LPI certification 102 (release 2) exam prep, Part 4

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Before you start

About this tutorial

Welcome to "USB, secure shell, and file sharing," the last of four tutorials designed to prepare you for the Linux Professional Institute's 102 exam. In this tutorial, we'll introduce you to the ins and outs of using USB devices, how to use the secure shell (ssh) and related tools, and how to use and configure Network File System (NFS) version 3 servers and clients.

This tutorial is ideal for those who want to learn about or improve their foundational Linux USB, networking, and file sharing skills. It is particularly appropriate for those who will be setting up applications or USB hardware on Linux servers or desktops. For many, much of this material will be new, but more experienced Linux users may find this tutorial to be a great way of rounding out their important Linux system administration skills. If you are new to Linux, we recommend you start with *Part 1* and work through the series from there.

By studying this series of tutorials (eight in all for the 101 and 102 exams; this is the eighth and last installment), you'll have the knowledge you need to become a Linux Systems Administrator and will be ready to attain an LPIC Level 1 certification (exams 101 and 102) from the Linux Professional Institute if you so choose.

For those who have taken the *release 1 version* of this tutorial for reasons other than LPI exam preparation, you probably don't need to take this one. However, if you do plan to take the exams, you should strongly consider reading this revised tutorial.

The LPI logo is a trademark of the *Linux Professional Institute*.

About the authors

For technical questions about the content of this tutorial, contact the authors:

- Daniel Robbins, at *drobbins@gentoo.org*
- John Davis, at *zhen@gentoo.org*

Daniel Robbins lives in Albuquerque, New Mexico, and is the Chief Architect of Gentoo Technologies, Inc., the creator of *Gentoo Linux*, an advanced Linux for the PC, and the Portage system, a next-generation ports system for Linux. He has also served as a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as to a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at Sony Electronic Publishing/Psygnosis. Daniel enjoys spending time with his wife, Mary, and their daughter, Hadassah.

John Davis lives in Cleveland, Ohio, and is the Senior Documentation Coordinator for *Gentoo Linux*, as well as a full-time computer science student at Mount Union College. Ever since his first dose of Linux at age 11, John has become a religious follower and has not looked back. When he is not writing, coding, or doing the college "thing," John can be found mountain biking or spending time with his family and close friends.

Section 2. USB devices and Linux

USB preliminaries

USB, or Universal Serial Bus, is a means of attaching devices and peripherals to your computer using cute little rectangular plugs. Commonly used by keyboards, mice, printers, and scanners, USB devices come in all shapes and sizes. One thing is certain: USB devices have arrived and it's essential to be able to get them running under Linux.

Setting up USB under GNU/Linux has always been a fairly easy, but undocumented, task. Users are often confused about whether or not to use modules, what the difference between UHCI, OHCI, and EHCI is, and why in the world their specific USB device is not working. This section should help clarify the different aspects of the Linux USB system.

This section assumes that you are familiar with how to compile your kernel, as well as the basic operation of a GNU/Linux system. For more information on these subjects, please visit the other LPI tutorials in this series, starting with *Part 1*, or The Linux Documentation Project homepage (see the Resources on page 17 at the end of this tutorial for links).

Modular vs. monolithic USB

Kernel support for USB devices can be configured in two ways: as modules or statically compiled into the kernel. Modular design allows for a smaller kernel size and quicker boot times. Statically compiled support allows for boot-time detection of devices and takes away the fuss of module dependencies. Both of these methods have their pros and cons, but the use of modules is suggested because they are easier to troubleshoot. Later, when everything is working, you may statically compile your USB device modules for convenience.

Grab a kernel

If you do not already have Linux kernel sources installed on your system, it is recommended that you download the latest 2.4 series kernel from kernel.org or one of its many mirrors (see the Resources on page 17 for a link).

Look at your hardware

Before compiling support for anything USB into your kernel, it is a good idea to find out what kind of hardware your computer is running. A simple, very useful set of tools called **pciutils** will get you on the right track. If you don't already have pciutils installed on your system, the sources for the most recent pciutils can be found at its homepage, which is listed in the Resources on page 17 .

Enter lspci

Running `lspci` should produce output similar to this:

```
# lspci
00:00.0 Host bridge: Advanced Micro Devices [AMD] AMD-760 [IGD4-1P] System Controller (r
00:01.0 PCI bridge: Advanced Micro Devices [AMD] AMD-760 [IGD4-1P] AGP Bridge
00:07.0 ISA bridge: VIA Technologies, Inc. VT82C686 [Apollo Super South] (rev 40)
00:07.1 IDE interface: VIA Technologies, Inc. VT82C586A/B/VT82C686/A/B/VT8233/A/C/VT8235
00:07.2 USB Controller: VIA Technologies, Inc. USB (rev 1a)
00:07.3 USB Controller: VIA Technologies, Inc. USB (rev 1a)
00:07.4 SMBus: VIA Technologies, Inc. VT82C686 [Apollo Super ACPI] (rev 40)
00:08.0 Serial controller: US Robotics/3Com 56K FaxModem Model 5610 (rev 01)
00:0b.0 VGA compatible controller: nVidia Corporation NV11DDR [GeForce2 MX 100 DDR/200 D
00:0d.0 Ethernet controller: 3Com Corporation 3c905C-TX/TX-M [Tornado] (rev 78)
00:0f.0 Multimedia audio controller: Creative Labs SB Live! EMU10k1 (rev 08)
00:0f.1 Input device controller: Creative Labs SB Live! MIDI/Game Port (rev 08)
01:05.0 VGA compatible controller: nVidia Corporation NV25 [GeForce4 Ti 4400] (rev a2)
```

Enable the right host controller

As you can see, `lspci` gives a complete listing of all PCI/PCI Bus Masters that your computer uses. The lines that are highlighted should be similar to what you are looking for in your lspci readout. Since the example controller is a VIA type controller, it would use the UHCI USB driver. For other chipsets, you would pick from one of these choices:

Driver	Chipset
EHCI	USB 2.0 Support
UHCI	All Intel, all VIA chipsets
JE (Alternate to UHCI	If UHCI does not work, and you have an Intel or VIA chipset, try JE
OHCI	Compaq, most PowerMacs, iMacs, and PowerBooks, OPTi, SiS, ALi

```
# cd /usr/src/linux
# make menuconfig
# make modules && make modules_install
```

Those cute USB modules

When the building completes, load the modules with `modprobe`, and your USB system will be ready to use.

The following line loads core USB support:

```
# modprobe usbcore
```

If you are using an EHCI controller, execute this line:

```
# modprobe ehci-hcd
```

If you are using an UHCI controller, execute this line:

```
# modprobe usb-uhci
```

If you are using a JE controller, execute this line:

```
# modprobe uhci
```

If you are using an OHCI controller, execute this line:

```
# modprobe usb-ohci
```

USB peripherals -- mice

Perhaps the most commonly used USB device is a USB mouse. Not only is it easy to install, but it offers plug-and-play flexibility for laptop users who would rather not use the homicide-inducing trackpad.

Before you can start using your USB mouse, you need to compile USB mouse support into your kernel. Enable these two options:

Menuconfig location	Option	Reason
Input Core Support	Mouse Support *(Don't forget to input your screen resolution!)*	Enabling this will hone your mouse tracking to your resolution, which makes mousing across large resolutions much nicer.
USB Support/ USB Human Interface Devices (HID)	USB HIDPB Mouse (basic) Support	Since your USB mouse is a USB HID (Human Interface Device), choosing this option will en the necessary HID subsystems.

USB mice, continued

Now, compile your new USB mouse-related modules:

```
# cd /usr/src/linux
# make menuconfig
# make modules && make modules_install
```

Once these options are compiled as modules, you are ready to load the `usbmouse` module and proceed:

```
# modprobe usbmouse
```

When the module finishes loading, go ahead and plug in your USB mouse. If you already had the mouse plugged in while the machine was booting, no worries, as it will still work.

Once you plug in the mouse, use `dmesg` to see if it was detected by the kernel:

```
# dmesg
hub.c: new USB device 10:19.0-1, assigned address 2
input4: USB HID v0.01 Mouse [Microsoft Microsoft IntelliMouse Optical] on usb2:2.0
```

When you have confirmed that your mouse is recognized by the kernel, it is time to configure XFree86 to use it. That's next.

USB mice and Xfree86

If you already have XFree86 installed and running with a non-USB mouse, not much configuration change is needed to use it. The only item that you need to change is what device XFree86 uses for your mouse. For our purposes, we will be using the /dev/input/mice device, since it supports hotplugging of your mouse, which can be very handy for desktop and laptop users alike. Your XF86Config file's "InputDevice" section should look similar to this:

```
Section "InputDevice"
   Identifier            "Mouse0"
   Driver               "mouse"
   Option        "Protocol"      "IMPS/2"
   #The next line enables mouse wheel support
   Option             "ZAxisMapping"  "4 5"
   #The next line points XFree86 to the USB mouse device
   Option             "Device"        "/dev/input/mice"
EndSection
```

Now, restart XFree86, and your USB mouse should be working just fine. Once everything is working, go ahead and compile your USB modules into the kernel statically. Of course, this is completely optional, so if you would like to keep your modules as modules, make sure they are loaded at boot time so that you can use your mouse after you reboot.

Configuring a USB digital camera

Yet another great feature in GNU/Linux is its digital imaging support. Powerful photo editing programs, such as the GIMP (see the Resources on page 17 for a link), make digital photography come alive.

Before any digital picture editing can take place, you'll need to retrieve the pictures that are going to be edited. Many times, digital cameras will have a USB port, but if yours does not, these instructions will work for your media card reader as long as the file system on your media card is supported in the Linux kernel.

USB Mass Storage works for anything that uses USB to access an internal drive of some sort. Feel free to experiment with other hardware, such as USB MP3 players, as these instructions will work the same. Additionally, note that older cameras with built-in serial ports are not compatible with these instructions.

USB storage -- the modules

Most USB Mass Storage devices use SCSI emulation so that they can be accessed by the

Linux kernel. Therefore, kernel support must be enabled for SCSI support, SCSI disk support, SCSI generic support, and USB Mass Storage support.

```
# cd /usr/src/linux
# make menuconfig
```

Enable the following options:

Menuconfig location	Option	Reason
SCSI support	SCSI support	Enables basic SCSI support
SCSI support	SCSI disk support	Enables support for SCSI disks
SCSI support	SCSI generic support	Enables support for generic SCSI devices, as well as some emulation
USB support/ USB Device Class drivers	USB Mass Storage support	Enables basic USB Mass Storage support; be sure to enable the options listed below it if you need support for any of that hardware

Build the USB storage modules

Since the options in the previous screen were compiled into your kernel as modules, there is no need to rebuild your kernel or reboot your computer! We just need to remake modules, and then load the newly compiled modules using `modprobe`.

```
# make modules && make modules_install
```

Please note that your third-party modules, such as NVIDIA drivers and ALSA drivers, may be overwritten by the module installation. You might want to reinstall those right after running `make modules_install`.

Did it work?

Once your modules are rebuilt, plug in your camera or media card reader and load the USB Mass Storage module:

The following line loads the SCSI disk support module:

```
# modprobe sd_mod
```

The following line loads the USB Mass Storage support module:

```
# modprobe usb-storage
```

Running `dmesg` should produce output similar to this:

```
# dmesg
Initializing USB Mass Storage driver...
usb.c: registered new driver usb-storage
scsi1 : SCSI emulation for USB Mass Storage devices
  Vendor: SanDisk   Model: ImageMate CF-SM   Rev: 0100
  Type:   Direct-Access                      ANSI SCSI revision: 02
  Vendor: SanDisk   Model: ImageMate CF-SM   Rev: 0100
  Type:   Direct-Access                      ANSI SCSI revision: 02
WARNING: USB Mass Storage data integrity not assured
USB Mass Storage device found at 2
USB Mass Storage support registered.
```

USB storage is go!

Congratulations! If you see something like the output in the previous screen, you're in business. All you have left to do is mount the camera or media card reader, and you can directly access your pictures.

On our machine, the card reader was mapped to /dev/sda1; yours might be different.

To mount your device, do the following (and note that your media's file system might not be vfat, so substitute as needed):

```
# mkdir /mnt/usb-storage
# mount -t vfat /dev/sda1 /mnt/usb-storage
```

Section 3. Secure shell

Interactive logins

Back in the old days, if you wanted to establish an interactive login session over the network, you used `telnet` or `rsh`. However, as networking became more popular, these tools became less and less appropriate. Why? Because they're horrendously insecure: the data going between the telnet client and server isn't encrypted, and can thus be read by anyone snooping the network.

Not only that, but *authentication* (the sending of your password to the server) is performed in plain text, making it a trivial matter for someone capturing your network data to get instant access to your password. In fact, using a network sniffer it's possible for someone to reconstruct your entire telnet session, seeing everything on the screen that you see! Obviously, tools such as telnet were designed with the assumption that the network was secure and unsniffable and are inappropriate for today's distributed and public networks.

Secure shell

A better solution was needed, and that solution came in the form of a tool called `ssh`. A popular modern incarnation of this tool is available in the openssh package, available for virtually every Linux distribution, not to mention many other systems.

What sets `ssh` apart from its insecure cousins is that it *encrypts* all communications between the client and the server using strong encryption. By doing this, it becomes very difficult or impossible to monitor the communications between the client and server. In this way, `ssh` provides its service as advertised -- it is a *secure* shell. In fact, `ssh` has excellent "all-around" security -- even authentication takes advantage of encryption and various key exchange strategies to ensure that the user's password cannot be easily grabbed by anyone monitoring data being transmitted over the network.

In this age of Internet popularity, `ssh` is a valuable tool for enhancing network security when using Linux systems. Most security-savvy network admins discourage or disallow the use of `telnet` and `rsh` on their systems because `ssh` is such a capable and secure replacement.

Using ssh

Generally, most distributions' openssh packages can be used without any manual configuration. After installing openssh, you'll have a couple of binaries. One is, of course, `ssh`, the secure shell client that can be used to connect to any system running `sshd`, the secure shell server. To use `ssh`, you typically start a session by typing something like:

```
$ ssh drobbins@remotebox
```

Above, you instruct `ssh` to log in as the "drobbins" user account on remotebox. Like `telnet`, you'll be prompted for a password; after entering it, you'll be presented with a new login session on the remote system.

Starting sshd

If you want to allow `ssh` connections to your machine, you'll need to start the `sshd` server. To start the `sshd` server, you would typically use the rc-script that came with your distribution's openssh package by typing something like:

```
# /etc/init.d/sshd start
```

or

```
# /etc/rc.d/init.d/sshd start
```

If necessary, you can adjust configuration options for `sshd` by modifying the /etc/ssh/sshd_config file. For more information on the various options available, type `man sshd`.

Secure copy

The openssh package also comes with a handy tool called `scp` (secure copy). You can use this command to securely copy files to and from various systems on the network. For example, if you wanted to copy ~/foo.txt to our home directory on remotebox, you could type:

```
$ scp ~/foo.txt drobbins@remotebox:
```

Note the trailing colon -- without it, `scp` would have created a local file in the current working directory called "drobbins@remotebox." However, with the colon, the intended action is taken. After being prompted for the password on remotebox, the copy will be performed.

If you wanted to copy a file called bar.txt in remotebox's /tmp directory to the current working directory on our local system, you could type:

```
$ scp drobbins@remotebox:/tmp/bar.txt .
```

Again, the ever-important colon separates the user and host name from the file path.

Secure shell authentication options

Openssh also has a number of other authentication methods. Used properly, they can let you authenticate with remote systems without having to type in a password or passphrase for every connection. To learn more about how to do this, read Daniel's openssh key management articles on *developerWorks* (see the links in the Resources on page 17 at the end of this tutorial).

Section 4. NFS

Introducing NFS

The Network File System (NFS) is a technology that allows the transparent sharing of files between UNIX and Linux systems connected via a Local Area Network, or LAN. NFS has been around for a long time; it's well known and used extensively in the Linux and UNIX worlds. In particular, NFS is often used to share home directories among many machines on the network, providing a consistent environment for a user when he or she logs in to a machine (*any* machine) on the LAN. Thanks to NFS, it's possible to mount remote file system trees and have them fully integrated into a system's local file system. NFS' transparency and maturity make it a useful and popular choice for network file sharing under Linux.

NFS basics

To share files using NFS, you first need to set up an NFS server. This NFS server can then "export" file systems. When a file system is exported, it is made available for access by other systems on the LAN. Then, any authorized system that is also set up as an NFS client can mount this exported file system using the standard `mount` command. After the mount completes, the remote file system is "grafted in" in the same way that a locally mounted file system (such as /mnt/cdrom) would be after it is mounted. The fact that all of the file data is being read from the NFS server rather than from a disk is not an issue to any standard Linux application. Everything simply works.

Attributes of NFS

Shared NFS file systems have a number of interesting attributes. The first is a result of NFS' stateless design. Because client access to the NFS server is stateless in nature, it's possible for the NFS server to reboot without causing client applications to crash or fail. All access to remote NFS files will simply "pause" until the server comes back online. Also, because of NFS' stateless design, NFS servers can handle large numbers of clients without any additional overhead besides that of transferring the actual file data over the network. In other words, NFS performance is dependent on the amount of NFS data being transferred over the network, rather than on the number of machines that happen to be requesting that data.

NFS version 3 under Linux

When you set up NFS, we recommend that you use NFS version 3 rather than version 2. Version 2 has some significant problems with file locking and generally has a bad reputation for breaking certain applications. On the other hand, NFS version 3 is very nice and robust and does its job well. Now that Linux 2.2.18+ supports NFS 3 clients and servers, there's no reason to consider using NFS 2 anymore.

Securing NFS

It's important to mention that NFS version 2 and 3 have some very clear security limitations. They were designed to be used in a specific environment: a secure, trusted LAN. In

particular, NFS 2 and 3 were designed to be used on a LAN where "root" access to the machine is only allowed by administrators. Due to the design of NFS 2 and NFS 3, if a malicious user has "root" access to a machine on your LAN, he or she will be able to bypass NFS security and very likely be able to access or even modify files on the NFS server that he or she wouldn't normally be able to otherwise. For this reason, NFS should not be deployed casually. If you're going to use NFS on your LAN, great -- but set up a firewall first. Make sure that people outside your LAN won't be able to access your NFS server. Then, make sure that your internal LAN is relatively secure and that you are fully aware of all the hosts participating in your LAN.

Once your LAN's security has been thoroughly reviewed and (if necessary) improved, you're ready to safely use NFS (see *Part 3* of the 102 series for more on this).

NFS users and groups

When setting up NFS, it's important to ensure that all NFS machines (both servers and clients) have identical user and group IDs in their user and group databases. Why? Because NFS clients and servers use numeric user and group IDs internally and assume that the IDs correspond to the same users and groups on all NFS-enabled machines.

Because of this, using mismatched user and group IDs with NFS can result in security breaches -- particularly if two different users on different systems happen to be sharing the same numerical UID.

So, before getting NFS set up on a larger LAN, it's a good idea to first set up NIS or NIS+. NIS(+), which stands for "Network Information Service," allows you to have a user and group database that can be centrally managed and shared throughout your entire LAN, ensuring NFS ownership consistency as well as reducing administrative headaches.

NIS and NFS combined

When NIS+ and NFS are combined, you can configure Linux systems on your network so that your users can log in to any box on your LAN -- and access their home directories and files from that box. NIS+ provides the shared user database that allows a user to log in anywhere, and NFS delivers the data. Both technologies work very well together.

While NIS+ is important, we don't have room to cover it in this tutorial. If you are planning to take the LPI exams -- or want to be able to use NFS to its full potential -- be sure to study the "Linux NFS HOWTO" by Thorsten Kukuk (see the Resources on page 17 for a link).

Setting up NFS under Linux

The first step in using NFS 3 is to set up an NFS 3 server. Choose the system that will be serving files to the rest of your LAN. On our machine, we'll need to enable NFS server support in the kernel. You should use a 2.2.18+ kernel (2.4+ recommended) to take advantage of NFS 3, which is much more stable than NFS 2. If you're compiling your own custom kernel, enter your /usr/src/linux directory and run `make menuconfig`. Then select the "File Systems" section, then the "Network File Systems" section, and ensure that the following options are enabled:

```
<*> NFS file system support
[*]   Provide NFSv3 client support
<*> NFS server support
[*]   Provide NFSv3 server support
```

Getting ready for /etc/exports

Next, compile and install your new kernel and reboot. Your system will now have NFS 3 server and client support built in.

Now that our NFS server has support for NFS in the kernel, it's time to set up an /etc/exports file. The /etc/exports file will describe the local file systems that will be made available for export as well as:

- What hosts will be able to access these file systems
- Whether they will be exported as read/write or read-only
- Other options that control NFS behavior

But before we look at the format of the /etc/exports file, a big implementation warning is needed! The NFS implementation in the Linux kernel only allows the export of *one local directory per file system*. This means that if both /usr and /home are on the same ext3 file system (on /dev/hda6, for example), then you can't have both /usr and /home export lines in /etc/exports. If you try to add these lines, you'll see error like this when your /etc/exports file gets reread (which will happen if you type `exportfs -ra` after your NFS server is up and running):

```
sidekick:/home: Invalid argument
```

Working around export restrictions

Here's how to work around this problem. If /home and /usr are on the same underlying local file system, you can't export them both, so just export /. NFS clients will then be able to mount /home and /usr via NFS just fine, but your NFS server's /etc/exports file will now be "legal," containing only one export line per underlying local file system. Now that you understand this implementation quirk of Linux NFS, let's look at the format of /etc/exports.

The /etc/exports file

Probably the best way to understand the format of /etc/exports is to look at a quick example. Here's a simple /etc/exports file that we use on our NFS server:

```
# /etc/exports: NFS file systems being exported. See exports(5).
/ 192.168.1.9(rw,no_root_squash)
/mnt/backup 192.168.1.9(rw,no_root_squash)
```

As you can see, the first line in the /etc/exports file is a comment. On the second line, we select our root ("/") file system for export. Note that while this exports everything under "/", it

will not export any other local file system. For example, if our NFS server has a CD-ROM mounted at /mnt/cdrom, the contents of the CDROM will not be available unless they are exported explicitly in /etc/exports. Now, notice the third line in our /etc/exports file. On this line, we export /mnt/backup; as you might guess, /mnt/backup is on a separate file system from /, and it contains a backup of our system.

Each line also has a "192.168.1.9(rw,no_root_squash)" on it. This information tells nfsd to only make these exports available to the NFS client with the IP address of 192.168.1.9. It also tells nfsd to make these file systems writeable as well as readable by NFS client systems ("rw",) and instructs the NFS server to allow the remote NFS client to allow a superuser account to have true "root" access to the file systems ("no_root_squash".)

Another /etc/exports file

Here's an /etc/exports that will export the same file systems as the one in the previous panel, except that it will make our exports available to all machines on our LAN -- 192.168.1.1 through 192.168.1.254:

```
# /etc/exports: NFS file systems being exported. See exports(5).
/ 192.168.1.1/24(rw,no_root_squash)
/mnt/backup 192.168.1.1/24(rw,no_root_squash)
```

In the above example /etc/exports file, we use a host mask of /24 to mask out the last eight bits in the IP address we specify. It's very important that there is no space between the IP address specification and the "(", or NFS will interpret your information incorrectly. And, as you might guess, there are other options that you can specify besides "rw" and "no_root_squash"; type "man exports" for a complete list.

Starting the NFS 3 server

Once /etc/exports is configured, you're ready to start your NFS server. Most distributions will have an "nfs" initialization script that you can use to start NFS -- type `/etc/init.d/nfs start` or `/etc/rc.d/init.d/nfs start` to use it -- or use "restart" instead of "start" if your NFS server was already started at boot-time. Once NFS is started, typing `rpcinfo` should display output that looks something like this:

```
# rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100024    1   udp  32802  status
    100024    1   tcp  46049  status
    100011    1   udp    998  rquotad
    100011    2   udp    998  rquotad
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
    100003    2   tcp   2049  nfs
    100003    3   tcp   2049  nfs
    100021    1   udp  32804  nlockmgr
    100021    3   udp  32804  nlockmgr
    100021    4   udp  32804  nlockmgr
    100021    1   tcp  48026  nlockmgr
```

```
    100021    3   tcp  48026  nlockmgr
    100021    4   tcp  48026  nlockmgr
    100005    1   udp  32805  mountd
    100005    1   tcp  39293  mountd
    100005    2   udp  32805  mountd
    100005    2   tcp  39293  mountd
    100005    3   udp  32805  mountd
    100005    3   tcp  39293  mountd
```

Changing export options

If you ever change your /etc/exports file while your NFS daemons are running, simply type `exportfs -ra` to apply your changes. Now that your NFS server is up and running, you're ready to configure NFS clients so that they can mount your exported file systems.

Configuring NFS clients

Kernel configuration for NFS 3 clients is similar to that of the NFS server, except that you only need to ensure that the following options are enabled:

```
<*> NFS file system support
[*]  Provide NFSv3 client support
```

Starting NFS client services

To start the appropriate NFS client daemons, you can typically use a system initialization script called "nfslock" or "nfsmount." Typically, this script will start rpc.statd, which is all the NFS 3 client needs -- rpc.statd allows file locking to work properly. Once all your client services are set up, running `rpcinfo` on the local machine will display output that looks like this:

```
# rpcinfo
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100024    1   udp  32768  status
    100024    1   tcp  32768  status
```

You can also perform this check from a remote system by typing `rpcinfo -p myhost`, as follows:

```
# rpcinfo -p sidekick
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100024    1   udp  32768  status
    100024    1   tcp  32768  status
```

Mounting exported NFS file systems

Once both client and server are set up correctly (and assuming that the NFS server is configured to allow connections from the client), you can go ahead and mount an exported NFS file system on the client. In this example, "inventor" is the NFS server and "sidekick" (IP address 192.168.1.9) is the NFS client. Inventor's /etc/exports file contains a line that looks like this, allowing connections from any machine on the 192.168.1 network:

```
/ 192.168.1.1/24(rw,no_root_squash)
```

Now, logged into sidekick as root, you can type:

```
# mount inventor:/ /mnt/nfs
```

Inventor's root file system will now be mounted on sidekick at /mnt/nfs; you should now be able to type `cd /mnt/nfs` and look around inside and see inventor's files. Again, note that if inventor's /home tree is on another file system, then /mnt/nfs/home will not contain anything -- another `mount` (as well as another entry in inventor's /etc/exports file) will be required to access that data.

Mounting directories *inside* exports

Note that inventor's `/ 192.168.1.1/24(rw,no_root_squash)` line will also allow us to mount directories *inside* /. For example, if inventor's /usr is on the same physical file system as /, and you are only interested in mounting inventor's /usr on sidekick, you could type:

```
# mount inventor:/usr /mnt/usr
```

Inventor's /usr tree will now be NFS mounted to the pre-existing /mnt/usr directory. It's important to again note that inventor's /etc/exports file didn't need to explicitly export /usr; it was included "for free" in our "/" export line.

Section 5. Summary and resources

Summary

This wraps up this tutorial and the LPI 102 series. We hope you've enjoyed the ride! You should now be well versed on the use of ssh, NFS, and USB. To expand your Linux knowledge even further, see the Resources on page 17 on the next panel.

Resources

Although the tutorial is over, learning never ends and we recommend you check out the following resources, particularly if you plan to take the LPI 102 exam:

For more information on USB under GNU/Linux, please check out the *official Linux USB project page* for more information.

If you do not have *pciutils* installed on your system, you can find the source at the *pciutils project homepage*.

Get more information on *XFree86 configuration* at Xfree86.org.

Visit the *project homepage for the venerable GIMP*, or GNU Image Manipulation Program.

Daniel's OpenSSH key management series of articles on *developerWorks* is a great way to gain a deeper understanding of the security features provided by OpenSSH:

- *Part 1 on RSA/DSA authentication*
- *Part 2 on ssh-agent and keychain*
- *Part 3 on agent forwarding and keychain improvements*

Also be sure to visit the *home of openssh*, which is an excellent place to continue your study of this important tool.

The best thing you can do to improve your NFS skills is to try setting up your own NFS 3 server and client(s) -- the experience will be invaluable. The second-best thing you can do is to read the quite good *Linux NFS HOWTO*, by Thorsten Kukuk.

We didn't have room to cover another important networked file-sharing technology: Samba. For more information about Samba, we recommend that you read Daniel's Samba articles on *developerWorks*:

- *Part 1 on key concepts*
- *Part 2 on compiling and installing Samba*
- *Part 3 on Samba configuration*

Once you're up to speed on Samba, we recommend that you spend some time studying the *Linux DNS HOWTO*. The LPI 102 exam is also going to expect that you have some familiarity with Sendmail. We didn't have enough room to cover Sendmail, but (fortunately for us!) Red Hat has a good *Sendmail HOWTO* that will help to get you up to speed.

In addition, we recommend the following general resources for learning more about Linux and preparing for LPI certification in particular:

Linux kernels and more can be found at the *Linux Kernel Archives*.

You'll find a wealth of guides, HOWTOs, FAQs, and man pages at *The Linux Documentation Project*. Be sure to check out *Linux Gazette* and *LinuxFocus* as well.

The Linux System Administrators guide, available from *Linuxdoc.org's "Guides" section*, is a good complement to this series of tutorials -- give it a read! You may also find Eric S. Raymond's *Unix and Internet Fundamentals HOWTO* to be helpful.

In the *Bash by example* article series on *developerWorks*, learn how to use `bash` programming constructs to write your own `bash` scripts. This series (particularly parts 1 and 2) are excellent additional preparation for the LPI exam:

- *Part 1 on fundamental programming in the Bourne-again shell*
- *Part 2 on more bash programming fundamentals*
- *Part 3 on the ebuild system*

The *Technical FAQ for Linux Users* by Mark Chapman is a 50-page in-depth list of frequently-asked Linux questions, along with detailed answers. The FAQ itself is in PDF (Acrobat) format. If you're a beginning or intermediate Linux user, you really owe it to yourself to check this FAQ out. The *Linux glossary for Linux users*, also from Mark, is excellent as well.

If you're not very familiar with the vi editor, you should check out Daniel's *tutorial on vi*. This *developerWorks* tutorial will give you a gentle yet fast-paced introduction to this powerful text editor. Consider this must-read material if you don't know how to use vi.

For more information on the Linux Professional Institute, visit the *LPI home page*.

Feedback

Please send any tutorial feedback you may have to the authors:

- Daniel Robbins, at *drobbins@gentoo.org*
- John Davis, at *zhen@gentoo.org*

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.