drive is filled with files that are “deleted”?
4. What kind of performance loss and fragmentation will occur when files have to be written around “deleted” space?

Each of these questions can be answered and worked around. If you want to do it, go right ahead and try—the ext2 filesystem has space reserved to help you. But I have some solutions that require zero lines of C source code.

I have two similar solutions, and your job as a system administrator is to determine which method is best for you. The first method is a user-by-user no-root-needed approach, and the other is a system-wide approach implemented by root for all (or almost all) users.

The user-by-user approach can be done by anyone with shell access and it doesn't require root privileges, only a few changes to your **.profile** and **.login** or **.bashrc** files and a bit of drive space. The idea is that you alias the rm command to move the files to another directory. Then, when you log in the next time, those files that were moved are purged from the filesystem using the real **/bin/rm** command. Because the files are not actually deleted by the user, they are accessible until the next login. If you're using the bash shell, add this to your **.bashrc** file:

```
alias waste='/bin/rm'
alias rm='mv $1 ~/.rm'
```

and in your

```
.profile:
if [ -x ~/.rm ];
 then
   /bin/rm -r ~/.rm
   mkdir ~/.rm
   chmod og-r ~/.rm
 else
   mkdir ~/.rm
   chmod og-r ~/.rm
 fi
```

Advantages:

- can be done by any user
- only takes up user space
- /bin/rm is still available as the command waste
- automatically gets rid of old files every time you log in.

Disadvantages:

- takes up filesystem space (bad if you have a quota)
- not easy to implement for many users at once
- files get deleted each login (bad if you log in twice at the same time)

System-Wide

The second method is similar to the user-by-user method, but everything is done in **/etc/profile** and cron entries. The **/etc/profile** entries do almost the same job as above, and the cron entry removes all the old files every night. The other big change is that deleted files are stored in **/tmp** before they are removed, so this will not create a problem for users with quotas on their home directories.

The cron daemon (or **crond**) is a program set up to execute commands at a specified time. These are usually frequently-repeated tasks, such as doing nightly backups or dialing into a SLIP server to get mail every half-hour. Adding an entry requires a bit of work. This is because the user has a **crontab** file associated with him which lists tasks that the **crond** program has to perform. To get a list of what **crond** already knows about, use the **crontab -l** command, for "list the current cron tasks". To set new cron tasks, you have to use the **crontab <*file*** command for "read in cron assignments from this file". As you can see, the best way to add a new cron task is to take the list from **crontab -l**, edit it to suit your needs, and use **crontab <*file*** to submit the modified list. It will look something like this:

```
~# crontab -l > cron.fil
~# vi cron.fil
```

To add the necessary cron entry, just type the commands above as root and go to the end of the **cron.fil** file. Add the following lines:

```
# Automatically remove files from the
# /tmp/.rm directory that haven't been
# accessed in the last week.
0 0 * * * find /tmp/.rm -atime +7 -exec /bin/rm {} \;
```

Then type:

```
~# crontab cron.fil
```

Of course, you can change **-atime +7** to **-atime +1** if you want to delete files every day; it depends on how much space you have and how much room you want to give your users.

Now, in your **/etc/profile** (as root):

```
if [ -n "$BASH" == "" ] ;
then # we must be running bash
   alias waste='/bin/rm'
   alias rm='mv $1 /tmp/.rm/"$LOGIN"'
   undelete () {
     if [ -e /tmp/.rm/"$LOGIN"/$1 ] ; then
       cp /tmp/.rm/"$LOGIN"/$1 .
     else
       echo "$1 not available"
     fi
   }   if [ -n -e /tmp/.rm/"$LOGIN" ] ;
   then
     mkdir /tmp/.rm/"$LOGIN"
     chmod og-rwx /tmp/.rm/"$LOGIN"
   fi
fi
```

Once you restart cron and your users log in, your new `undelete' is ready to go for all users running bash. You can construct a similar mechanism for users using csh, tcsh, ksh, zsh, pdksh, or whatever other shells you use. Alternately, if all your users have **/usr/bin** in their paths ahead of **/bin**, you can make a shell script called **/usr/bin/rm** which does essentially the same thing as the alias above, and create an undelete shell script as well. The advantage of doing this is that it is easier to do complete error checking, which is not done here.

Advantages:

- one change affects all (or most) users
- files stay longer than the first method
- does not take up user's file space

Disadvantages:

- some users may not want this feature
- can take up a lot of space in /tmp, especially if users delete a lot of files

These solutions will work for simple use. More demanding users may want a more complete solution, and there are many ways to implement these. If you implement a very elegant solution, consider packaging it for general use, and send me an e-mail message about it so that I can tell everyone about it here.

Tar Tips

And, as a last-minute correction/addition to a previous article (specifically my article on mtools in *LJ* issue 5), an alert reader noticed that while mtools can copy Unix files to a DOS diskette, how can you preserve the 256 character name of the original Unix file if DOS can only handle 11 characters at most, and is not case-sensitive? The case was one in which two Unix machines could use DOS diskettes, but could not communicate directly. However, this can apply to backups in which you want your files stored on DOS floppies, or to any other case in which you want long file names preserved. There is a way to do it.

The **tar** command is used to create one big file which can contain a number of little files. Using the **tar** command, you can create an archive file which contains a bunch of 256 character file names, while the tar file itself is a legal DOS name. DOS (or the FAT filesystem, anyway) does not care what is in the file, as long as it has at most eight characters plus a three character extension.

Be sure that when you copy the tar file that you do not give the -t (text) option to mtools. The tar file has to be copied in binary format, even if the tar file only contains text files.

So, to copy a few long filenames to the first floppy drive (**A:** or **/dev/fd0**):

```
tar -cvf file.tar longfilename \
reallylongfilename \ Not.In.Dos.Format.Filename.9999 /
mcopy file.tar a:
```

Then at the remote Unix machine (or to restore it):

```
mcopy a:file.tar file.tar
tar -xvf file.tar
```

or

```
mread a:file.tar | tar -xf -
```

And assuming the remote Unix system has mtools and supports 256 character filenames, a copy of the files will now be on each system.

Tune in next time when I find the real relationships between virtual beer, BogoMIPS, and a VIC-20. In the meantime, please send me your comments or questions or even suggestions for future articles to: komarimf@ craft.camp.clarkson.edu.

Mark Komarinski (komarimf@craft.camp.clarkson.edu) graduated from Clarkson University (in very cold Potsdam, New York) with a degree in computer science and technical communication. He now lives in Troy, New York, and spends much of his free time working for the Department of Veterans Affairs where he is a programmer.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Unix Systems for Modern Architectures

Randolph Bentson

Issue #9, January 1995

Understanding the subtleties of hardware-cache-bus-memory interactions is an essential component of “doing” a kernel for a multiprocessor system.

Book Review

Unix Systems for Modern Architectures

Author: Curt Schimmel

Publisher: Addison-Wesley

ISBN: 0-201-63338-8

Reviewer: Randolph Bentson (bentson@grieg.seaslug.org)

“What is involved in a multiprocessor version of Linux?” has almost become a “Frequently Asked Question” in the Linux newsgroups. The answer is contained in Curt Schimmel's *UNIX Systems for Modern Architectures.*

Schimmel can speak from experience on this topic. He worked on Unix systems at AT&T Bell Laboratories and at Silicon Graphics, Inc., and has offered tutorials on symmetric multiprocessor Unix systems at USENIX and UKUUG. This book is an outgrowth of those tutorials.

At first glance, the book seems to offer too much detail about hardware for a programmer. But as one proceeds, one sees that understanding the subtleties of hardware-cache-bus-memory interactions is an essential component of “doing” a kernel for a multiprocessor system.

After a brief (17 page) description of Unix processes, another 130 pages are devoted to discussing uniprocessor cache systems. I was surprised and delighted to find out how hard it can be to get the right results. Fortunately, folks do seem to have done this right on the systems I've used.

With this foundation well established, the remainder of the book deals with the new domain of multiprocessor systems.

The keys to any such system are protecting shared data and efficient interprocess communication. Mutual exclusion mechanisms are cast in three forms - short term, medium term, and long term. We are shown how uniprocessor implementations of Unix depended on a single-threaded kernel and interrupt masking to protect shared data and, more importantly, we are shown how these methods are inappropriate for a multiprocessor system.

Schimmel shows how one can build locks for all three levels of mutual exclusion (and points out where they are needed in a typical Unix kernel). Although the master/slave scheme is straightforward to implement, it has much the flavor (and bottlenecks) of a uniprocessor system. The more promising symmetric multiprocessor scheme is not as easy to do correctly. The essence of the problem is finding the right granularity (or size) for the critical sections. Granularity that is either too large or too small can harm system performance. We are shown the analysis that leads to good designs.

The book concludes with more memory access and caching issues - this time with multi-processor systems. Some recent RISC chips have memory models which allow for stores and loads to be re-ordered from what the programmer intended, in order to gain performance. We are shown how RISC chips have mechanisms to force the correct results for implementing locks and accessing data in critical sections. Even when memory requests are issued in the order they were programmed, cache consistency is a serious issue in multiprocessor systems. The final chapters of the book address the interactions that must be dealt with by a serious system designer.

This book is written as a textbook, with questions and references at the end of each chapter. Selected questions have answers provided in an appendix. Another appendex summarizes a dozen popular chips found in Unix systems.

Randolph Bentson can be reached at: (bentson@grieg.seaslug.org)

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

LINUX JOURNAL

Advanced search

Letters to the Editor

Various

Issue #9, January 1995

Readers sound off.

Who's Counting?

Thanks for this superb magazine. I'd find it very informative if you could start a counter for important and/or interesting applications—both commercial and not—that have been ported to Linux. A compact layout might include one line per application: name and a short description of what it does. Also, articles about windows emulation and WWW are very welcome! Cheers, —Veli-Pekka Pulkkinenvpp@vipunen.hut.fi

LJ responds:

As far as non-commercial applications, almost everything has been ported—far too much to include in a counter in a small magazine. Most of the commercial apps that have been ported now advertise in *Linux Journal*, so a counter for them would be redundant.

Some day, we might be able to provide something like this. We would need to be a larger magazine (more pages) and have a clearer vision of what would be listed and what not.

We will try to keep occasional progress reports about Wine coming, and we arranged for Bob Amstadt, the head of the Wine project, to speak last month at the Open Systems World Linux Conference, as well.

Hacker's OS

I have been running the Slackware Pro 2.0 distribution at home now for about a month, and I LOVE Linux! It is a fast, well-thought-out OS. *Linux Journal* is undoubtedly the most useful, well-written magazine I have EVER subscribed to. I look forward to each new issue and read it many times. It's great to have a hacker's OS, instead of being forced into MS-DOS or MS-Windows. —Stephen E. Farlowsefarlow@crl.com KJ5YN

ez = ?

Terry Gleidt's articles on AUIS in *LJ* (issue #4) got me interested in ez, and I think it's great, but I can't get the equation editor to print equations properly. I'm using the auis63L1-wp package from sunsite with Slackware 2.0, including groff 1.09 and ghostscript 3.01 but the equations look like they have skipped a formatter, and the program won't convert them to other formats. y=x^2 comes out like this:

```
delim##define above "to" define below "from" define zilch "" define ...
#y=x sup[2]#delim off
```

How can I fix this?

Also, the help file mentions a “chart” program that isn't included. Where can I find this? —David Jacksonjackson@cfn.cs.dal.ca

Terry responds:

In **/usr/andrew/README*** it says:“If you print/preview equations, you should modify /usr/andrew/etc/{atkprint ,atkpreview} so that geqn and gtbl are invoked. See the comments in [those shell scripts] for more details.”

Hope this helps. If you have more questions, drop me some mail. —Terry Gliedttpg@mr.net

LJ responds, too:

Although this bug is probably fixed in a newer release of Terry's AUIS packages, I noticed that it exists with the package I have, as well. Try (as root, or some other user with write permissions in /usr/ andrew): **cd /usr/andrew/ bin; ln -s runapp chart** which worked for me.

Stop: You're Making Us Blush

I received in the mail today my seventh issue of *Linux Journal*. Thank you for providing the Linux community with such a fine publication. I started running Linux on a 486DX33 machine just about one year ago. At that time, I was very new to Unix and Un*x-like operating systems. However, over the past 12 months, I've installed and configured several Linux machines, learning more each time than I knew before. I'm now very comfortable administering my own system, as well as offering advice to people who are just starting out with an advanced OS.

Fortunately for me, I subscribed to *LJ* well before the first issue went to press. I say “fortunately” because your publication has provided me with timely advice and guidance. In fact, I am in your debt for publishing such clear, concise, and pertinent Linux-specific information as that which appears in Clarence Smith's article, “Linux Performance Tuning for the Faint of Heart” (issue #7). Please keep up the good, practical job you are doing.

Moreover, it has been a real joy watching *LJ* develop. I'm eager to see where the next 9 issues will take all of us. —Louis Dehnerlouis@winter.net.com

Debugging the Printer

I am a regular reader of this wonderful journal. I would like to read some articles on printing, specifically, with the following items: 1) a minimal working printcap for a single machine with a printer and a client machine of a Linux print server; 2) filters for text and postscript files; 3) notes on obvious bugs, if any (for example, when the printer is on everything works fine, but when the printer is off, the print queue empties as if lpd is sending the files through the printer port); and 4) where to find the latest lp stuff. —Genaldo L. Nunesnunes@mtm.ufsc.br

LJ responds:

We have an author who is currently writing an article on printing, although we haven't scheduled it for print yet.

Correction

In the December issue of *Linux Journal*, the address of Linux International was given as info@li.org.au. This has now been changed to info@li.org.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

What We've Been Up To

Michael K. Johnson

Issue #9, January 1995

Our greatest improvements will come from our readers.

For several issues, I have sacrificed the space reserved for this column to include more interesting and useful articles, but it is time to give you an update on what is happening at *Linux Journal*.

Despite several setbacks, including my computer dying, *LJ* has improved over the past few months, as we are told over and over again in letters to the editor. However, we see much more room for improvement—and the more subscribers we have, the faster we are able to improve. Our thanks to all our subscribers!

Our greatest improvements will come from our readers. There is only so much that we can write; you, our readers, use Linux for things that we haven't thought of yet. In this issue, for instance, you'll read how Vance Petree at Virgina Power has implemented a system for managing large amounts of data using Linux systems (see page 23). In our September issue, Greg Wettstein wrote about using Linux to manage patient care for a large cancer research center. It is our policy to print at least one article each month about how Linux is being used in the real world, but we are dependent on you, our readers, to keep us informed.

What We've Been Up To

We have hired several new staff members to process all our new subscription orders and to spend more time editing the articles. We have designed short monthly features with useful information, including ftp sites where information about Linux is available. Over the last few issues, we have instituted a policy of including a guide to available applicable resources of all types (including Internet sites, WWW URLs, and books) with most articles. We are now in the process of publishing a book called The Linux Sampler, filled with a mix of articles from *Linux Journal*, with sections on Linux history, systems administration, resources, and real world applications.

We exhibited at Unix Expo, as was covered in last month's *Linux Journal*, and helped make technical contacts between Linux developers and hardware and software vendors interested in Linux. We sponsored a two-day Linux Conference at Open Systems World in December, which included several short classes on a variety of topics and one full-day tutorial introduction to Linux.

Also, we like having fun with Linux just as much as the rest of you do. Many of you have seen our “My Other Computer Is a Linux System” stickers and t-shirts; we are now offering a Linux bumpersticker, and we sell other Linux-related products through our catalog. If you have other ideas for fun Linux-related items, feel free to send them to info@linuxjournal.com.

Michael K. Johnson is the editor of *Linux Journal*, and is also the author of the Linux Kernel Hackers' Guide (the KHG).

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Looking into the Future

Phil Hughes

Issue #9, January 1995

As I write this in mid-November and look back on 1994, I see some amazing happenings in the computing field—and most of the amazement has to do with Linux itself.

Here are a few examples:

- Decus invites Linus to speak at their conference in New Orleans.
- The Australian Unix Users' Group invites Linus to speak at their annual meeting.
- *Linux Journal* has a booth at Unix Expo and gives out 3,000 copies of the magazine.
- PC Week names Linux as software Product of the Week.
- Rumors are that HP, IBM, and other big names are using Linux internally.
- I went into three mainstream computer stores and said I was looking for a laptop to run Linux. One store didn't know what I was talking about. Another suggested a particular system because it had a larger hard disk which would be good for Linux. The other told me that the laptop they sold didn't run Linux (they had tried it), but suggested another brand that did.
- Companies such as the Roger Maris Cancer Center and Virginia Power port applications from commercial operating systems to Linux and explain that they did it because they needed something “better”.
- Open Systems World, in its sixth year as a Washington, DC, based trade show decides to have a Linux Conference right along with the SCO and Solaris conferences.
- Unix Expo contacts *Linux Journal* about doing a Linux section at their New York show in 1995.

Why is this amazing?

Because these aren't “Linux events”, these are cases of the “real world” showing an interest in Linux. At the beginning of the year, mainstream computing hadn't even heard of Linux. At Uniforum in March, *Linux Journal* had a booth. Although the booth was very popular, the most common question was “What is Linux?” Today people seem more likely to ask “Why should I run Linux?”, and then actually listen to the answer.

Now comes the assignment. If we want people to take us seriously we need to give good answers. Yes, Linux is a fun “hacker” system. But some of these people want to do real work. We need to listen to their needs and either explain how Linux can meet those needs or, if it can't, tell everyone who is interested in development what these people are looking for.

And this is where *Linux Journal* can help. Let us know what people are asking for and we'll help get the word out.

To start off this effort, here is an idea I am working on—a Linux-based “freenet” system. For those of you not familiar with a freenet, it is a public-access computer system designed to offer communications within a local community. Access is free, users are generally not computer oriented and much of the information on the system is supplied by local volunteers.

Current freenet software has a lot of shortcomings which include being inefficient, not particularly user-friendly and generally running on expensive hardware. Combining the openness and capabilities of Linux with this movement could offer a much better starting point for these freenet projects. And implementing a freenet on Linux could help us spread the word on how useful Linux can be.

In future issues of *Linux Journal* I want to explore the current limitations and see what Linux can offer in the way of a good base for the freenet of the future. Again, your input is welcome. If you want to get involved, write me via *Linux Journal* or e-mail info@linuxjournal.com.

Phil Hughes is the publisher of *Linux Journal*.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Virginia Power—Linux Hard at Work

Vance Petree

Issue #9, January 1995

This is a story of Linux in the Real World—a tale rife with adventure and suspense, brimming with excitement and sacrifice and drama and—well, maybe not all of those things. But it is a story of the considerable (and ongoing) success of Linux in an area that affects just about all of us: electric power. And as for the suspense and excitement—well, you'll see. But first we need a little background, and then we'll set the scene.

I'm a programmer for Virginia Power, an electric utility that serves about two million customers in much of Virginia and a small part of Northeastern North Carolina. Virginia Power's service territory is partitioned into five divisions: Northern, Southern, Eastern, Western, and Central. Among the responsibilities of the Operations Services group in which I work is the maintenance of a centralized archive database of 30-minute averages of all analog values which are retrieved by SCADA master computers located in each division. This averaged data is of great importance to system planners and load forecasters, and is vital when planning large construction projects which may cost millions of dollars. Processing all of this data is a demanding task, but Linux has saved the day.

A few words of explanation for those of you who are not power system aficionados: SCADA is an acronym for Supervisory Control And Data Acquisition, which basically means the retrieval of real-time analog and status data from various locations in the service territory through remote terminal units (RTUs) installed in substations. This information is obtained over dedicated communications lines by central master computers, where it is stored, analyzed, and presented to operations personnel who make sure the lights stay on, and who get *very* busy when the lights aren't on. These system operators (as they're called) can also remotely operate field devices like line breakers and capacitor banks when necessary (this is the Supervisory Control part of SCADA); the master computers even contain several feedback algorithms which can automatically operate devices based on system conditions.

Virginia Power has another SCADA system which monitors the entire power grid and also provides automatic feedback control to the generation stations; this is called the Energy Management System, or **EMS**. And in case you're wondering: yes, the EMS retrieves analog values just like the division computers. Yes, we archive them into our central database. And yes, Linux has saved the day here as well.

Also scattered throughout the service territory are scores of intelligent electronic devices such as digital megawatt-hour meters, data recorders, fault detecting relays, and line tension monitors, which are not directly tied into the SCADA systems, but which must periodically be dialed into over the public phone network to obtain data for use not only by system operators, planners, and engineers, but by folks in the accounting department as well. You guessed it—Linux has saved the day in *this* area, too!

That should be enough background to get us started; now comes the scene as it existed when I joined the Operations Services group in late 1992:

Averaged analog data was dumped by the division SCADA computers every 15 minutes over dedicated serial lines to redundant PDP 11/84 computers located in our local computer room; these machines strained (and frequently failed) to process all the information being shovelled at them. Averaged analogs from the EMS system were dumped every 6 hours to a MicroVAX over a serial DECnet link which, due to internal security concerns, could only be activated when a transfer was to take place. Another computer, an IBM PS/2 model 60 running some of the most odious commercial software I have ever seen (which will remain nameless; it's not polite to insult the dead), slogged through dialing as many digital meters and recorders as possible, one at a time, over a single phone line.

Once a day in the early morning hours, all of the previous day's information from the PDPs, the MicroVAX, and the PS/2 was masticated in an orgy of sorting, calculating, merging, interpolating, and updating, and finally reduced to a set of 30-minute averages which were then shipped over DECnet to our main archive system (a VAX 4000) and merged into the master database. Whew! I was caught up in other projects at the time, but in my spare moments I looked into the scraps of code for this system—all I remember of those encounters are shudders, cold sweats, and endless nightmares of Fortran source.

Enter the first hero of our tale: Joel Fischer, engineer and technical evangelist extraordinaire, a believer in the true potential of microprocessor-based computers, and a Unix initiate—something of a rarity at a utility company. (Most companies, as I sure all of you know, develop a dominant computer philosophy which tends to color any approaches to a problem. At many utility companies, the official watchwords are often IBM Mainframes, VAXes running VMS, PCs running DOS or Windows or Novell. Very little TCP/IP. And very little Unix.)

Joel joined our group in early 1993. His primary responsibility was maintenance of the averaged analog database system. After a short time of dealing with balky PDPs, uncooperative PS/2s, and temperamental MicroVAXes, he began to share my conviction that there *had* to be a better way to process all of our incoming data.

And so began a series of productive and energizing "cubicle chats". Joel would drop by my cubicle with some ideas on how to improve the system, and I would reciprocate with some ideas on *his* ideas, and so on. Our group is something of a skunk works anyhow, so this low-overhead approach to problem solving was a well-established principle. Joel was much more familiar than I with our company's network, and informed me of two important facts: Our enterprise backbone reached all division offices except the Western division, and all of our routers were native TCP/IP—despite the fact that TCP/IP wasn't used by more than a handful of special-purpose systems.

Well, well, *well*. By the most serendipitous of circumstances, I was at that same time looking into Linux on my home system, and I was impressed enough that I waxed somewhat evangelical myself. I had been the Minix and Coherent route, and I was no stranger to GNU and the FSF; we were running gcc and Emacs on our VAX 4000.

It had taken me exactly one evening at home to realize that Linux was a Good Thing. A few weeks of use and my software intuition (I'd like to think that 18 years of programming has been good for *something*) told me that Linux was a Very Good Thing Indeed. So I cleared off a disk partition on my machine at work and installed Linux, to demonstrate that it could do the things I claimed it could. Conversations with visitors to my cubicle followed this general pattern:

"Hey, what's that?"

"Linux, a copylefted Unix clone..." I gave a short speech on free software and the many advantages thereof. But you already *know* all of that.

"Huh? Will it do (fill in the blank)?"

"Sure." Clickety-clack on the keys. "There you go."

(And when I fired up X-windows—oh, my!)

Figure 1. AMC Dialing Subsystem

Gradually, the design of a new, distributed data gathering system took shape. We could install a PC in each of the division computer centers to receive the averaged analog data from the SCADA master computer and dial all of the field devices (meters, relays, etc.) in that division. A machine in our central office would receive a duplicate data feed from the division SCADA computers to provide data redundancy, and serve as a backup dialing system. All of these PC systems could be linked together over the corporate network with TCP/IP (except for Western division, where we could use UUCP). Vax connectivity was something of a problem, but we settled on a high-speed dedicated serial link with a simple file transfer protocol.

Of course, it wasn't all smooth sailing. We had our detractors, who questioned the idea of using *personal computers* to replace minicomputers. How could we be sure a PC was powerful enough to process all of the incoming data and dial remote devices and handle networking? And what about all of the software we would need to develop in-house? Protocol translators to talk to all of the remote devices? Software to do data translation and reduction into a format suitable for submission to our VAX database system? (All valid concerns, certainly. Many discussions with our “devil's advocates” were invaluable in helping to hammer out the gritty details of our system and provide the best answer of all: working, reliable software!)

Enter the second hero in our tale: Lynn Lough, our supervisor. She knew how important this data was to efficient company operations and planning, and she understood the need for a reliable, redundant retrieval system. Joel and I presented our proposed system. Hardware costs: the price of six 486/66 PCs with healthy doses of RAM and disk space. Software costs: zip. (Well, not *exactly*. Don't forget all of the in-house software we needed to develop. But you get the idea.)

We explained to her the underlying philosophies of the Free Software Foundation; how *free* did not always signify bad, but often meant better—because the software was not shoved out the door to meet some arbitrary marketing deadline, but was released in an environment of continual refinement and improvement, where hundreds of the sharpest software minds anywhere (that's you, folks!) would provide feedback. Besides, with all of the source code for everything, we would never be at the mercy of a vendor who decided to drop support for a particular piece of software, or who only fixed a bug by upgrading and charging a princely sum for the upgrade.

Lynn weighed all the pros and cons. She must have seen the fervor in our eyes, because she made the single most fateful decision in our entire story: She said *yes*.

Yow! Within weeks, by mid-August, we had 6 dandy new PCs in our computer room. As the primary coder for our project (actually, because we were a little short-handed, I was the *only* coder), I rolled up my virtual sleeves and dove in.

(And after a year of developing code for Linux, I'll say one thing loud and clear: I'd crawl over an acre of 'Visual This++' and 'Integrated Development That' to get to gcc, Emacs, and gdb. Thank you. Back to our story.)

Our first step was to choose a kernel which provided all of the services we needed—nothing exotic, just solid networking and System V-type IPC primitives. We settled on 0.99.13s. We purchased 8-port serial boards to give us 10 serial ports per PC. A few quick changes to serial.c to support the board (impossible without access to the source code, I repeated *ad infinitum* to anyone who would listen) and we had our base system. As soon as I have a breather, I'd like to post the patches for the serial board we're using: an Omega COMM-8 which provides individually-selectable IRQs for each port. In the meantime, please e-mail me if you're interested.

Our next step was to replace the ailing PDPs as quickly as possible, since their unreliability was resulting in lost data. We already had dual data feeds. (Actually, data from each division came in over a single serial line and was split off at the back end of the modem, so we didn't really have redundant data input; that was one of the weaknesses our system was designed to rectify.) We decided to take two of our PCs and temporarily configure them as plug-in replacements for the PDPs. To accomplish this, we had to be able to accept input data in its current form (for various reasons, the division master computers couldn't be changed to modify the way they sent the averaged data to us), reduce it to database input format, and move it to the database VAX.

I took the Unix toolkit philosophy to heart, and instead of a big blob program to do Absolutely Everything, I wrote a set of small utilities, each of which did one thing and one thing only. For run-time modifications I used small configuration text files, and for interprocess communications (where necessary) I used message queues.

Pretty soon, I had an input daemon that hung on the serial lines watching for data, a checker program that made sure we got our quarter-hour files in a timely fashion and verified the points they contained, an averager that calculated 30-minute average files, a builder that built a database submission file hourly, and a PC-to-VAX daemon that implemented a simple transfer protocol over a high-speed serial line. (Of course, this last program required a corresponding daemon on the VAX side, which meant a foray into VMS-land. Good thing I had gcc and Emacs over there!)

By the third week of October we were ready for a trial run. We halted one of the PDPs, moved the data input lines over to our PC, and booted Linux. Within 15 minutes, our next cycle of averaged data had been sent, and tucked safely away on our PC's hard drive were 5 of the most wonderful data files I had ever seen: an input file from each division, every byte present and accounted for. Within 30 minutes, we had our first calculated average file. Within an hour, we had our first database transfer file, neatly deposited in a spooling directory where eventually it would make its way to the VAX.

I didn't get much sleep that night. Logged in from home, I watched each data cycle come in with the sort of anticipation usually reserved for really good science fiction movies. Of course, I had tested all of this software beforehand, but seeing it work with real data in real time was more exciting than I care to admit.

After this milestone, events moved pretty quickly. By the beginning of November, the remaining PDP was relegated to backup status, and our PC was the primary data source for our archive database. By the end of the year, that PDP was gone as well; a second PC would serve as our backup machine while we installed our PCs in the division operating centers.

Over the next several months, as our PCs gradually migrated out to their permanent homes in the division SCADA master computer centers, I turned my efforts to meeting our intelligent device dialing needs—remember all of those megawatt-hour meters, data recorders, and so on requiring periodic polling? More code to develop—just my cup of tea!

To meet our current requirements, as well as provide for future dialing needs, I designed a general-purpose dialing system which could connect to just about any device with a modem and a byte oriented communications protocol. Requests to dial devices are posted to a message queue; a daemon process manages all the phone lines allocated to device dialing. When a request comes in on the message queue, the dial-up manager allocates a phone line, forks a copy of itself to handle the chores of dialing and connecting to the device, and once a connection is established, execs the appropriate protocol task to actually talk to the remote device. When the protocol process terminates, the parent dial-up manager recycles the available phone line for any further dial requests. If there are more dialing requests than available phone lines, the dial-up manager maintains an internal queue with all the usual timeouts, etc.

Figure 2. Linux-Based Data Retrieval System

This dial-up manager scheme should sound pretty familiar—it was inspired by the **inetd** superserver. Now what was that quote by Newton about standing on the shoulders of giants...?

I won't go into the fascinating and esoteric details of writing protocol tasks for the various devices we interrogate. So far, we've developed protocol tasks to talk to half a dozen different types of devices, with more on the way. At last count, our Linux PCs dial nearly a hundred separate devices on a regular basis.

Our story has pretty much reached the present day, and the sailing has gotten smoother and smoother. We've installed a TCP/IP stack on our VAX database machine, so connectivity with our Linux machines is easier than ever. We've installed a sixth remote Linux machine in our System Operations Center (that's the EMS system mentioned at the beginning of our tale) to take care of retrieving EMS averaged analogs, along with handling some dialing requirements for that department. And we're currently developing a virtual dial-up SCADA system to supplement our SCADA Master computers... But that's a topic for another article.

Our network of Linux systems has been handling round-the-clock data retrieval chores—processing about 12,000 data points every 15 minutes—for nearly a year, and *not* a single byte of data has been lost due to any system software problems! I can think of no better tribute to all the hard-working and immensely talented Linux developers than the simple fact that our systems purr contentedly hour after hour after hour, utterly reliable. By golly, I'm beginning to think Linux really *is* the best thing since sliced bread!

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

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

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

New Products

LJ Staff

Issue #9, January 1995

Comeau Computing Releases C++3.0.1

Comeau Computing Releases C++3.0.1

Comeau Computing has released *Comeau C++ 3.0.1 with Templates*. This high-quality, cfront-based C++ compiler comes with lifetime technical support available through Internet e-mail, vendor conferences on BIX, Compuserve, and Prodigy, as well as fax and voice telephone numbers. Comeau C++ for Linux requires at least 2MB of free RAM, 2MB of free disk space, and gcc; it sells for $250, with free second-day air shipping in the continental U.S. Comeau maintains a presence on the ISO/ANSI C++ standards committee.

Comeau Computing also has a Bourne shell compiler called *CCsh*, which is also available for Linux; contact the company for details.

Comeau Computing can be reached at 9134 120th Street, Richmond Hill, NY 11418; phone (718) 945-0009; fax (718) 441-2310; e-mail on the Internet c++@csanta.attmail.com, on BIX as comeau, on Prodigy as tshp50a, or on CompuServe as 72331,3421.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

An introduction to block device drivers

Michael K. Johnson

Issue #9, January 1995

Last month, we inaugurated a column on Linux kernel programming with an article on how to write Linux device drivers without doing any kernel programming. This month we touch the kernel as we explore block device drivers.

It is customary for authors explaining device drivers to start with a complete explanation of character devices, saving block device drivers for a later chapter. To explain why this is, I need to briefly introduce character devices as well. To do that, I'll give a little history.

When Unix was written 25 years ago, its design was eclectic. One unusual design feature was that every physical device connected to the computer was represented as a file. This was a bold decision, because many devices are very different from one another, especially at first glance. Why use the same interface to talk to a printer as to talk to a disk drive?

The short answer is that while the devices are very much different, they can be thought of as having most of the same characteristics as files. The entire system is then kept smaller and simpler by only using one interface with a few extensions.

This is fine, except that it hides important differences between devices. For example, it is possible to read any byte on a disk at any time, but it is only possible to read the **next** byte from a terminal.

There are other differences, but this is the most fundamental one: Some devices (like disks) are **random-access**, and others (like terminals) are **sequential-access**. Of course, it is possible to pretend that a random-access device is a sequential-access device, but it doesn't work the other way around.

A practical effect of the difference is that filesystems can only be mounted on block devices, not on character ones. For example, most tapes are **character** devices. It is possible to copy the contents of a raw, quiescent (unmounted and not being modified) filesystem to a tape, but you will not be able to mount the tape, even though it contains the same information as the disk.

Most textbooks and tutorials start by explaining character devices, the sequential-access ones, because a minimal character device driver is easier to write than a minimal block device driver. My own *Linux Kernel Hackers' Guide* (the *KHG*) is written the same way.

My reason for starting this column with block devices, the random-access devices, is that the KHG explains simple character devices better than it does block devices, and I think that there is a greater need for information on block devices right now. Furthermore, **real** character device drivers can be quite complex, just as complex as block device drivers, and fewer people know how to write block device drivers.

I am not going to give a complete example of a device driver here. I am going to explain the important parts, and let you discover the rest by examining the Linux source code. Reading this article and the ramdisk driver (**drivers/block/ramdisk.c**), and possibly some parts of the KHG, should make it possible for you to write a simple, non-interrupt-driven block device driver, good enough to mount a filesystem on. To write an interrupt-driven driver, read **drivers/block/hd.c**, the AT hard disk driver, and follow along. I've included a few hints in this article, as well.

The Heart of the Driver

Whereas character device drivers provide procedures for directly reading and writing data from and to the device they drive, block devices do not. Instead, they provide a single **request()** procedure which is used for both reading and writing. There are generic **block_read()** and **block_write()** procedures which know how to call the **request()** procedure, but all you need to know about those functions is to place a reference to them in the right place, and that will be covered later.

The **request()** procedure (perhaps surprisingly for a function designed to do I/O) takes no arguments and returns void. Instead of explicit input and return values, it looks at a queue of requests for I/O, and processes the requests one at a time, in order. (The requests have already been sorted by the time the **request()** function reads the queue.) When it is called, if it is not interrupt-driven, it processes requests for blocks to be read from the device, until it has exhausted all pending requests. (Normally, there will be only one request in the queue, but the **request()** procedure should check until it is empty. Note that other requests may be added to the queue by other processes while the current request is being processed.)

On the other hand, if the device is interrupt-driven, the **request()** procedure will usually schedule an interrupt to take place, and then let the interrupt handling procedure call **end_request()** (more on **end_request()** later) and then call the **request()** procedure again to schedule the next request (if any) to be processed.

An idealized non-interrupt-driven **request()** procedure looks something like this:

```
static void do_foo_request(void) {
repeat:
  INIT_REQUEST;
  /* check to make sure that the request is for a
     valid physical device */
  if (!valid_foo_device(CURRENT->dev)) {
     end_request(0);
     goto repeat;
  }
  if (CURRENT->cmd == WRITE) {
     if (foo_write(
          CURRENT->sector,
          CURRENT->buffer,
          CURRENT->nr_sectors < 9)) {
        /* successful write */
        end_request(1);
        goto repeat;
     } else
        end_request(0);
        goto repeat;
     }
  if (CURRENT->cmd == READ) {
     if (foo_read(
          CURRENT->sector,
          CURRENT->buffer,
          CURRENT->nr_sectors << 9)) {
        /* successful read */
        end_request(1);
        goto repeat;
     } else
        end_request(0);
        goto repeat;
     }
  }
}
```

The first thing you notice about this function may be that it never explicitly returns. It does not run off the end and return, and there is no return statement. This is not a bug; the **INIT_REQUEST** macro takes care of this for us. It checks the request queue and, if there are no requests in the queue, it returns. It does some simple sanity checks on the new **CURRENT** request if there is another request in the queue to make **CURRENT**.

CURRENT is defined by default as

```
blk_dev[MAJOR_NR].current_request
```

in drivers **/block/blk.h**. (We will cover **MAJOR_NR** and **blk.h** later.) This is the **current** request, the one at the head of the request queue that is being processed. The request structure includes all the information needed to process the request, including the device, the command (read or write; we'll assume read here), which sector is being read, the number of sectors to read, a pointer to memory to store the data in, and a pointer to the next request. There is more than that, but that's all we are concerned with.

The **sector** variable contains the block number. The length of a sector is specified when the device is initialized (more later), and the sectors are numbered consecutively, starting at 0. If the physical device is addressed by some means other than sectors, it is the responsibility of the **request()** procedure to translate.

In some cases, a command may read or write more than one sector. In those cases, the **nr_sectors** variable contains the number of contiguous sectors to read or write.

end_request() is called whenever the **CURRENT** request has been processed—either satisfied or aborted.

If it has been satisfied, it is called with an argument of 1 and, if it has been aborted, it is called with an argument of 0. It complains if the request was aborted, does magic with the buffer cache, removes the processed request from the queue, “ups” a semaphore if the request was for swapping, and wakes up all processes that were waiting for a request to complete.

It may allow a task switch to occur if one is needed.

end_request() is a static function defined in **blk.h**. A separate version is compiled into each block device driver, using special **#define**'d values that are used throughout blk.h and the block device driver. This brings us to...

blk.h

We have already seen several macros which are very helpful in writing block device drivers. Many of these are defined in **drivers/block/blk.h**, and have to be specially set up.

At the top of the device driver, after including the standard include files your driver needs (which must include **linux/major.h** and **linux/blkdrv.h**), you should write the following lines:

```
#define MAJOR_NR FOO_MAJOR
#include "blk.h"
```

This, in turn, requires that you define **FOO_MAJOR** to be the major number of the device you are writing in **linux/major.h**.

Now you need to edit **blk.h**. One section of **blk.h**, right near the top, includes definitions of macros that depend on the definition of **MAJOR_NR**. Add an entry to the end which looks like this:

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

These are the required macros for each block device driver. There are more macros that can be defined; they are explained in the KHG.

DEVICE_NAME is the name of the driver. The AT hard drive driver uses the abbreviation “hd” in most places; for example, the **request()** procedure is called **do_hd_request()**. However, its **DEVICE_NAME** is “harddisk”. Similarly, the floppy driver, “fd”, has a **DEVICE_NAME** of “floppy”. Other drivers are even more descriptive; read **blk.h** and follow suit.

DEVICE_REQUEST is the **request()** procedure for the driver.

DEVICE_NR is used to determine the actual physical device. For example, the standard AT hard disk driver uses 64 minor devices for each physical device, so **DEVICE_NR** is defined as (**MINOR(device)>6**). The SCSI disk driver uses 16 minor device numbers per physical device, so for it, **DEVICE_NR** is defined as (**MINOR(device)>4**). If you have only one minor device number per physical device, define **DEVICE_NR** as (**MINOR(device)**).

DEVICE_ON and **DEVICE_OFF** are only used for devices that have to be turned on and off. The floppy driver is the only driver that uses this capability. You will most likely want to define these to be nothing at all.

All these macros, as well as many others, can be used in your driver where appropriate. **blk.h** includes a lot of macros, and studying how they are used in other drivers will help you use them in your own driver. I won't document them fully here, but I will briefly mention some of them to make your life easier.

DEVICE_INTR, **SET_INTR**, and **CLEAR_INTR** make support for interrupt-driven devices much easier. **DEVICE_ TIMEOUT**, **SET_TIMER**, and **CLEAR_TIMER** help you set limits on how long may be taken to satisfy a request.

The First Shall Be the Last

I've saved the first, and perhaps most important, thing for last. Before you can read or write a single block, the kernel has to be notified that the device exists. All device drivers are required to implement an initialization function, and there are some special requirements for block device drivers. Here is a sample idealized initialization function:

```
long foo_init(long mem_start, int length)
{
  if (register_blkdev(FOO_MAJOR,"foo", & foo_fops)) {
    printk("FOOBAR: Unable to get major %d.\n",
           FOO_MAJOR);
    return 0;
  }
  if (!foo_exists()) {
    /* the foobar device doesn't exist */
    return 0;
  }
  /* initialize hardware if necessary */
  /* notify user device found */
  printk("FOOBAR: Found at address %d.\n",
           foo_addr());
  /* tell buffer cache how to process requests */
  blk_dev[FOO_MAJOR].request_fn = DEVICE_REQUEST;
  /* specify the blocksize */
  blksize_size[MAJOR_NR] = 1024;
  return(size_of_memory_reserved);
}
```

The three things here that are specific to block device drivers are:

- **register_blkdev()** registers the file operations structure with the Virtual Filesystem Switch (VFS), which is the system that manages access to files.
- **blk_dev** tells the buffer cache where the request procedure is.
- **blksize_size** tells the buffer cache what size blocks to request.

It is worth noting that the hardware device detection and initialization, which I have denoted as **foo_exists()** here, is very delicate code. If you can rely on a string somewhere in the BIOS of the computer to determine whether the device exists and where it is, it's relatively easy. However, if you have to check various I/O ports, you can hang the computer by writing the wrong value to the wrong port, **or even reading the wrong port**. Check only well-known ports if you must check ports, and provide kernel command-line arguments for other ports. To do this, read **init/main.c** and add a section of your own. If you can't figure out how to do it, an explanation is forthcoming in the next version of the KHG.

Of course, none of this initialization will happen if **foo_init()** is never called. Add a prototype to the top of **blk.h** with the other prototypes, and add a call to **foo_init()** in **ll_rw_blk.c** in the **blk_dev_init()** function. That call should be protected by **#ifdef CONFIG_FOO** like the rest of the ***_init()** functions there, and a corresponding line should be added to the **config.in** file:

```
bool `Foobar disk support' CONFIG_FOO y
```

drivers/block/Makefile should have a section added that looks like this:

```
ifdef CONFIG_FOO
OBJS := $(OBJS) foo.o
```

```
    SRCS := $(SRCS) foo.c
    endif
```

This done, configuration should work correctly. Your device driver file does not need to have any references to **CONFIG_FOO**; the only specific reference to it is commented out in **ll_rw_blk.c**, and the makefile only builds it if it has been configured in.

Now all you have to do is write and debug your own new block device driver. I wish you the best of luck, and I hope that this whirlwind tour has given you a head start.

Other Resources

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

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.