



Linux Basics and Installation

(Course code LX02)

Student Notebook

ERC 7.2

Authorized

IBM | Training

Trademarks

IBM® and the IBM logo are registered trademarks of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide:

AIX®

System i®

System p®

System x®

System z®

U®

PostScript is either a registered trademark or a trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

VMware and the VMware "boxes" logo and design, Virtual SMP and VMotion are registered trademarks or trademarks (the "Marks") of VMware, Inc. in the United States and/or other jurisdictions.

Other product and service names might be trademarks of IBM or other companies.

January 2012 edition

The information contained in this document has not been submitted to any formal IBM test and is distributed on an "as is" basis without any warranty either express or implied. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will result elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Contents

Trademarks	xi
Course description	xiii
Agenda	xv
Unit 1. Introduction to Linux.	1-1
Unit objectives	1-2
A short history of Linux (1 of 2)	1-3
A short history of Linux (2 of 2)	1-5
What's so special about Linux (GNU/Linux)?	1-7
Effects of the license model	1-9
Linux has become a way of life	1-11
Linux today	1-13
Linux hardware support	1-15
Looking at Linux	1-17
Unit review	1-19
Checkpoint	1-20
Unit summary	1-21
Unit 2. Installing Linux	2-1
Unit objectives	2-2
Preparing a system for installation	2-3
Installing Linux	2-4
Network installations	2-5
Installation steps	2-7
Select language, keyboard, and mouse	2-8
Install class	2-9
Disk partitioning	2-10
Configure a boot loader	2-12
Configure network	2-14
Configure root and user accounts	2-15
Select package groups	2-16
Configure X	2-17
Other (optional) installation screens	2-18
Installing packages	2-19
Post-install configuration	2-20
Checkpoint (1 of 2)	2-21
Checkpoint (2 of 2)	2-22
Exercise: Linux installation	2-23
Unit summary	2-24
Unit 3. Using the system	3-1
Unit objectives	3-2

Linux is multiuser and multitasking	3-3
Virtual terminals	3-4
Logging in: Text mode VT	3-5
Logging in: Graphical mode VT	3-6
Linux commands	3-7
Starting a terminal emulator	3-8
Command prompt	3-9
Linux command syntax	3-10
Command format examples	3-11
Some basic Linux commands	3-12
Changing your password	3-13
The date command	3-15
The cal command	3-16
Who is on the system	3-17
Finding information about users	3-18
The clear, echo, write, and wall commands	3-19
Talk with another user	3-20
Keyboard tips	3-21
Locking	3-23
Logging out	3-24
Unit review	3-25
Checkpoint	3-26
Exercise: Using the system	3-27
Unit summary	3-28
 Unit 4. Working with files and directories	 4-1
Unit objectives	4-2
A file	4-3
File types	4-4
Linux file names	4-5
Directory structure	4-6
/bin, /lib, and /sbin	4-7
/boot, /dev, and /etc	4-8
/home, /root, and /tmp	4-9
/proc and /sys	4-10
/usr and /var	4-11
Other directories in /	4-12
Typical file system layout	4-13
Example directory structure	4-14
Linux path names	4-15
Where am I?	4-16
Change current directory	4-17
Create directories	4-18
Removing directories	4-19
Working with multiple directories	4-20
List the contents of directories	4-21
The touch command	4-22
Copying files (1 of 2)	4-23

Copying files (2 of 2)	4-24
Moving and renaming files (1 of 2)	4-25
Moving and renaming files (2 of 2)	4-26
Removing files	4-27
Listing file contents	4-28
Displaying files page by page	4-29
Displaying binary files	4-30
File managers	4-31
Virtual unified file system (1 of 2)	4-32
Virtual unified file system (2 of 2)	4-33
The mount command	4-35
The umount command	4-36
The /etc/fstab file	4-37
Mounting and unmounting removable media	4-38
Hard links and soft (symbolic) links	4-39
Unit review	4-41
Checkpoint	4-42
Exercise: Working with files and directories	4-43
Unit summary	4-44
Unit 5. File and directory permissions	5-1
Unit objectives	5-2
Users and groups	5-3
Permissions	5-4
Viewing permissions (command line)	5-5
Permissions notation	5-6
Required permissions	5-7
Who can change permissions?	5-8
Changing permissions (1 of 2)	5-9
Changing permissions (2 of 2)	5-11
umask	5-12
Access Control Lists	5-13
Permissions in the GUI	5-15
Unit review	5-16
Checkpoint	5-17
Exercise: File and directory permissions	5-18
Unit summary	5-19
Unit 6. Linux documentation	6-1
Unit objectives	6-2
The man command	6-3
man example (1 of 2)	6-4
man example (2 of 2)	6-5
man sections	6-6
The info command	6-7
info example	6-9
The --help option	6-10
HOWTO documents	6-11

HOWTO example	6-12
Other documentation	6-13
Internet	6-14
Unit review	6-15
Checkpoint	6-16
Exercise: Linux documentation	6-17
Unit summary	6-18
 Unit 7. Editing files	 7-1
Unit objectives	7-2
Determining file content	7-3
The Vi text editor	7-5
Vi modes	7-7
Starting Vi	7-8
Cursor movement in command mode	7-9
Editing text in command mode	7-10
Switching to edit mode	7-11
Adding text in edit mode	7-12
Exiting the edit mode	7-13
Searching for patterns	7-14
Replacing patterns	7-15
Cut, copy, and paste	7-17
Cut and paste	7-18
Copy and paste	7-19
Vi options	7-20
Exiting Vi	7-22
Vi cheat sheet	7-23
Other editors	7-24
Unit review	7-25
Checkpoint	7-26
Exercise: Editing files	7-27
Unit summary	7-28
 Unit 8. Shell basics	 8-1
Unit objectives	8-2
The shell	8-3
Shell features	8-4
Metacharacters and reserved words	8-5
Basic wildcard expansion	8-6
Advanced wildcard expansion	8-7
File descriptors	8-8
Input redirection	8-9
Output redirection	8-10
Error redirection	8-11
Combined redirection	8-12
Pipes	8-13
Filters	8-14
Common filters	8-15

Split output	8-17
Command substitution	8-18
Command grouping	8-20
Shell variables	8-21
Referencing shell variables	8-22
Exporting shell variables	8-23
Standard shell variables	8-24
Return codes from commands	8-26
Quoting metacharacters	8-27
Quoting non-metacharacters	8-28
Aliases	8-29
Unit review	8-30
Checkpoint	8-31
Exercise: Shell basics	8-32
Unit summary	8-33
Unit 9. Working with processes.....	9-1
Unit objectives	9-2
What is a process?	9-3
Starting and stopping a process	9-4
Login process environment	9-5
Parents and children	9-6
Monitoring processes	9-7
Viewing process hierarchy	9-8
Controlling processes	9-9
Starting processes	9-10
Job control in the bash shell	9-11
Job control example	9-12
Kill signals	9-13
Running long processes	9-15
Managing process priorities	9-16
The nice command	9-18
The renice command	9-19
Integrated process management	9-20
Daemons	9-21
Unit review	9-22
Checkpoint	9-23
Exercise: Working with processes	9-24
Unit summary	9-25
Unit 10. Linux utilities	10-1
Unit objectives	10-2
The find command	10-3
Sample directory structure	10-4
Using find	10-5
Executing commands with find	10-6
Interactive command execution	10-7
Additional find options	10-8

find examples	10-9
locate command	10-10
The grep command	10-11
grep sample data files	10-12
Basic grep	10-13
grep with regular expressions	10-14
Regular expression	10-15
grep options	10-16
Other greps	10-17
The cut command	10-18
cut example (1 of 2)	10-19
cut example (2 of 2)	10-20
The sort command	10-21
sort examples	10-22
The head and tail commands	10-23
The type, which, and whereis commands	10-24
The file command	10-25
The join and paste commands	10-26
Compressing and uncompressing Files	10-27
Unit review	10-29
Checkpoint	10-30
Exercise: Linux utilities	10-31
Unit summary	10-32

Unit 11. Shell scripting	11-1
Unit objectives	11-2
What is a shell script?	11-3
Invoking shell scripts (1 of 3)	11-4
Invoking shell scripts (2 of 3)	11-5
Invoking shell scripts (3 of 3)	11-6
Invoking shell scripts in another shell	11-7
Typical shell script contents	11-8
Shell script arguments	11-9
Complex redirection	11-10
Conditional execution	11-11
The test command (1 of 2)	11-12
The test command (2 of 2)	11-13
The && and commands	11-14
The if command	11-15
Command repetition	11-16
The while command	11-17
The for command	11-18
Shifting shell script arguments	11-19
User interaction: The read command	11-21
Arithmetic using let and \$()	11-22
Arithmetic using expr	11-24
Command search order	11-25
Unit review	11-26

Checkpoint (1 of 2)	11-27
Checkpoint (2 of 2)	11-28
Exercise: Shell scripting	11-29
Unit summary	11-30
Unit 12. The Linux GUI	12-1
Unit objectives	12-2
The Linux graphical user interface	12-3
Client/server architecture	12-4
X components	12-5
Desktop environments	12-6
The KDE	12-7
The GNOME desktop environment	12-8
Starting X	12-9
Choosing your desktop environment	12-10
Unit review	12-11
Checkpoint	12-12
Exercise: The Linux GUI	12-13
Unit summary	12-14
Unit 13. Customizing the user environment	13-1
Unit objectives	13-2
Bash initialization: Login shell	13-3
Bash initialization: Non-login shell	13-4
Bash initialization with Red Hat extensions	13-5
Bash initialization with SLES extensions	13-6
KDE customization	13-7
GNOME customization	13-8
Unit review	13-9
Checkpoint	13-10
Exercise: Customizing the user environment	13-11
Unit summary	13-12
Unit 14. Basic system configuration	14-1
Unit objectives	14-2
Why system configuration?	14-3
System configuration tools	14-5
Adding or removing software using RPM	14-7
Querying the RPM database	14-9
Adding or removing software from a .tar.gz file	14-10
Other Linux software installers	14-11
Printer configuration	14-12
Sound card configuration	14-14
Network configuration	14-15
NetworkManager	14-16
Unit review	14-18
Checkpoint	14-19
Exercise: Basic system configuration	14-20

Unit summary	14-21
Unit 15. Integrating Linux in a Windows environment	15-1
Unit objectives	15-2
Differences in file systems	15-3
Mounting Windows file systems	15-5
Accessing Windows file systems directly	15-6
Running Windows programs	15-8
Emulators and virtual machines	15-9
VMWare screenshot	15-11
Windows emulators	15-12
WINE screenshot	15-13
Accessing Windows servers	15-14
smbclient examples	15-15
Mount.cifs examples	15-16
Reading Windows document formats	15-17
Other useful programs	15-18
Unit review	15-19
Checkpoint	15-20
Exercise: Integrating Linux in a Windows environment	15-21
Unit summary	15-22
Appendix A. Checkpoint solutions	A-1
Appendix B. List of abbreviations	B-1

Trademarks

The reader should recognize that the following terms, which appear in the content of this training document, are official trademarks of IBM or other companies:

IBM® and the IBM logo are registered trademarks of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide:

AIX®

System i®

System p®

System x®

System z®

U®

PostScript is either a registered trademark or a trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

VMware and the VMware "boxes" logo and design, Virtual SMP and VMotion are registered trademarks or trademarks (the "Marks") of VMware, Inc. in the United States and/or other jurisdictions.

Other product and service names might be trademarks of IBM or other companies.

Course description

Linux Basics and Installation

Duration: 4 days

Purpose

The objective of the course is to teach students enough about Linux to successfully install, configure, and run Linux on the student's personal workstation and be productive with it.

Audience

This course is designed for students with little or no Linux knowledge or experience who want to make a start with Linux.

Prerequisites

Students should be able to use a Microsoft Windows-based workstation.

Objectives

After completing this course, you should be able to:

- Install and configure Linux on a workstation
- Use Linux for daily work

Contents

- Introduction to Linux
- Installing Linux
- Using the system
- Working with files and directories
- Linux documentation
- Editing files
- Working with processes
- Shell basics
- Linux utilities
- Shell scripting

- The Linux GUI
- Customizing the user environment
- Basic system configuration
- Connecting to the Internet
- Integrating Linux in a Windows environment

Agenda

Day 1

- Unit 1: Introduction to Linux
- Unit 2: Installing Linux
- Exercise 1: Linux installation
- Unit 3: Using the system
- Exercise 2: Using the system
- Unit 4: Working with files and directories
- Exercise 3: Working with files and directories

Day 2

- Unit 5: File and directory permissions
- Exercise 4: File and directory permissions
- Unit 6: Linux documentation
- Exercise 5: Linux documentation
- Unit 7: Editing files
- Exercise 6: Editing files
- Unit 8: Shell basics
- Exercise 7: Shell basics

Day 3

- Unit 9: Working with processes
- Exercise 8: Working with processes
- Unit 10: Linux utilities
- Exercise 9: Linux utilities
- Unit 11: Shell scripting
- Exercise 10: Shell scripting
- Unit 12: The Linux GUI
- Exercise 11: The Linux GUI

Day 4

- Unit 13: Customizing the user environment
- Exercise 12: Customizing the user environment
- Unit 14: Basic system configuration
- Exercise 13: Basic system configuration
- Unit 15: Integrating Linux in a Windows environment
- Exercise 14: Integrating Linux in a Windows environment

Unit 1. Introduction to Linux

What this unit is about

This unit covers the history of Linux and names some important people involved in the history of Linux. This unit also discusses the GNU general public license.

What you should be able to do

After completing this unit, you should be able to:

- Discuss the history of Linux
- Name some important people in the history of Linux
- Discuss the GNU general public license

How you will check your progress

- Checkpoint questions

Unit objectives

After completing this unit, you should be able to:

- Discuss the history of Linux
- Name some important people in the history of Linux
- Discuss the GNU general public license

© Copyright IBM Corporation 2012

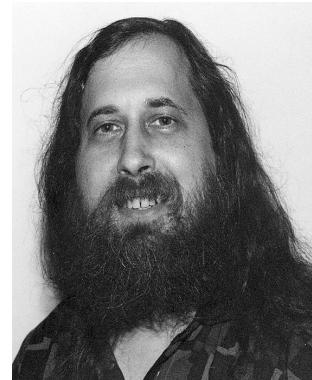
Figure 1-1. Unit objectives

LX027.2

Notes:

A short history of Linux (1 of 2)

- 1984: Richard Stallman starts GNU project
 - GNU's not UNIX
 - <http://www.gnu.org>
- Purpose: Free UNIX
 - "Free as in free speech, not free beer"
- First step: Re-implementation of UNIX utilities
 - C compiler, C library
 - emacs
 - bash
- To fund the GNU project, the Free Software Foundation founded
 - <http://www.fsf.org>



© Copyright IBM Corporation 2012

Figure 1-2. A short history of Linux (1 of 2)

LX027.2

Notes:

The history of Linux starts properly in 1984. In that year, a system administrator working at Massachusetts Institute of Technology (MIT), Richard Stallman, received a new version of the UNIX flavor they were using. But in contrast to previous versions, this time they did not receive the source of the operating system with it, and could not obtain the source separately without signing a Non-Disclosure Agreement (NDA). Richard Stallman was therefore not able to implement a certain additional feature into the operating system, which his users had come to like.

Richard Stallman became so upset with these developments in general that he vowed to write a new UNIX-like operating system from scratch. That new operating system was supposed to be free (as in free speech): Everybody would have the right to use and adapt the software for its own use, and to distribute the software to others. (More about this later.) This project was called GNU, which stands for GNU's not UNIX.

To fund the GNU project and to advocate the use of free software in general, the Free Software Foundation (FSF) was founded.

The first steps taken by the GNU project was to re-implement various essential utilities in a UNIX operating system. Although hundreds of little tools were written, four tools stand out:

- The GNU C Compiler (GCC) was essential for compiling all software, including the kernel and the C compiler itself.
- The GNU C Library (GLIBC) implements a large set of standardized system calls.
- Emacs is a full-featured, world-class editor which can be extended into a sort of application development environment.
- Bash (Bourne again shell) is a command interpreter and programming environment. Having a shell is essential on a UNIX system, because the shell interprets and executes the commands you type.

Later on, the GNU project also started development on a UNIX-like kernel¹, called Hurd. This kernel has never been important for Linux, however, because it was released for the first time at the end of the 1990s, when Linux was already thriving.

¹ The kernel of an operating system is the program that runs 24 hours a day and takes care of scheduling, device handling, memory management and so forth.

A short history of Linux (2 of 2)

- 1991: Linus Torvalds writes first version of Linux kernel
 - Initially, a research project about the 386 protected mode
 - Linus' UNIX -> Linux
 - Combined with the GNU and other tools to form a complete UNIX system
- 1992: First distributions emerge
 - Linux kernel
 - GNU and other tools
 - Installation procedure
- The rest is history



© Copyright IBM Corporation 2012

Figure 1-3. A short history of Linux (2 of 2)

LX027.2

Notes:

In 1991, a student at the University at Helsinki, Linus Torvalds, started a small research project into the workings of the Intel 80386 processor, which by then was state-of-the-art. He was interested in exploring a new feature, which, up to then, was not present in any Intel processor, namely a memory management unit (MMU). This MMU offered hardware support for running multiple processes simultaneously, each in its own memory segment. With such an MMU, processes cannot access memory areas owned by other processes, and this effectively means that if one process crashes, it cannot take the whole system down with it.

The operating systems available for the 386 (Windows for Workgroups and Minix) all did not use this feature and were therefore very prone to crashing. ("Who is General Failure and why is he reading my hard disk?")

Linus started out writing three small programs:

- A small program that continuously printed the letter A on the screen
- A small program that continuously printed the letter B on the screen

- A slightly larger program that switched the processor to protected mode and scheduled the other two programs to take turns

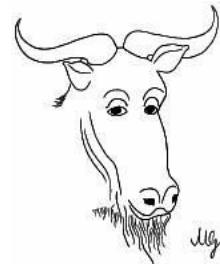
When Linus finally managed to see the output of both programs on his screen, in turn (ABABABAB...), he knew he had the beginnings of a kernel of a multitasking operating system. Linus continued to improve and refine the kernel, and at the end of 1991, he was able to run the GNU C compiler and the Bash shell under his kernel, which by then was dubbed Linux, for Linus' UNIX. Linus then decided to upload this to the Internet (which by then was still largely a university network) for others to use:

From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID: <1991Aug25.205708.9541@klaava.Helsinki.FI>
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki
Hello everybody out there using minix -
I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since April, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things). I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-(
Linus (torvalds@kruuna.helsinki.fi)
PS. Yes - it's free of any minix code, and it has a multi-threaded fs.
It is NOT portable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(.

Other people on the Internet started picking up this software, using it, and refining it. The patches were sent to Linus, who incorporated this into the main kernel stream. Starting to use Linux was a major undertaking, however. The Linux kernel, the C compiler, the shell, and all the other tools you need to make a complete operating system were all distributed in source code form. Before you can make use of them, you need to compile them, which requires a C compiler, which itself also needs to be compiled first. To break through this vicious circle, people started creating distributions which contain precompiled versions of the kernel, C compiler, various tools, and some sort of installation program. All this is stored in a convenient format for installation (originally floppy disk images, but today CD-ROM and DVD, or ISO, images are prevalent). The rest, as they say, is history. Because of the close association between Linux (the kernel) and the GNU project, an operating system distribution including the Linux kernel and GNU libraries and utilities is often called "GNU/Linux". This is Richard Stallman's strong preference.

What's so special about Linux (GNU/Linux)?

- Most software (including the Linux kernel) is GPL'ed (GNU General Public License).
 - <http://www.gnu.org/copyleft/gpl.html>
- Linux is called *copyleft* (instead of *copyright*).
 - You can copy the software.
 - You get the source code.
 - You can alter the source code and recompile it.
 - You can distribute the altered source and binaries.
 - You can charge money for all this.
- You cannot change the license.
 - All your customers have the same rights as you.
 - You really cannot make money from selling the software alone.
- Other Open Source licenses (for example, BSD) are also used.



© Copyright IBM Corporation 2012

Figure 1-4. What's so special about Linux (GNU/Linux)?

LX027.2

Notes:

To understand what is so special about Linux, it is necessary to quickly look at international copyright laws. The principle of copyright is very simple: When an author creates a unique piece of work, such as a computer program, then he is the owner of all rights to that piece of work. He may decide what others can and cannot do with it.

What others may do with that piece of work is usually written down in a License Statement, a contract between the creator and the user, which describes the rights that the user has. These rights may be granted for free, but in most cases the user has to pay for them.

A typical license in the world of computer software entitles the user to run the binary program on the number of machines that the license was purchased for. It is not allowed to make more copies of the software than needed for running it, and one extra backup copy. Furthermore, the user cannot claim any rights to the source code and is not allowed to disassemble the binary code to learn and/or alter its inner workings. In short, a typical copyright statement does not give you the right to copy.

In contrast, the GNU General Public License or GPL for short, turns this around. The aim of the GPL is to keep all software free so that everybody can adapt the software to their own

needs, without being dependent on the goodwill of the author. This means that any piece of software that has been placed under the GPL by the original author gives the user the following rights:

- The user can copy the (binary) software as often as the user wishes.
- The user has the right to obtain the source code.
- The user has the right to alter the source code and recompile the source code into binary form.
- The user can distribute the sources and the binaries.
- The user can charge money for all of this.

Basically, the only restriction that the GPL imposes on all users is that the license statement may not be changed. This means that all your customers have the same rights to the software as you do. And as a practical aside, that means that in general, it is impossible to make any money from selling the software (apart from a nominal fee for media and distribution).

The GPL is the most-often used license statement for open source projects, but other licenses, such as the BSD license, are also being used. BSD-style licenses have very few restrictions.

Effects of the license model

- Everybody has access to the source.
 - Volunteer software development on the Internet with centralized coordination.
 - Linus Torvalds coordinates new core kernel development.
 - Others coordinate other pieces of the kernel and OS.
- Peer reviews are possible.
 - Security
 - Performance
 - Reliability: “Given enough eyeballs, all bugs are shallow.”
- License cannot change.
 - Your changes (and name) will stay in forever.

© Copyright IBM Corporation 2012

Figure 1-5. Effects of the license model

LX027.2

Notes:

The effects of this license model are far reaching.

The first effect is that, since everybody has access to the source code, everybody interested can improve the code, or add new features. This means that software development is potentially very rapid, with possibly hundreds of developers working on the same piece of code. People in the Linux community understand the inherent risk of a code fork with a development model like this, and a lot of effort is spent in coordinating the work of various developers. (A *code fork* is when one project becomes two independent projects, likely with different goals, so the source code will begin to diverge from a common base.) This usually comes down to two things.

A volunteer or group of volunteers are responsible for coordination of the development. Linus Torvalds, for instance, hardly writes any code anymore, but spends most of his time coordinating others who write code for the kernel. Other people coordinate the development on other programs.

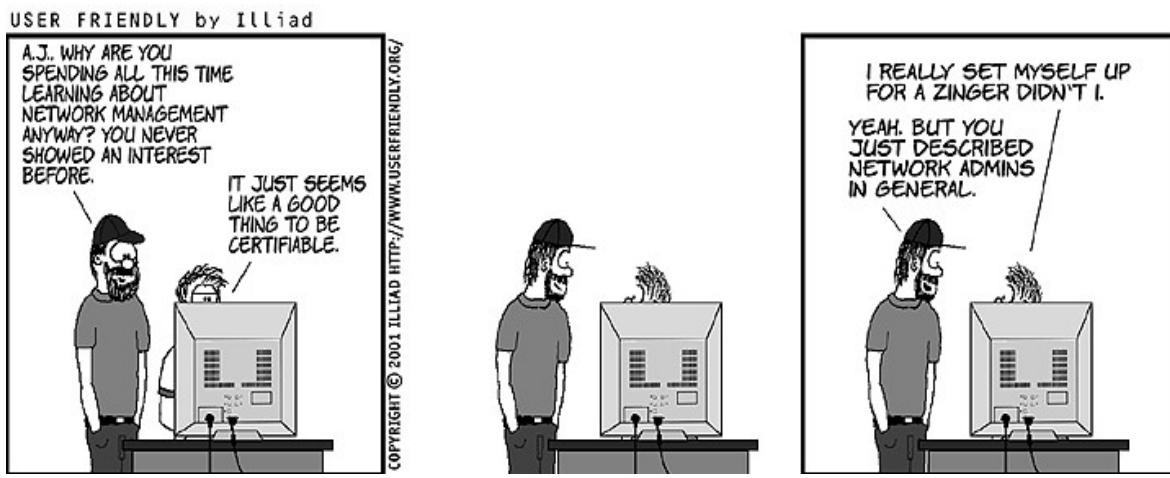
Some sort of automated support for distinguishing and integrating contributions of developers. Common source code control systems include git (currently used for the Linux kernel), cvs, subversion.

Another effect of having the source available is that peer reviews are possible. It is easy for people to look through the code and identify performance or security problems. Eric Raymond, in his essay “The Cathedral and the Bazaar”, coined what he called “Linus’s Law”, more formally stated “Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix will be obvious to someone.”

A third effect of the license model is that if you make significant changes, or add a feature, then that feature is owned by you, and not by the original author of the software. This means that your name (as part of the copyright statement for that feature) stays in forever. This is usually a great motivation factor for people.

Linux has become a way of life

- Culture
- Celebrities
 - Linus Torvalds
 - Richard Stallman
 - Eric Raymond
- Humor
 - User friendly
 - XKCD
- Mascot
 - Tux



© Copyright IBM Corporation 2012

Figure 1-6. Linux has become a way of life

LX027.2

Notes:

For a large number of people, Linux is not just another operating system, but it has become a way of life for them. It is something they believe in, and they want to express that belief. Devotees often associate Linux with “freedom”, as expressed with the GPL license, as well as freedom from the “tyranny of a monopoly”, usually with reference to Microsoft. Linux is also associated with the following:

- **Capability:** Any newly desired feature can be realized with a simple matter of programming.
- **Flexibility:** You can do whatever you want since you have all of the source code.
- **Performance:** Since you can recompile everything, you can optimize everything to suit your specific environment.

As part of the Linux identity, early in 1996, people felt the need for a logo for Linux. After having discussed various designs over the Internet, Linus stepped in and said that he would like to see a penguin as logo for Linux. Simply because he liked penguins. Several authors then started drawing penguins to use as the Linux logo, and by popular acclaim the

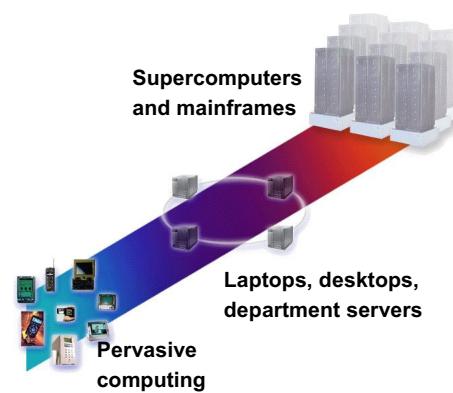
penguin drawn by Larry Ewing was chosen as the official logo; however, the penguin should be seen more as a mascot than as a logo, and people are free to create their own penguin-based logos (or adapt Larry Ewing's picture) for their own purposes.

The penguin was eventually named Tux, which officially stands for [T]orvalds [U]NI[X]. A real Tux exists as well: A number of UK Linux fans, lead by Alan Cox, and the Linux World magazine have sponsored a live penguin at the Bristol Zoo as a birthday present for Linus.

For a complete overview of the history of Tux, including links to other sites, see
<http://www.sjbaker.org/tux>.

Linux today

- Linux covers the whole spectrum of computing.
 - Embedded devices, smartphones
 - Laptops
 - Desktop systems
 - Development systems
 - Small and large servers
 - Mega clusters and supercomputers
- Linux is used throughout the world and in space.
- Linux is used by home users, by most of the largest companies in the world, and by many governments and institutions.



© Copyright IBM Corporation 2012

Figure 1-7. Linux today

LX027.2

Notes:

The smallest implementations of Linux can be found in embedded devices: the microchips that control your VCR, microwave, router, and so forth. The Tivo (<http://www.tivo.com>), for instance, runs on Linux. And so does every Android-based smartphone.

In the regular IT-world, Linux runs on laptops, desktops and servers. And it is even used to run most of the largest supercomputers in the world: As of November 2011, Linux ran on 457 of the Top 500 supercomputers in the world (source: www.top500.org).

Linux is used by home users as well most of the largest companies in the world, numerous governments and educational institutions. Linux powers most of the web servers on the internet.

Linux is used throughout the world, and in space: Various experiments on board of the International Space Station are controlled by systems running Linux.

Linux powered the Deep Blue supercomputer that beat reigning world chess champion Garry Kasparov. And recently, IBM created a Power-based supercomputer called

“Watson”, which defeated the two most successful contestants of the Jeopardy! game show.²

² <http://www.ibm.com/innovation/us/watson/index.html>

Linux hardware support



- Also supported:
 - ARM, Itanium, Sparc, PA-Risc, 68000 and so forth

© Copyright IBM Corporation 2012

Figure 1-8. Linux hardware support

LX027.2

Notes:

Almost all operating systems that are currently available are written for one specific hardware architecture. Originally, that was the case for Linux as well. Linus Torvalds only had an Intel-based PC. This changed in 1994 when Digital Equipment Corporation (DEC, later bought by Compaq, which subsequently merged with HP) gave a DEC Alpha to Linus Torvalds, no questions asked. A few months later, Linux ran on the DEC Alpha.

People took on the effort of porting Linux to other architectures and fed back the architecture-dependent code to Linus. Later, companies with a commercial interest in running Linux on their hardware architecture started contributing as well. This means that Linux now runs natively on a very large number of platforms, including the following:

```
$ ls /usr/src/linux/arch
alpha  blackfin  h8300  m68k        mips      powerpc  sh       x86
arm    cris       ia64   m68knommu  mn10300  s390     sparc    xtensa
avr32  frv        m32r   microblaze parisc    score    um
```

The list in the visual is just the list of architectures that are supported in the mainline kernel. Patches exist for other architectures as well, but have not all been made part of the mainline kernel.

Within an IBM context, the most important architectures are x86/ia64 (Intel, both in 32 and 64 bit, for IBM System x), powerpc (for IBM Power-based systems, such as IBM System p and IBM System i) and s390 (for IBM System z mainframes). All these architectures are natively supported, meaning that the same source code base is used on all architectures. Furthermore, all are tier-1 architectures for Red Hat and SUSE, meaning that the availability of new distributions, service packs, patches and so forth happens for all these architectures at (virtually) the same time.

Looking at Linux

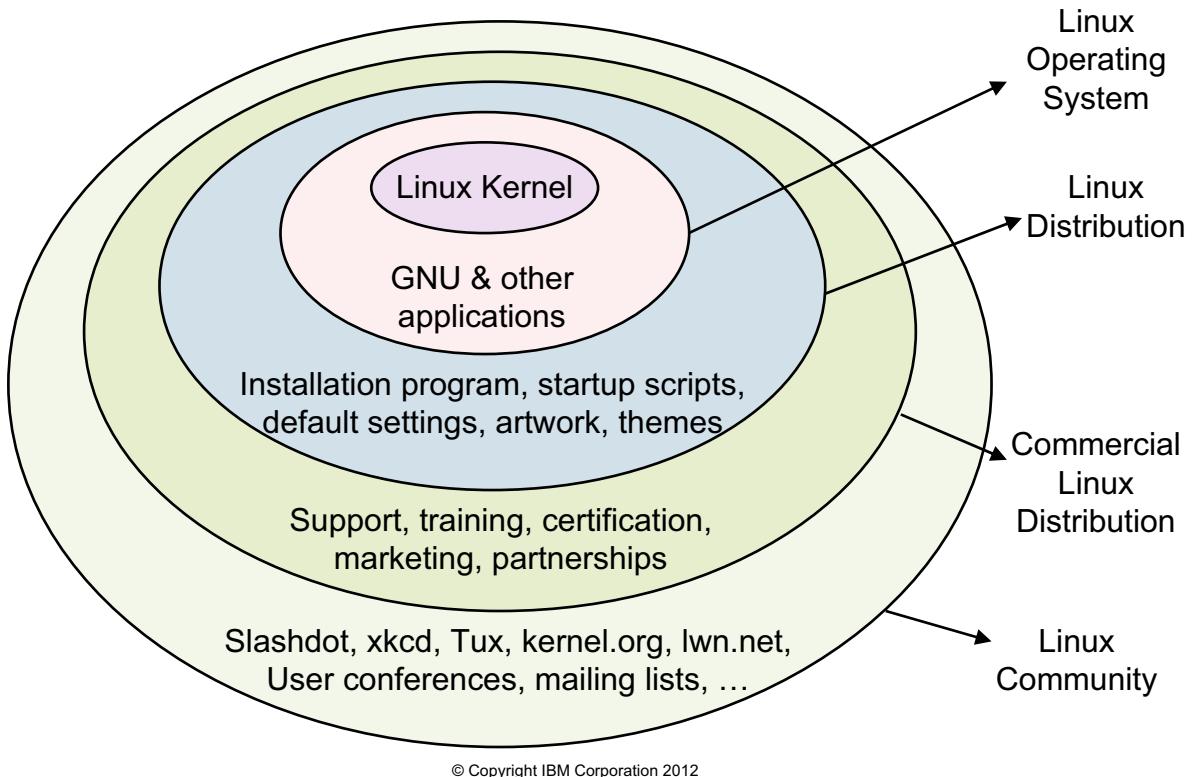


Figure 1-9. Looking at Linux

LX027.2

Notes:

Linux, in its most narrow definition, refers to the kernel, the heart of the operating system that was originally written by Linus Torvalds. But today, the word "Linux" is used in much broader definitions too.

Any operating system that uses the Linux kernel, augmented by GNU and other tools, can be thought of as a "Linux Operating System".

Recompiling the Linux kernel and the hundreds of tools that are required to run a Linux Operating System can be a very daunting task. That's why "Linux Distributions" appeared quickly after the first releases of the Linux kernel. A distribution generally consists of an installation program to install the precompiled Linux kernel and other tools quickly and easily on your system. It will also include things like startup scripts, default settings, management tools, package installation tools, artwork, themes and so forth.

Creating a Linux Distribution is a large task. Some distributions, most notably Debian, manage to enlist the help of sufficient volunteers so that their distribution can be downloaded and supported for free. Most distributions today however have a model where the users of that distribution need to pay for support, training, certification, marketing,

partnerships and so forth. And this income stream is then used to pay for the further development of the distribution itself. Examples of companies that have created a commercial enterprise surrounding their distribution are Red Hat, SUSE and Ubuntu.

The last way of looking at Linux is by looking at the Linux Community. This obviously includes the code itself, and the people developing and supporting that code. But it includes more. There are various websites that are very Linux-oriented, there are user conferences, mailing lists, support forums and so forth. And of course there's Tux, the mascot of Linux.

Unit review

- The Linux kernel, combined with the GNU and other tools, forms a complete UNIX-like operating system.
- A distribution pulls together versions of the Linux kernel, libraries, and tools, tests them as a cohesive operating system, and adds an installation procedure and a convenient format for distribution.
- Most software in a Linux distribution is licensed under Open Source licenses such as the GNU GPL.
- Linux is developed as a world-wide collaboration of corporations and volunteers.
- Linux has been ported to more than 20 hardware architectures including virtually all PC hardware.
- Linux is used in a variety of small and large applications, homes, schools, small and large businesses, and governments.

© Copyright IBM Corporation 2012

Figure 1-10. Unit review

LX027.2

Notes:

Checkpoint

1. True or False: Linus Torvalds wrote the Linux operating system all by himself.

2. Which of the following statements is *not* true about software licensed under the GNU GPL?
 - a. You have the right to obtain and review the source code.
 - b. You cannot charge any money for the software.
 - c. You cannot change the license statement.
 - d. You can modify the source code and subsequently recompile it.

3. Who is the mascot of Linux?

© Copyright IBM Corporation 2012

Figure 1-11. Checkpoint

LX027.2

Notes:

Unit summary

Having completed this unit, you should be able to:

- Discuss the history of Linux
- Name some important people in the history of Linux
- Discuss the GNU general public license

© Copyright IBM Corporation 2012

Figure 1-12. Unit summary

LX027.2

Notes:

Unit 2. Installing Linux

What this unit is about

This unit covers the installation process of Linux.

What you should be able to do

After completing this unit, you should be able to:

- Prepare a system for installation
- Install Linux from CD-ROM
- Perform a network installation

How you will check your progress

- Checkpoint questions
- Machine exercises

Unit objectives

After completing this unit, you should be able to:

- Prepare a system for installation
- Install Linux from CD-ROM
- Perform a network installation

© Copyright IBM Corporation 2012

Figure 2-1. Unit objectives

LX027.2

Notes:

Preparing a system for installation

- Know your hardware.
 - CPU, memory, keyboard, mouse
 - Hard disks, CD-ROM players
 - Graphical adapters, monitor capabilities
 - Network adapters, IP addresses
 - Printers
- Is all your hardware supported?
 - Linux Hardware-HOWTO
 - Distributor's hardware compatibility list
 - Hardware manufacturer
 - If unsure, just try it!
- Determine where Linux will be installed.

© Copyright IBM Corporation 2012

Figure 2-2. Preparing a system for installation

LX027.2

Notes:

Before you install a Linux system, there are some things you should do. One of the most important steps is knowing what hardware you have, and all the characteristics and configuration options of that hardware.

Furthermore, you need to verify that all your hardware is supported by Linux. Since not all hardware manufacturers make the specifications of their hardware public, some hardware is not supported, or not supported in full. For a detailed list of hardware supported by Linux, refer to the Hardware-HOWTO at <http://www.tldp.org>. Also, your distributor might have several restrictions on the hardware that their distribution supports. You might also be able to obtain information from the hardware manufacturer itself. If you are unsure whether your hardware is supported, just go ahead and try it.

Another important step is making space for Linux partitions. Linux must have its own physical or virtual disk partitions available for install.

Installing Linux

- Boot system from bootable media.
 - CD/DVD
 - USB
 - PXE
- After booting, install from:
 - CD/DVD
 - Local hard disk
 - Network

© Copyright IBM Corporation 2012

Figure 2-3. Installing Linux

LX027.2

Notes:

Installing Linux starts with booting a very tiny Linux system from some sort of bootable media. All systems in use today can boot from the installation CD/DVD directly. (If you still have a system that can only boot from a floppy disk, there's a good chance it will not be supported by your distribution anyway.)

Quite a large number of distributions today also support booting from USB. This is generally used to run "live" distributions, where the whole operating system is run from USB. The reason for using USB for "live" distributions over a DVD is the form factor (it's easier to hang a USB key on your keychain and bring it along than a DVD) and the read/write capability of the USB key.

The third common method of booting a Linux installation program is by using PXE (Preboot Execution Environment), where all software components are downloaded from a PXE server. In this case, no local medial whatsoever is required to start the installation.

Once the installation program has been started, it needs access to the repository of packages that need to be installed on the system. This repository is typically stored on the local CD/DVD, the local hard disk or on a network server.

Network installations

- Installations where packages to install are downloaded from the network
- Network protocols supported depends on distribution
 - Network File System (NFS)
 - File Transfer Protocol (FTP)
 - Hypertext Transfer Protocol (HTTP)
 - Server Message Block (SMB)
- Requires a network install server
- Usually requires special network-enabled boot media
 - Preboot Execution Environment (PXE) boot requires no media



© Copyright IBM Corporation 2012

Figure 2-4. Network installations

LX027.2

Notes:

Most Linux systems are installed from the distribution CD-ROMs (or DVDs). This is a convenient method if you only need to install one or a few systems but quickly becomes tedious if you need to install ten or more systems, especially if each system has to be installed with the same settings.

More advanced installation methods exist which are convenient for these situations, and in all but a few cases, this comes down to network installations where the Red Hat Package Manager software packages (RPMs) to be installed are downloaded from the network.

Network protocols

Various network protocols exist to retrieve the installation RPMs, and the protocols that are supported depend on your distribution. Support might be included for Network File System (NFS), File Transfer Protocol (FTP), Hypertext Transfer Protocol (HTTP), and Server Message Block (SMB).

An obvious requirement for a network-based install is that somewhere on the network you need to configure a network install server, which holds all the RPMs for your distributions.

Finally, you will need some network-enabled boot media. This can be the first CD (or DVD) of your regular installation or a minimal install CD or DVD ISO image. If your system supports the Preboot Execution Environment (PXE), you can boot and install your distribution over the network without the need for physical boot media.

For PPC distributions, the distribution media includes a network bootable install kernel that can be used to install the machine over the network.

Installation steps

- All installation programs need to perform essentially the same steps.
 1. Choose language, keyboard type, mouse type.
 2. Create partitions/logical volumes.
 3. Set up a boot loader.
 4. Configure network.
 5. Configure users and authentication.
 6. Select package groups.
 7. Configure X.
 8. Install packages.
 9. Register.
- The order of steps might vary from distribution to distribution.
- Other steps might also be included.
 - For example, firewall, printers, and sound.

© Copyright IBM Corporation 2012

Figure 2-5. Installation steps

LX027.2

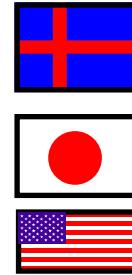
Notes:

After the system is booted, the installation program takes over, and asks you a number of questions regarding the Linux configuration. It then installs Linux and configures it according to the options you specified.

Obviously, every distribution has its own installation program; so the exact order in which questions are being asked is different from distribution to distribution. However, every distribution basically needs to do the same things in its installation process; so if you look beyond the order of the various menu screens, the installation programs do not differ that much from each other.

Select language, keyboard, and mouse

- Select the language to be used during installation process.
 - Different distributions support different languages.
- Select the keyboard layout.
 - Different countries use different keyboard layouts.
 - Dead (compose) keys allow you to input accented or special characters, such as é, ç, ß, and so forth.
- Select your mouse.
 - A mouse can be connected using a PS/2, USB, or serial connector.
 - If your mouse has only two buttons, you can emulate the third (middle) button by clicking both buttons simultaneously.



© Copyright IBM Corporation 2012

Figure 2-6. Select language, keyboard, and mouse

LX027.2

Notes:

One of the first things the installation program needs to do is to determine the language to be used during installation and to determine the keyboard layout and mouse type.

Install class

- Some distributions have installation classes for typical users.

- Desktop



- Laptop



- Server



- Developer

- A custom class allows you to make all decisions yourself.

- Packages to be installed
 - Various configuration options

© Copyright IBM Corporation 2012

Figure 2-7. Install class

LX027.2

Notes:

Some distributions allow you to select an installation class. These classes allow you to quickly install a typical system. Among other things, a class determines the packages that are installed and various configuration options. If a distribution uses these classes, then it always supports a “custom” class, too, which allows you to make all decisions yourself.

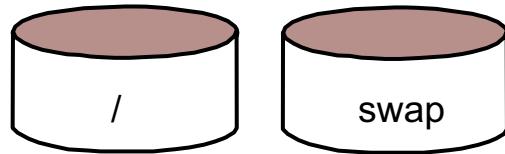
Note that some distributions also make assumptions regarding partitioning, depending on the class chosen. As an example, a Red Hat Workstation install removes all existing Linux partitions, and a Red Hat Server install removes ALL existing partitions, including any non-Linux partitions. If you choose to use an installation class, make sure to read the documentation.

Most distributions also support an upgrade class, which does not make any configuration changes but upgrades all currently installed software to the latest level.

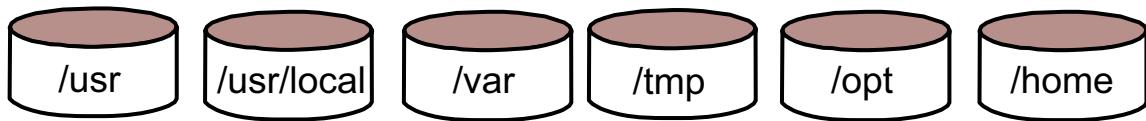
Disk partitioning

- Linux installation requires you to create Linux partitions.

- At a minimum, create / and swap:



- You might need or want to create other partitions.
 - For example, /usr, /usr/local, /var, /tmp, /opt, and /home



- Some distributions will use Logical Volume Management (LVM) by default

© Copyright IBM Corporation 2012

Figure 2-8. Disk partitioning

LX027.2

Notes:

Almost every Linux distribution allows you to partition your disks during the installation process. Most distributions use **fdisk** for this, but some distributions include their own partitioning tool. No matter what tool you use, you need to create at least two partitions:

- The first partition to create is your root partition. This partition holds the filesystem, which in turn holds all your data. Sizing depends on whether other file systems are going to be created. If this is the only filesystem, it should be a minimum of 5Gb in size.

Note that it is possible to create more partitions (/usr, /tmp and so forth). In that case, your root partition doesn't have to be this large.

- The second partition does not hold a file system but is used as swap space. This is virtual memory, which is used when your system exhausts its real memory. The opinions on the size of this swap space vary, but it is usually best to take the amount of real memory on your system as swap space. (When your system starts using the swap space, it generally means that you do not have enough real memory to run all your processes in. This leads to a huge performance loss, since hard disk accesses are far

slower than memory accesses. Memory today is so cheap that you should size your system so that you do not need swap space, except in a rare situation.)

Note that the swap space needs to have another partition type (type 82), and does not get a mountpoint.

Even though it is not strictly needed on most systems, it is a good idea always to create a /boot partition (that is short for saying “a partition which holds a filesystem that is mounted at the /boot mountpoint” of about 100 Megabytes¹. This partition holds everything that is needed by the Linux boot process. The most important program here is the Linux kernel itself, but you also need to have some components of LILO or GRUB stored here.

The reason to store these components on a separate partition is complex and outside the scope of this course.

Depending on what you are going to do with your system, you might also need to create other partitions for /usr, /usr/local, /var, /tmp, /opt, /home and so forth. When this is needed and how these are created is outside the scope of this course; they're not needed for a Linux workstation anyway. When partitioning, make sure you don't delete any existing Windows partitions, and make sure that you format only all newly created Linux partitions.

Some distributions will use Logical Volume Management (LVM) by default. LVM is an improvement over traditional disk partitioning since it allows you to expand/reduce a volume in size on the fly, and it permits volumes that are larger than a single disk size. It is however slightly more complex to understand, and outside the scope of this course.

¹ Red Hat requires your /boot partition to be at least 100 Megabytes so that you can install several kernel images.

Configure a boot loader

- A boot loader loads and starts the Linux kernel.
- It can boot other operating systems as well.
 - Windows, BSD, and so on
 - Give each OS a unique label
- It can be password protected.
 - Prevents users from passing boot parameters to Linux or booting any OS
- Should generally be configured in the MBR unless another boot loader is used.
- Common boot loaders include:
 - GRUB: Grand Unified Boot Loader
 - YaBOOT: Yet Another Boot Loader

© Copyright IBM Corporation 2012

Figure 2-9. Configure a boot loader

LX027.2

Notes:

One of the next screens allows you to configure a boot loader. This is a program that loads and starts the Linux kernel. It can also pass various boot parameters to the Linux kernel, such as device information.

A boot loader can also be used to boot non-Linux operating systems, such as Windows. For this to work, your boot loader, your boot loader typically needs to be stored in the master boot record (/dev/hda). If you use another boot loader, such as BootMagic, then the boot loader that loads Linux is usually stored in the Linux root partition (the partition that holds the root filesystem).

Every OS that needs to be bootable is identified with a label, which you can choose yourself. This label is used to select the operating system that is booted when your system is switched on. If you don't make a selection, then after a number of seconds (usually five), the default OS is booted. Currently, two boot loaders are in use: GRUB and YaBOOT.

GRUB, the Grand Unified Bootloader, is a successor to LILO, an older boot loader specifically written to load Linux. YaBOOT (Yet Another Boot Loader) is an open firmware executable that bootstraps the Linux kernel on ppc hardware.

All boot loaders support passwords. These passwords, if configured, are required if the user wants to pass parameters to the kernel when the system is booting. These parameters could, for instance, be used to boot the system into single user mode, where the user automatically becomes root without having to log in.

Configure network

- Most distributions configure your network adapter as part of the installation process.
- You need the following information:
 - IP address
 - Subnetmask
 - Network address
 - Broadcast address
 - Host name
 - Default router/gateway
 - DNS server addresses
- They can also be configured to use DHCP.
- Most distributions do not support wireless adapters at install time.

© Copyright IBM Corporation 2012

Figure 2-10. Configure network

LX027.2

Notes:

Most distributions can configure your (Ethernet or Token Ring) network adapter during the installation. For this to work, you need to obtain the following information from your network manager:

- IP address
- Subnetmask
- Network address
- Broadcast address
- Host name
- Default router/gateway
- DNS server addresses

If your workstation resides on a network where a DHCP server is present, you can also configure your system to use this DHCP server to obtain this information.

Configure root and user accounts

- *root* is the superuser of the system.
 - It can do anything.
 - It needs a strong password.
 - Do not use your system as root unless you need to!
- Most distributions allow you to add user accounts during installation as well.
 - Create a user account for every individual user that is going to use the system.

© Copyright IBM Corporation 2012

Figure 2-11. Configure root and user accounts

LX027.2

Notes:

On a Linux system, the superuser is called **root**. If you (or anybody else) is logged into the system as this user, you can do anything to the system. This is considered very dangerous, and that is why you need to configure a strong password for this account. A good policy to live by is never to use the root account unless you really need to. You can use authorization elevation tools such as '**sudo**' instead.

Most distributions also allow you to add regular user accounts during the installation. If your distribution does this, then create user accounts for every user that is going to use your system.

Some distributions also allow you to configure your workstation as an NIS, LDAP, or Kerberos client, or as a client of some other network authentication method. Only do this if your network supports this (ask your network administrator).

Select package groups

- Most distributions group individual packages in package groups.
- Only select the package groups you need on your workstation.
- Selecting individual software packages is usually still possible but tedious.
 - A typical distribution has over 1000 packages.

© Copyright IBM Corporation 2012

Figure 2-12. Select package groups

LX027.2

Notes:

A typical distribution consists of over 1000 individual packages (software components) that should or can be installed. To save the user from having to make 1000 or more informed decisions, these packages are often grouped into package groups. Instead of having to decide on each and every individual package, you decide whether to install a package group. This greatly reduces the complexity of selecting the packages you want to install.

Most distributions still offer the select individual packages option though, in case you've got nothing better to do or need to run a really tight system (security-wise or hard disk space-wise).

Configure X

- X (X Window System) is the graphical user interface of Linux.
- It needs to be configured for your system.
 - Graphical adapter
 - Monitor
- Most adapters and monitors are autodetected.
 - If not autodetected, select manually or use a generic adapter or monitor
- Usually, customization is allowed.
 - Resolution, refresh rate
 - Color depth
- Test settings if possible.
- If nothing works, skip X configuration.

© Copyright IBM Corporation 2012

Figure 2-13. Configure X

LX027.2

Notes:

The X Window System (X for short) is the graphical user interface (GUI) of Linux. It needs to be configured for your graphical adapter and monitor to provide optimal performance. Most distributions incorporate auto detection mechanisms, which detect your adapter and monitor automatically. If this fails, select your adapter and monitor manually or use a generic adapter or monitor.

Within the limits of your adapter and monitor, you can customize your resolution (the amount of pixels on your screen), your refresh rate (the number of times your screen is refreshed, per second) and the color depth (the amount of colors that can be displayed simultaneously). There usually is a trade-off to be made here: a higher resolution usually means less color depth, and a higher resolution also means a lower refresh rate. It is therefore important that, if possible, you test the configuration before you continue the installation process.

Occasionally, it happens that X cannot be configured from the installation process at all. Trying to configure X might in some rare case even hang the system altogether.

Other (optional) installation screens

- Some distributions offer additional installation screens.
 - Printer configuration
 - Firewall configuration
 - Sound card configuration
 - Modem configuration
 - Time zone configuration
- These are usually straightforward



© Copyright IBM Corporation 2012

Figure 2-14. Other (optional) installation screens

LX027.2

Notes:

Some distributions offer other configuration screens in addition to the ones we've covered. This might include configuration of printers, firewalls, sound cards, modems, time zones and so forth. These screens are usually straightforward. And if they are not, there is usually help available.

Installing packages

- The installation will take several minutes to complete.
 - Most distributions provide a progress bar or total time indication or both.
- While the installation is going on, various virtual terminals provide background information on the progress.
 - Switch between VTs using **Ctrl-Alt-F1**, **Ctrl-Alt-F2**, and so forth.
- Insert additional media when prompted.

© Copyright IBM Corporation 2012

Figure 2-15. Installing packages

LX027.2

Notes:

After all configuration choices have been made, the system starts installing packages. This installation might take a short or a long time, depending on your hardware and amount of packages to install. Most distributions, however, display some sort of progress bar or time indication so that you can estimate how long your coffee or lunch break is going to be.

While the installation is going on, you can get additional information about it from various virtual terminals. Virtual terminals are pseudo-monitors, which can be accessed using **Ctrl-Alt-F1**, **Ctrl-Alt-F2** and so forth. Make sure to insert additional media when prompted.

Post-install configuration

- After installation has finished, your system will reboot to activate the newly installed kernel.
 - SLES performs the reboot during installation.
- After reboot, some post-installation configuration might happen.
 - Configure graphics.
 - Configure sound card.
 - Install documentation, updates, and drivers.
 - Create user accounts.
 - Register.

© Copyright IBM Corporation 2012

Figure 2-16. Post-install configuration

LX027.2

Notes:

For most distributions, after the packages have been installed and configuration has been done, the install is over. The only thing left to do is to reboot the system to activate the newly installed kernel. SUSE is an exception to this: SUSE already activates the kernel (via a reboot) during the installation.

After the reboot, some post-installation configuration might happen, depending on the distribution. Red Hat, for instance, attempts to register your system with RHN, the Red Hat Network.

Checkpoint (1 of 2)

1. True or False: Linux can coexist with Windows on the same hard disk.

2. Which of the following steps is not essential in the installation process:
 - a. Create partitions for Linux.
 - b. Configure your printer.
 - c. Select your keyboard type.
 - d. Identify the packages to install.

3. What is the best source of information about your hardware?

© Copyright IBM Corporation 2012

Figure 2-17. Checkpoint (1 of 2)

LX027.2

Notes:

Checkpoint (2 of 2)

4. True or False: A network install server needs to be a Linux system.

5. Which of the following install methods does not require a network server?
 - a. NFS
 - b. SMB
 - c. FTP
 - d. CD-ROM

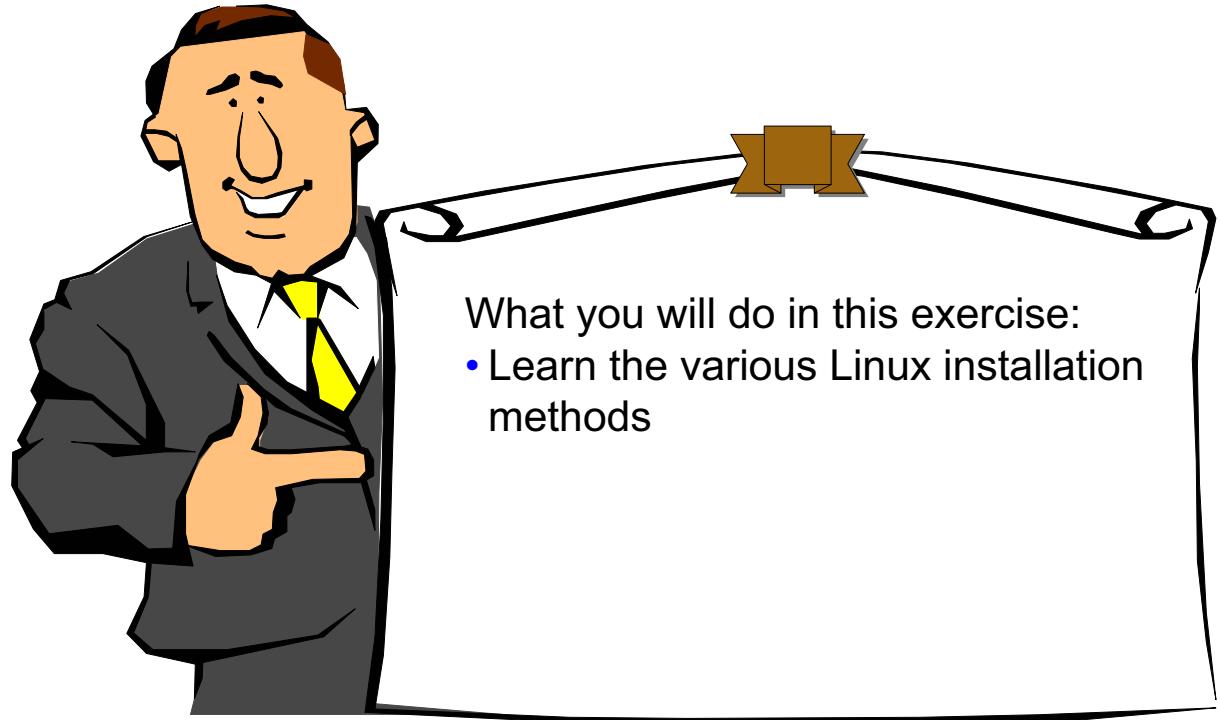
© Copyright IBM Corporation 2012

Figure 2-18. Checkpoint (2 of 2)

LX027.2

Notes:

Exercise: Linux installation



© Copyright IBM Corporation 2012

Figure 2-19. Exercise: Linux installation

LX027.2

Notes:

Unit summary

Having completed this unit, you should be able to:

- Prepare a system for installation
- Install Linux from CD-ROM
- Perform a network installation

© Copyright IBM Corporation 2012

Figure 2-20. Unit summary

LX027.2

Notes:

Unit 3. Using the system

What this unit is about

This unit describes logging in and out of the system, the structure and execution of Linux commands, and using Linux commands to communicate with other users. This unit will also discuss using the keyboard and mouse effectively and the X Window System.

What you should be able to do

After completing this unit, you should be able to:

- Log in and out of the system
- State the structure of Linux commands
- Execute basic Linux commands
- Use Linux commands to communicate with other users
- Use the keyboard and mouse effectively
- Discuss the X Window System

How you will check your progress

- Checkpoint questions
- Machine exercises

Unit objectives

After completing this unit, you should be able to:

- Log in and out of the system
- State the structure of Linux commands
- Execute basic Linux commands
- Use Linux commands to communicate with other users
- Use the keyboard and mouse effectively
- Discuss the X Window System

© Copyright IBM Corporation 2012

Figure 3-1. Unit objectives

LX027.2

Notes:

Linux is multiuser and multitasking

- Linux is a multiuser, multitasking operating system.
 - Multiple users can run multiple tasks simultaneously, independent of each other.
- You always need to log in before using the system.
 - Identify yourself with user name and password.
 - “root” is the superuser of the system.
- There are multiple ways to log in to the system.
 - You can log in through the console: Directly attached keyboard, mouse, monitor.
 - [Usually used for virtual terminals](#)
 - You can log in through a serial terminal (rare).
 - You can log in through a network connection (for example, SSH or Telnet).

© Copyright IBM Corporation 2012

Figure 3-2. Linux is multiuser and multitasking

LX027.2

Notes:

Just like conventional UNIX systems, Linux is designed from the ground up as a multiuser, multitasking operating system. This means that multiple users can run multiple tasks simultaneously on the same system, independent of each other. Security is of course paramount on such systems, because it would be unacceptable if one regular user could stop or otherwise influence processes of other users. For this separation to work properly, user authentication is needed.

User authentication on a Linux system is done when you first start using the system. Before you can do anything, you need to identify yourself using your username and password. On a real multiuser system, typically, the system administrator assigns you a username and initial password. When using Linux on your personal workstation, you'll often administer the system yourself.

Administrators use a special super-user account, called root, which has security privileges to be able to create other user accounts and set their passwords.

There are multiple ways of logging into a Linux system and you can be logged in as the same user multiple times. A single log in instance is referred to as a *session*.

Virtual terminals

- In most Linux distributions, the console emulates a number of virtual terminals (VT).
- Each virtual terminal can be seen as a separate, directly attached console.
 - Different users can use different virtual terminals.
- The following is a typical setup:
 - VT 1 through 6: Text mode logins.
 - VT 7: Graphical mode login prompt (if enabled).
- Switch between VTs with **Alt-Fn** (or **Ctrl-Alt-Fn** if in X) or with the **chvt** command.

© Copyright IBM Corporation 2012

Figure 3-3. Virtual terminals

LX027.2

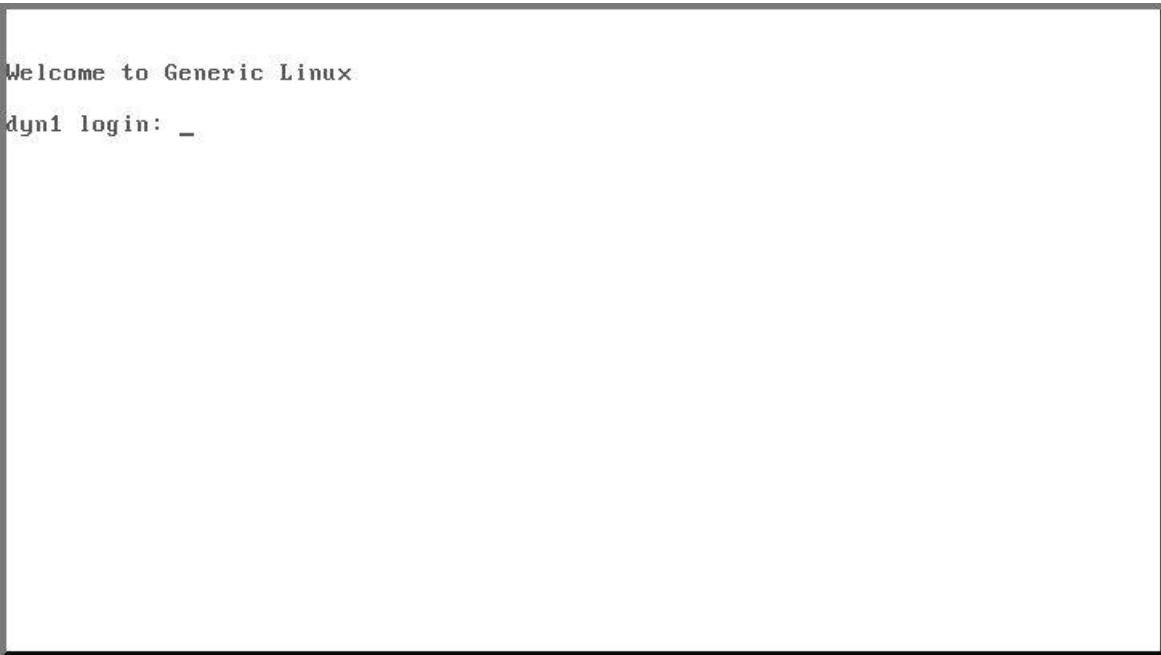
Notes:

In most Linux distributions, the console emulates a number of virtual terminals. These virtual terminals can be seen as separate, directly attached consoles and can be used by different users, although, in practice, this is rather inconvenient. Because the system in reality only has one console, there are hotkey combinations to switch from one VT to another. This hotkey combination is Alt-Fn, where the n is the virtual terminal number you want to access. When you are in an X environment, you need to use Ctrl-Alt-Fn instead.

The default virtual terminal setup differs from distribution to distribution, but the most common setup offers six text mode logins on VTs 1 through 6, and (if enabled) a graphical mode login on VT 7. (How this is enabled is covered later.)

Linux also has a **chvt** command that lets you switch virtual terminals.

Logging in: Text mode VT



```
Welcome to Generic Linux
dyn1 login: _
```

© Copyright IBM Corporation 2012

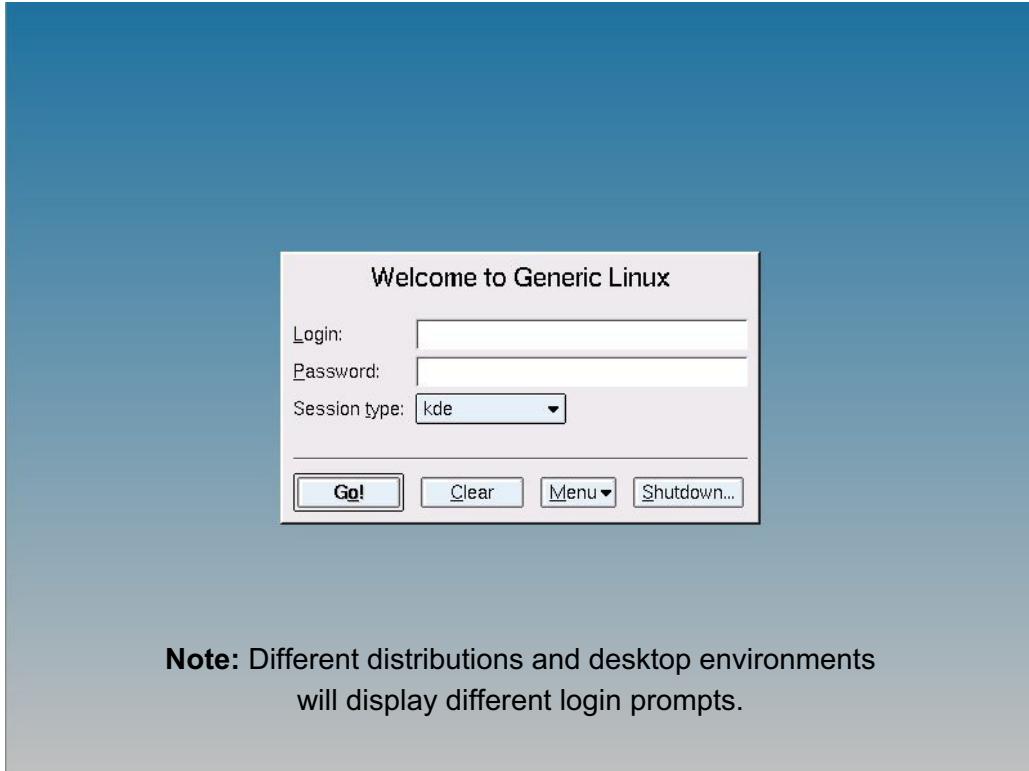
Figure 3-4. Logging in: Text mode VT

LX027.2

Notes:

The visual shows a text mode login session.

Logging in: Graphical mode VT



© Copyright IBM Corporation 2012

Figure 3-5. Logging in: Graphical mode VT

LX027.2

Notes:

The visual shows a graphical mode login session. The exact layout of the graphical login session varies from distribution to distribution and from desktop environment to desktop environment.

Linux commands

- Everything on a Linux system can be done by typing commands.
 - Even browsing the World Wide Web
- The graphical user interface (X Window System or X) is not needed for running a Linux system.
 - But is sometimes more convenient
- A terminal emulator will allow you to type commands when using the graphical interface.



© Copyright IBM Corporation 2012

Figure 3-6. Linux commands

LX027.2

Notes:

Every process that is running on a Linux system is started by a command, although for most processes, you never see that command because they are started automatically. In addition, every command can be executed from any login session, be it local or remote, which means a Linux system can be managed just as easily over the network as locally. You do not need to sit down at the console, in a possibly noisy and cold server room to manage a Linux system.

Similarly, most commands do not need a Graphical User Interface (X) to run. And if there are commands that do need X (such as the Web browsers Netscape and Konqueror), then there are usually text-based alternatives available that can do without X (such as Lynx, a text-based Web browser). Using the GUI is sometimes more convenient though, especially for things like graphics design, games and browsing the Web. Commands can obviously be run from a text-based terminal. But to run commands from within X, you need to start a terminal emulator.

Starting a terminal emulator

- To run a Linux command inside the X environment, open a terminal window or terminal emulator.
 - Emulates a text console
- Usually hidden in the menu under Applications; System Tools; Terminal
 - Drag to task bar for quick access

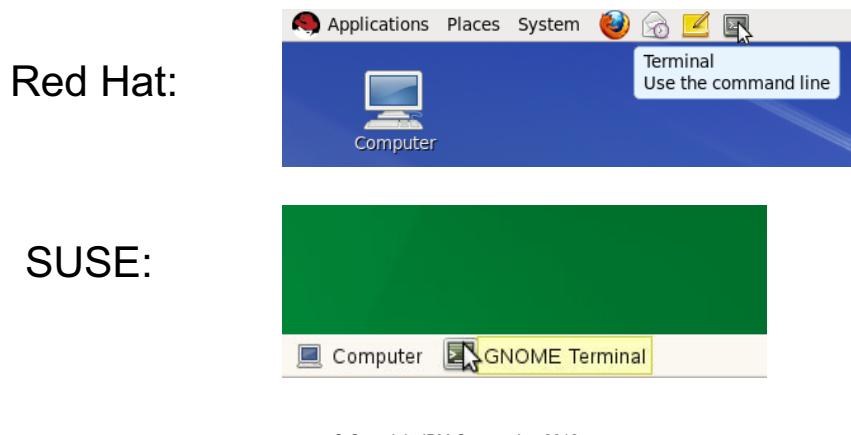


Figure 3-7. Starting a terminal emulator

LX027.2

Notes:

A *terminal emulator* is a program that emulates a text terminal in an X environment. The window that is consequently opened is called a terminal window. Various desktop environments have different terminal emulators, and have different buttons to start them. The visual shows the buttons for the GNOME and KDE desktop environments.

Command prompt

- The command prompt indicates that the system is ready to accept commands.
- It can be configured yourself (this will be covered later).
 - The default depends on distribution.
- Examples include:

```
[user@host dir]$  
dir$  
$  
#
```

- The dollar sign (\$) usually means you are logged in as a regular user.
- The hash sign (#) usually means you are logged in as root.

© Copyright IBM Corporation 2012

Figure 3-8. Command prompt

LX027.2

Notes:

The command prompt is the indication that the system is ready to accept commands. Only when the command prompt shows (in a text terminal or terminal emulator) can you type commands. (If the command prompt does not show, you can already type the beginnings of a command. The keys typed then appear as soon as the command prompt displays.)

What the command prompt looks like is something you can configure yourself; we do that later. Different distributions have different default settings, of which the visual shows some examples. What is important to note is that, for historic reasons, a dollar sign (\$) usually means that you are logged in as a regular user, and a hash sign (#) usually means that you are logged in as root.

Linux command syntax

- Linux commands have the following format:

- \$ command *options arguments*

```
$ ls  
$ ls -l  
$ ls /dev  
$ ls -l /dev
```

© Copyright IBM Corporation 2012

Figure 3-9. Linux command syntax

LX027.2

Notes:

The order and separation of the elements of a command is very important. The command or process name must come first. Spaces are used by the shell as separators on the command line and should not be placed within the command name.

The options should follow the command name, separated by a space and preceded by a minus sign (-) or a plus sign (+). Multiple options may be grouped immediately after a single minus (-) or separated by spaces and each preceded by a minus (-). Options are typically used to modify the operation of the process.

The arguments follow the options, again separated by a space. The order of the arguments depends on the command.

Hint: In order to determine valid command options and arguments, many Linux commands have built in help information.

Command help is accessed using the double minus sign (--) followed immediately by help, for example, ls --help .

We will go over this in greater detail in a later unit.

Command format examples

- Wrong

1. Separation

```
$ mail - f personal  
$ who -u
```

- Right

```
$ mail -f personal  
$ who -u
```

2. Order

```
$ mail test root -s  
$ -u who
```

```
$ mail -s test root  
$ who -u
```

3. Multiple options

```
$ who -m -u  
$ who -m u
```

```
$ who -m -u  
$ who -mu
```

© Copyright IBM Corporation 2012

Figure 3-10. Command format examples

LX027.2

Notes:

Some basic Linux commands

- **passwd:** Change your password.
- **date, cal:** Find out today's date and display a calendar.
- **who, finger:** Find out who else is active on the system.
- **clear:** Clear the screen.
- **echo:** Write a message to your own screen.
- **write:** Write a message to other screens.
- **wall:** Write a message to all screens.
- **talk:** Talk to other users on the system.
- **mesg:** Switch reception of write, wall, and talk messages on or off.

© Copyright IBM Corporation 2012

Figure 3-11. Some basic Linux commands

LX027.2

Notes:

In the next few visuals, we are going to look at the commands listed in the visual.

Changing your password

- The **passwd** command allows you to change your password.

```
$ passwd
Changing password for tux1
Old password:
New password:
Retype new password:
```



- Passwords are important for security, therefore you should choose a good password.
 - Minimum six characters
 - Not a dictionary word, birth date, license plate, and so on

© Copyright IBM Corporation 2012

Figure 3-12. Changing your password

LX027.2

Notes:

The user password is the primary mechanism for ensuring security on a Linux system. All passwords are encrypted and cannot be decoded. The **passwd** command is used to change the user password and is an example of a simple command that may be entered at the shell prompt.

The system starts the **passwd** process that prompts the user for a new password. To prevent users being locked-out of the system through simple typing errors, the new password is entered twice. Only if the two entries match is the new password accepted. The old password is invalid thereafter.

Choosing a good password is important, because too many systems have been broken into because of bad (easily guessed) passwords. The **passwd** command has options to set password expiration and age parameters of the users password.

The best passwords are passwords that have meaning to you (so they're easy to remember) but not (partly) listed in a database (for instance a dictionary) and are long enough not to be guessed easily by just trying out random passwords. You could for instance form a sentence that has meaning to you, and use the initial letters of each word

as your password. As an example, "MfewLwiD11@alc" (which is short for "My first encounter with Linux was in December 2011 at an IBM course") is a very strong password.

The date command

- **date** shows the current date and time.

```
$ date  
Wed Dec 14 12:17:46 UTC 2011
```

- All time/date functions in Linux are timezone aware.

```
$ export TZ=CET  
$ date  
Wed Dec 14 13:18:52 CET 2011
```

© Copyright IBM Corporation 2012

Figure 3-13. The date command

LX027.2

Notes:

The **date** command shows you the current system date and time.

Note that the time zone is displayed, too: Linux has a full implementation of time/date functions, which includes compensation for time zones and daylight savings time. This might seem overkill until you realize that people from all over the world might be using the same system at the same time, but all want their times to be displayed as local time.

You can play with your local timezone by setting the TZ variable. This will automatically set all times as displayed by **date** and other commands to that timezone. Setting variables will be covered later.

The cal command

- **cal** shows a calendar.
- Synopsis: **cal [Month] [Year]**

```
$ cal 1 2007
      January 2007
Su Mo Tu We Th Fr Sa
        1  2  3  4  5  6
 7  8  9  10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
$
```

© Copyright IBM Corporation 2012

Figure 3-14. The **cal** command

LX027.2

Notes:

With the **cal** command, you can look at the calendar of a given year or a given month in year.

Who is on the system

- **who** shows who is logged onto the system.

```
$ who
root          tty1          Jan 1 11:10
tux1      tty2          Jan 1 11:04

$ who am i
host!tux1      tty2          Jan 1 11:04

But:
$ whoami
tux1
```

© Copyright IBM Corporation 2012

Figure 3-15. Who is on the system

LX027.2

Notes:

To find out who is logged onto your system, you can enter one of the following two commands:

- The **who** command shows you the user's ID and display where the user logged in, date and time the user logged in, and (if a network is used) the host name the user logged in from.
- The **who am i** and **whoami** commands show you what user you logged in to the system as.

Finding information about users

- The **finger** command shows info about other users.
- Synopsis: **finger [user] [@host]**

```
$ finger
Login      Name      Tty      Idle      Login Time
tux1        Tux (1)   2          Jan 1    11:04
root        root      *1          7        Jan 1    11:10

$ finger tux1
Login:
tux1
Name:
Tux (1)
Directory: /home/tux1 Shell: /bin/bash
On since Mon Jan 1  11:04 (UTC) on tty2
No mail.
No plan.
```

© Copyright IBM Corporation 2012

Figure 3-16. Finding information about users

LX027.2

Notes:

The **finger** command shows you additional information about a user, for instance, yourself. The **finger** command shows you the user ID, full name, display, idle time, date and time the user logged in, and some office information. The asterisk (*) in the output of the finger command indicates that the issuer of the **finger** command cannot write to this (tty1) device.

Note that the **finger** service is by default disabled in most distributions, because it can also be accessed over the network and is considered a security risk: through the finger service, an intruder can easily obtain a list of all usernames on the system, and determine which accounts have been inactive for a while. This makes breaking in easier, and breaking into an inactive account is less likely to be noticed. How to install and enable the finger service is covered later in this course.

The clear, echo, write, and wall commands

- The **clear** command clears your screen.

```
$ clear
```

- The **echo** command writes messages to your own screen.

```
$ echo I want to go to lunch
```

```
I want to go to lunch
```

- Use **write** to display a text message on a user's terminal.

```
$ write tux2
```

```
Join me for lunch?
```

```
<ctrl-d>
```

- Use **wall** to place a message on all logged-in users' displays.

```
$ wall
```

```
We are back from lunch.
```

```
<ctrl-d>
```

© Copyright IBM Corporation 2012

Figure 3-17. The clear, echo, write, and wall commands

LX027.2

Notes:

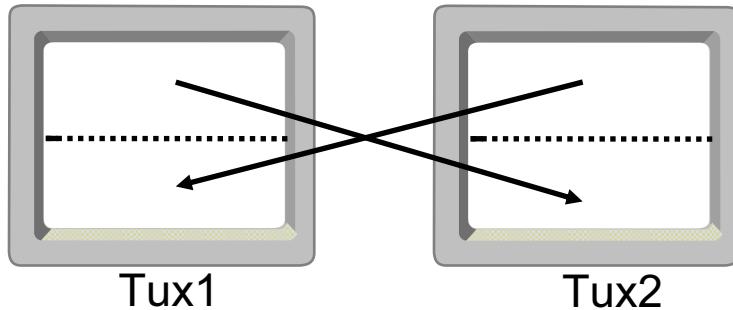
The **clear** command clears your screen.

The **echo** command writes a message to your own screen, while the **write** command writes messages to screens of other users.

The **wall** command finally writes a message to all screens.

Talk with another user

- If Tux1 wants to talk with Tux2, Tux1 enters:
 - \$ talk tux2
- If Tux2 also wants to talk with Tux1, Tux2 enters:
 - \$ talk tux1



- To disable **write**, **wall** and **talk** reception, use **mesg n**

© Copyright IBM Corporation 2012

Figure 3-18. Talk with another user

LX027.2

Notes:

talk is one of the earliest programs that allowed you to directly communicate to another user without the use of a telephone. It was later succeeded by programs like **IRC**, **ICQ** and **MSN**.

To prevent messages originating from programs like **write**, **wall** and **talk** reaching your terminal, use the **mesg n** command. To enable these messages again, use **mesg y**. Note that the root user is always able to **write** or **wall** to your terminal screen.

Keyboard tips

- | | |
|----------------------|--------------------------------|
| • <arrow up/down> | Previous/next command |
| • <arrow left/right> | One character left/right |
| • <Shift PgUp/PgDn> | Scroll window up/down |
| • <Tab> | Command or filename completion |
| • <Ctrl-R> | Search for command in history |
| • <Ctrl-C> | Terminate current command |
| • <Ctrl-D> | End of transmission/input/file |

© Copyright IBM Corporation 2012

Figure 3-19. Keyboard tips

LX027.2

Notes:

The bash shell keeps track of all commands that are entered (including options and arguments) in the "command history". Likewise, a text terminal or graphical terminal emulator keeps track of all the output of the commands as it appeared on screen.

Using the keyboard we can recall these command and output histories.

First, with your arrow keys you can scroll through the command history and thus retrieve, modify and re-execute previous commands. Ctrl-R even allows you to search in the command history.

Second, using Shift-PgUp and Shift-PgDn you can scroll through the terminal output history.

Another key you need to know about is the <Tab> key. This key is used for command completion (if used on the first argument on the command line) and filename completion (if used on any other arguments): Simply type the first few letters of the command or filename you're trying to type, and hit <Tab>. If the letters that you typed unambiguously lead to a command or filename, the whole command or filename will be substituted. If there is

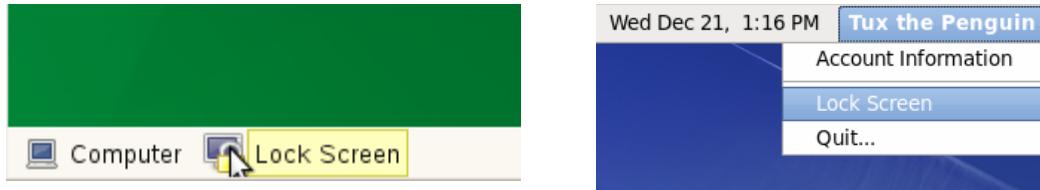
ambiguity, the command or filename will be completed as far as possible. In this case, hit <Tab> twice to get a list of all possible commands or filenames that fit the first few letters that you typed.

<Ctrl-C> is the key combination that's most often used to terminate a running command. Technically speaking, it sends a signal to the command that is supposed to cause it to stop. Signals will be covered in a later unit.

<Ctrl-D> is used as the end-of-transmission/input/file indicator. It is useful to indicate to certain programs that you've reached the end of what you wanted to send to this program. One example is that <Ctrl-D> is a signal to the shell itself to indicate you've finished working with it and want to leave the shell - effectively logging you out.

Locking

- When temporary leaving a system alone, always lock your terminal
- In a text mode terminal, use **vlock** (lock your terminal) or **vlock -a** (lock the whole console)
- In a graphical environment, use the menu, the “padlock” icon or **xlock**
 - Most screensavers support automatic locking too
- A locked terminal can only be unlocked with the users password



© Copyright IBM Corporation 2012

Figure 3-20. Locking

LX027.2

Notes:

When you’re going to be away from your system for a short period of time, lock your terminal. The easiest way of breaking in, after all, is finding a door which is already open.

Locking a text mode terminal can be accomplished with the **vlock** command. The **vlock -a** command not only locks your virtual terminal, but the whole console. Most distributions, unfortunately, do not install **vlock** by default.

In a graphical environment, use the menu options, the padlock icon or the **xlock** command. Most screensavers support locking too.

A locked terminal can only be unlocked with the users password¹.

¹ Occasionally you will find a configuration where you can also unlock a terminal with the **root** password. This requires the **xlock** command to run with root privileges though, and is generally considered a security risk.

Logging out

- When finished working on a system, always log out
- In a text mode terminal, use **logout**, **exit** or **Ctrl-D**
- In a graphical environment, use the menu

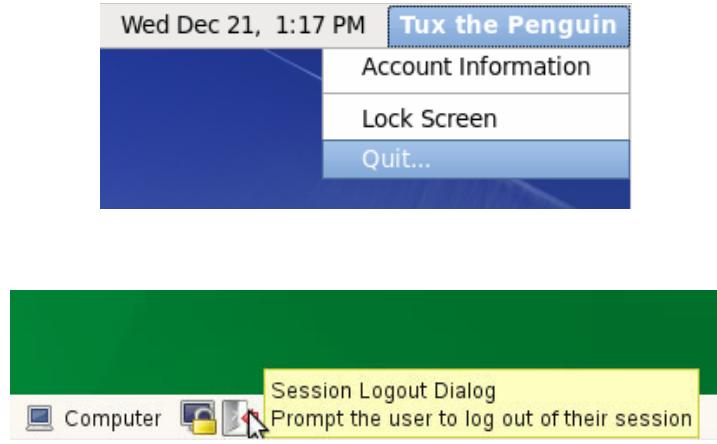


Figure 3-21. Logging out

LX027.2

Notes:

After you have finished using a system, always log out.

To log out of a text mode session, you can use the **logout** or **exit** commands, or use the hotkey sequence **<Ctrl-D>**.

To log out of a graphical environment, use the appropriate menus or buttons.

Unit review

- Linux is a multi-user, multi-tasking operating system.
- You always need to log in before using the system.
- You can login via a text virtual terminal, a graphical console or via the network.
- Everything on a Linux system can be done by typing commands.
- The command prompt indicates that the system is ready to accept commands.
- All commands have a common syntax.
- Sample commands include **passwd**, **date**, **cal**, **who**, **finger**, **clear**, **echo**, **write**, **wall** and **talk**

© Copyright IBM Corporation 2012

Figure 3-22. Unit review

LX027.2

Notes:

Checkpoint

1. True or False: A Linux system always needs a graphical user interface.

2. Which of the following commands is not a legal command in Linux?
 - a. ls/dev/bin
 - b. ls -al/dev/bin
 - c. ls -a -l .
 - d. ls -a-l/dev

3. How do you switch between virtual terminals?

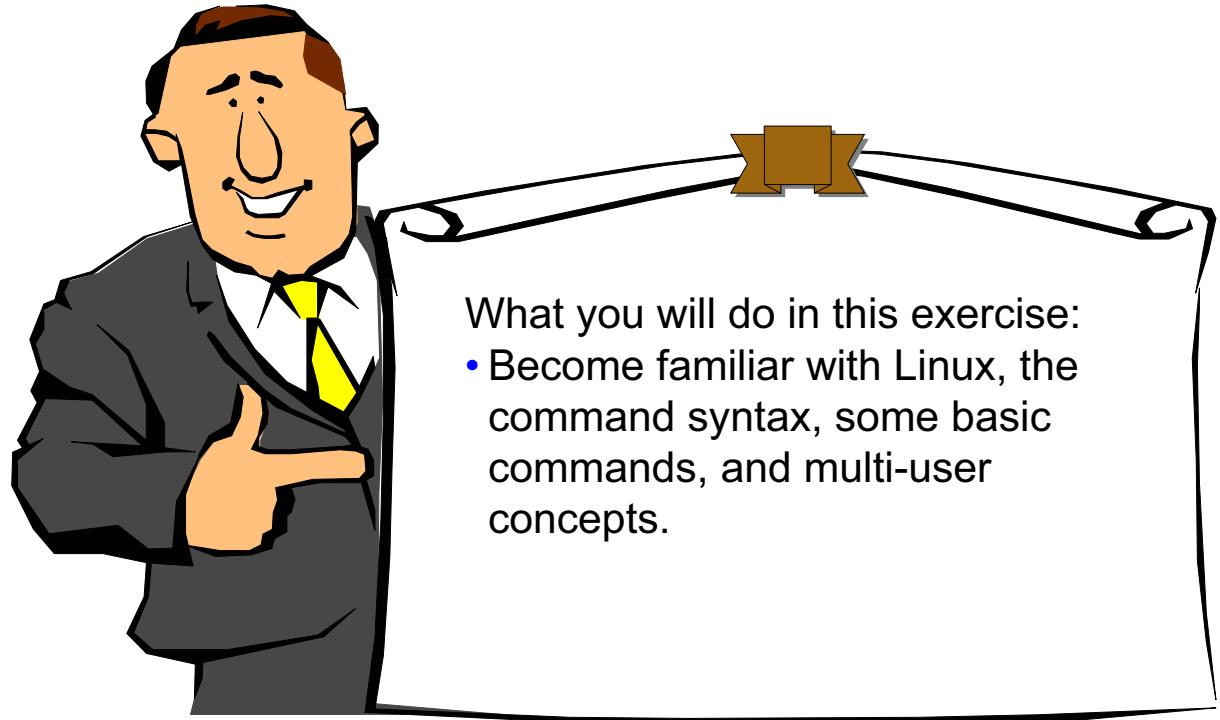
© Copyright IBM Corporation 2012

Figure 3-23. Checkpoint

LX027.2

Notes:

Exercise: Using the system



© Copyright IBM Corporation 2012

Figure 3-24. Exercise: Using the system

LX027.2

Notes:

Unit summary

Having completed this unit, you should be able to:

- Log in and out of the system
- State the structure of Linux commands
- Execute basic Linux commands
- Use Linux commands to communicate with other users
- Use the keyboard and mouse effectively
- Discuss the X Window System

© Copyright IBM Corporation 2012

Figure 3-25. Unit summary

LX027.2

Notes:

Unit 4. Working with files and directories

What this unit is about

This unit describes different file types and file names and path names. This unit will also illustrate how to create, delete, copy, and move directories and files as well as how to view the content of both text and binary files.

What you should be able to do

After completing this unit, you should be able to:

- Describe the different file types
- Describe file and path names
- Create, delete, copy, move, and list directories
- Create, delete, copy, and move files
- View the content of both text and binary files

How you will check your progress

- Checkpoint questions
- Machine exercises

Unit objectives

After completing this unit, you should be able to:

- Describe the different file types
- Describe file and path names
- Create, delete, copy, move, and list directories
- Create, delete, copy, and move files
- View the content of both text and binary files

© Copyright IBM Corporation 2012

Figure 4-1. Unit objectives

LX027.2

Notes:

A file

- A file is:

```
A_collection_of_data\nA_stream_of_characters_or_a_"byte_stream"\nNo_structure_is_imposed_on_a_file_by_the_operating_system\n
```

- `\n` is a newline character.
- `_` is a space character.

© Copyright IBM Corporation 2012

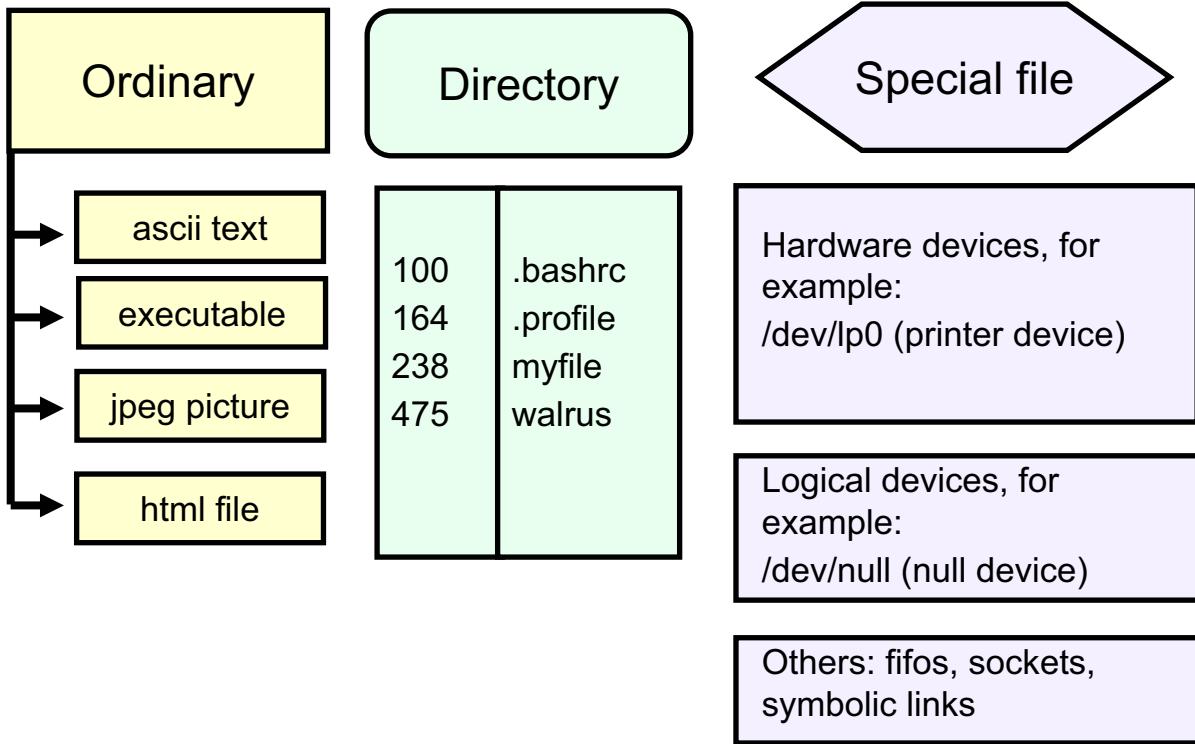
Figure 4-2. A file

LX027.2

Notes:

Linux imposes no internal structure on a file's content. The user is free to structure and interpret the contents of a file in whatever way is appropriate.

File types



© Copyright IBM Corporation 2012

Figure 4-3. File types

LX027.2

Notes:

An ordinary file can contain either text or code data. Text files are readable by a user and can be displayed or printed. Code data, also known as binary file, is readable by the computer. Binary files can be executed.

Directories contain information the system needs to access all types of files, but they do not contain the actual data. Each directory entry represents either a file or subdirectory. Special files usually represent devices used by the system. A very useful special file is `/dev/null`, which can be used as a sort of trash bin for unwanted output.

Linux file names

- Should be descriptive of the content
- Should use only alphanumeric characters
 - Uppercase, lowercase, number, @, _
- Should not include embedded blanks
- Should not contain shell metacharacters
 - * ? > < / ; & ! | \ ` ' " [] () { }
- Should not begin with plus sign (+) or minus sign (-)
- Are case sensitive
- Are hidden if the first character is a period (.)
- Can have a maximum of 255 characters

© Copyright IBM Corporation 2012

Figure 4-4. Linux file names

LX027.2

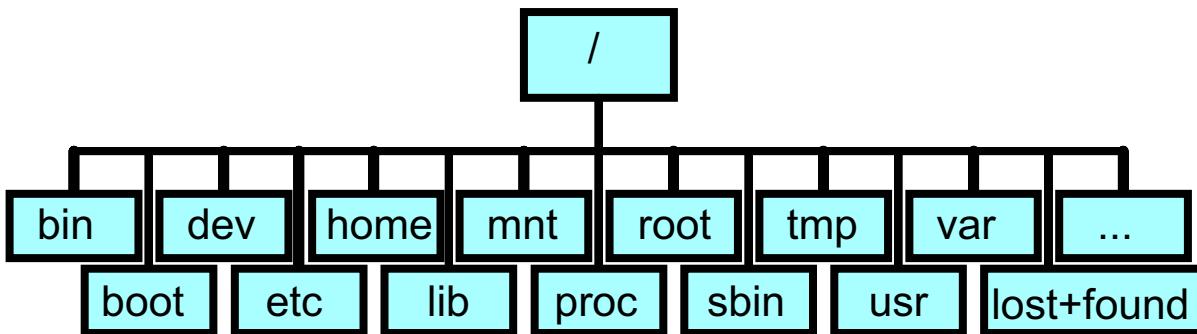
Notes:

In general, the naming of files is free in Linux. Linux for instance does not force filenames into an 8.3 format like MS-DOS did. Extensions in general have no value to the shell. They might be relevant for applications, though. Theoretically, every character on your keyboard can be used in a filename. But since the shell interprets various characters as metacharacters, it is best to stick to lowercase and uppercase letters, digits, the underscore and the at sign. Other characters, such as embedded blanks and metacharacters should preferably not be used. If you encounter them, you need to quote them properly, (discussed later).

The dot is a special case: Anywhere in the filename, it is simply used as part of the file name, except when the dot is the first character in the file name. If a file name begins with a dot (for example, .bash_profile), the file is considered to be a hidden file and does not show up when you enter the **ls** command, for example. Note that the **ls -a** command will show all files, including hidden ones.

Directory structure

- All Linux directories are contained in one virtual, unified file system.
- Physical devices and network shares are mounted on mount points.
 - USB flash drives
 - Hard disk partitions
 - Optical disk drives
 - NFS volumes, Windows shares
- There are no drive letters like A:, C:, and so on.



© Copyright IBM Corporation 2012

Figure 4-5. Directory structure

LX027.2

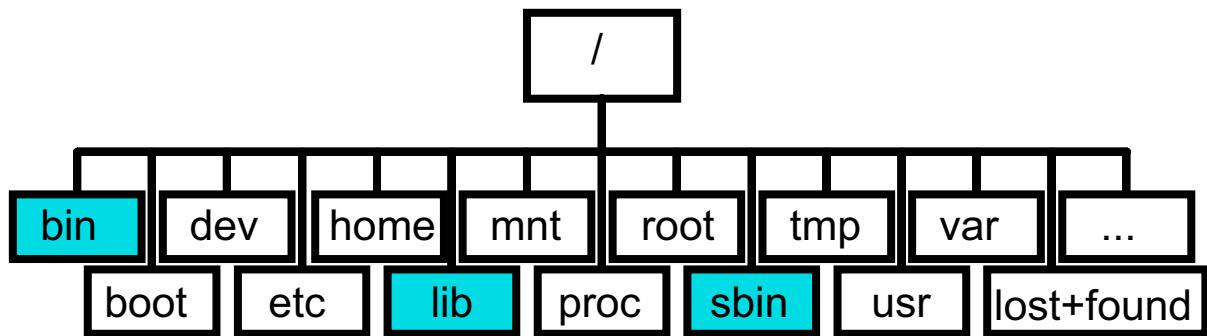
Notes:

All Linux directories and files are contained in one virtual unified file system. This means that all physical devices with file systems on them (floppy disks, hard disk partitions, optical disk drives) are all combined into one giant tree structure. Among other things, this means that Linux does not use drive letters, such as A:, C:, and so forth.

For the layout of this unified file system, most Linux distributions closely follow the Linux File System Hierarchy Standard, a collaborative document that defines the names and locations of many files and directories. The standard can be viewed at <http://www.pathname.com/fhs>.

The standard closely follows the conventional UNIX file system, with some minor modifications. In the next few graphics, you can view the contents of the various directories.

/bin, /lib, and /sbin



- `/bin` contains executables for every user.
- `/sbin` contains system administration executables.
- `/lib` contains libraries.
- They should always be available:
 - At system boot
 - In single user mode
 - When booting from rescue disk

© Copyright IBM Corporation 2012

Figure 4-6. `/bin`, `/lib`, and `/sbin`

LX027.2

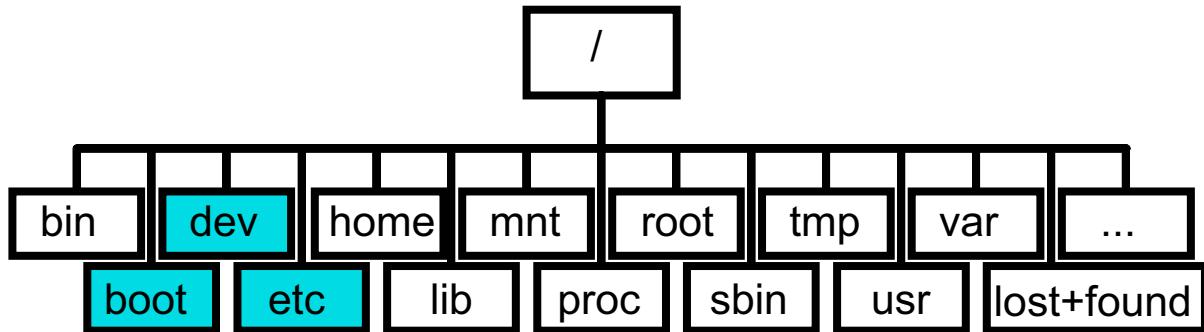
Notes:

`/bin`, `/lib`, and `/sbin` contain executables and libraries which always need to be available, even in the worst of scenarios, because these tools are essential for system maintenance and recovery.

The difference between `/bin` and `/sbin` is in the people who use them. `/bin` is for everybody, and `/sbin` is typically tools needed only by the system administrators. Therefore, you cannot find `/sbin` in the search path (`$PATH`) of a normal user, but you can in the path of a system administrator.

Libraries are shared parts of code, which are available to every program that might want to use the code. Because different programs use the same routines, for example, writing to the screen, it saves disk space and effort to put these routines into one central library. Otherwise, you might need to put them into individual programs.

/boot, /dev, and /etc



- /boot contains kernel image and some other goodies
- /dev contains device references
- /etc contains system-wide configuration files

© Copyright IBM Corporation 2012

Figure 4-7. /boot, /dev, and /etc

LX027.2

Notes:

The /boot directory contains the kernel images, some other things related to these images, and the files needed for the bootloader (LILO or GRUB).

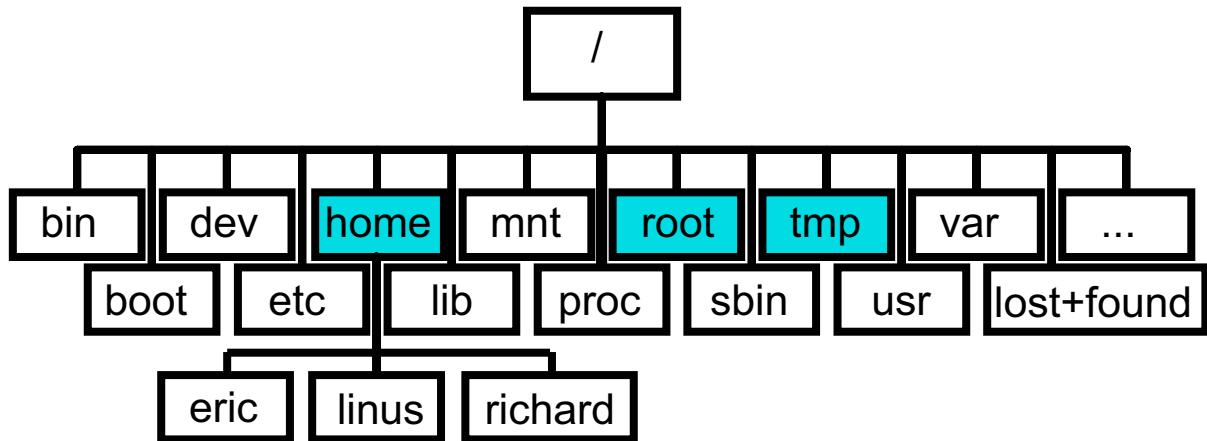
The /dev directory contains special files that represent the hardware of your system. By writing to these devices or reading from or both, you can usually (but not always) interact almost directly with the hardware. Note, however, that this generally is not a particularly safe thing to do. Access to these devices is, therefore, usually restricted to root.

There are two types of special files in the /dev directory. (Note that two other types of special files exist too: Named pipes and UNIX sockets. These are generally not located in /dev, but in /tmp.)

/etc contains the system-wide configuration files. These files apply to each and every program that is running, and each, and every user.

Some programs or subsystems create their own subdirectory in /etc because they have more than just a few configuration files and want to keep these files together. As an example, /etc/x11 contains configuration files specifically for the X Window System.

/home, /root, and /tmp



- /home contains the home directories of users.
- /root is the home directory of the root user.
- /tmp is used for temporary storage space.

© Copyright IBM Corporation 2012

Figure 4-8. /home, /root, and /tmp

LX027.2

Notes:

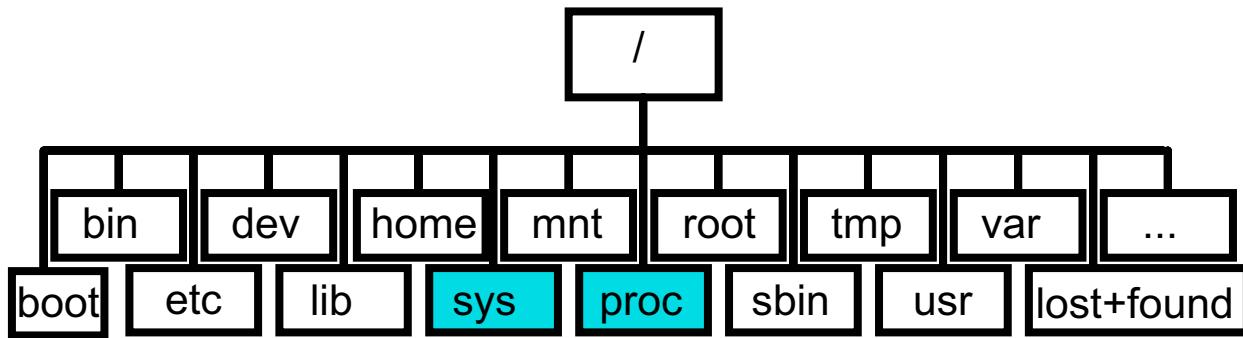
/home contains the home directories of the users. Within home, each user has its own directory, identified by the username, for instance, /home/tux1.

/root is the home directory of the root user.

/tmp is used as a temporary storage space for programs and users. Temporary in this context means a couple of minutes, hours at most, instead of days and weeks.

Some system administrators have automatic cleanup jobs running every night that clean /tmp of files older than a few days.

/proc and /sys



- These are virtual file systems.
- /proc represents kernel and process information.
- /sys represents driver and file system information.

© Copyright IBM Corporation 2012

Figure 4-9. /proc and /sys

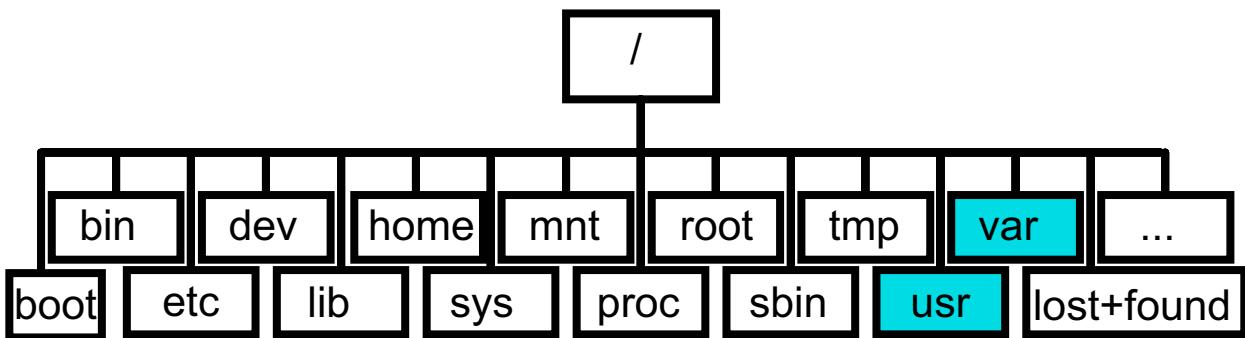
LX027.2

Notes:

/proc is a virtual file system that exists only in the imagination of the kernel. It is used for accessing the kernel's data structures, (for instance, the interrupts, imports, and so on), and for accessing process information.

/sys is also a virtual file system. It acts as a counterpart to /proc. It provides detailed information about kernel status to userspace including which modules/drivers that are loaded and specifics about the file system.

/usr and /var



- `/usr` contains all the user programs that the system needs.
- `/var` contains data that is changed when the system is running normally. It is specific for each system, that is, not shared over the network with other computers.
- Logs often reside on `/var`, and they can get voluminous.

© Copyright IBM Corporation 2012

Figure 4-10. /usr and /var

LX027.2

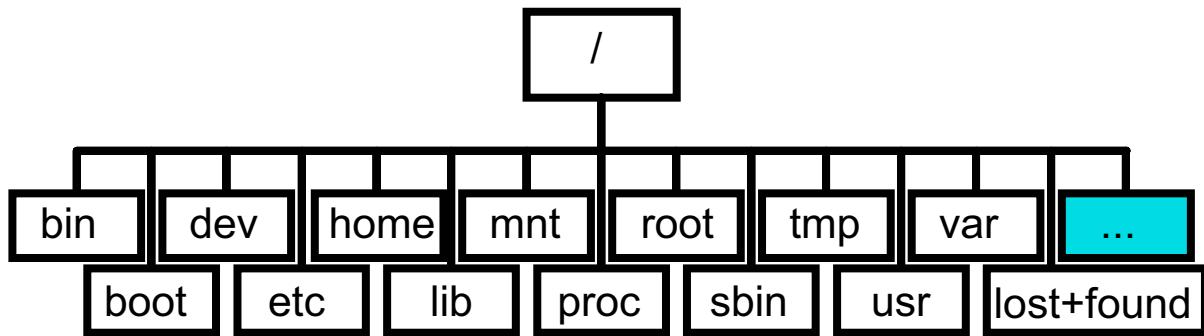
Notes:

`/usr` is by far the largest directory on a freshly installed Linux system. It contains all the programs that need to be available on the system, but need not be available at boot time or in an emergency.

`/var` holds files that might vary greatly in size. Typical examples of these files include log files, usually stored in `/var/log`. But other applications also generate files of which the size or contents vary greatly. These are then also stored somewhere in `/var`.

`/var/tmp` is sometimes also used as temporary storage space, just like `/tmp`, but with a longer retention period (weeks or more).

Other directories in /



- /opt used for some software from external providers
 - Separate file system advisable
- Whatever you create for yourself

© Copyright IBM Corporation 2012

Figure 4-11. Other directories in /

LX027.2

Notes:

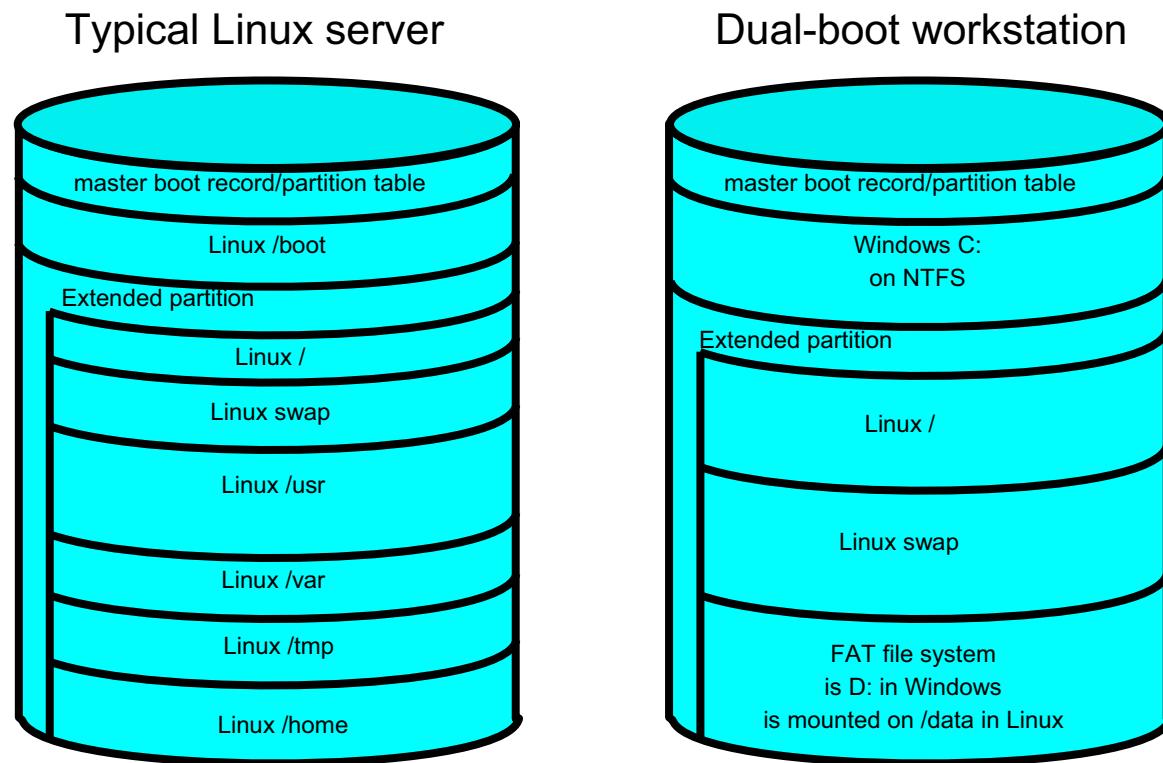
Various other directories might be present in the root of your file system. It is of course up to you to decide which directories to create and what to store in them. On a SUSE system, subsystems such as GNOME, KDE, and OpenOffice are installed in /opt.

/mnt is usually used as a placeholder for all the mount points you need for mounting non-standard file systems. For example: /mnt/windows for a Windows partition.

/media is commonly used as a placeholder for the mount points for various media: /media/cdrom, /media/floppy, and so forth.

Every file system has a lost+found directory, which is created when you create the file system. It is normally empty; however, should a system crash occur when the file system is not in a stable state, **fsck** (file system-check) checks the file system and places any files that do not have a name in lost+found. The system administrator then must decide who that file belonged to and find a way to give it back to the rightful owner.

Typical file system layout



© Copyright IBM Corporation 2012

Figure 4-12. Typical file system layout

LX027.2

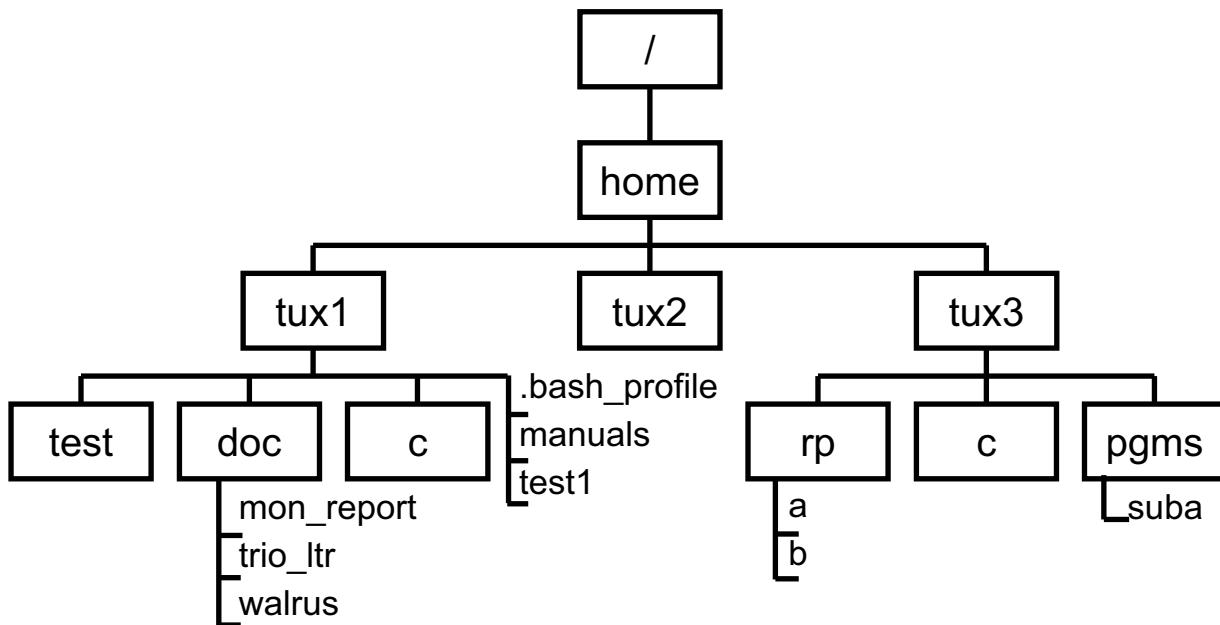
Notes:

The file system layout reflects the expected usage of the system.

A typical Linux server has its storage divided into various partitions as needed. It has separate file systems for each of the main directories. These file systems can then be backed up individually, can have different quota applied, have different mount permissions and so forth.

For comparison, if you had a workstation with a dual-boot configuration, it would have all its Linux-specific data in one big root file system. All Windows-specific data is in one big NTFS formatted Windows file system, and all data that needs to be shared across both platforms sits in a separate, FAT-formatted file system which can be accessed by both.

Example directory structure



© Copyright IBM Corporation 2012

Figure 4-13. Example directory structure

LX027.2

Notes:

This example directory structure is used in the rest of the unit.

Linux path names

Full path names:
Start from / (the root directory)

Relative path names:
Start from the present working directory

- Examples (working directory is /home/tux1):
 - /home/tux1/doc/mon_report (full)
 - doc/mon_report (relative)
 - ../tux3/pgms/suba (relative)
 - ./test (file in the current dir)
 - ~/test (file under home directory)

© Copyright IBM Corporation 2012

Figure 4-14. Linux path names

LX027.2

Notes:

The path name is written as a string of names separated by slashes (not back slashes like in DOS, OS/2, or Windows). The rightmost name can be any type of file (ordinary, directory, or special). The other names must be directories.

A path name is always considered to be relative unless it begins with a slash (/). An absolute path name or full path name always starts with a slash.

Where am I?

- The **pwd** command (Print Working Directory) can be used to find out what your current working directory is.

```
$ pwd  
/home/tux1  
$
```

© Copyright IBM Corporation 2012

Figure 4-15. Where am I?

LX027.2

Notes:

The **pwd** command always returns the full path name of your (current) present working directory. It is not a bad idea to use this command often, especially when you are removing files (to be sure you are removing them from the correct directory). Note that most distributions, by default, configure a shell prompt, which lists the last part of the current directory as well.

Change current directory

- With the **cd** (Change Directory) command:

— \$ cd dir_name

```
$ cd doc           (relative)
$ cd /home/tux1/doc      (full)
$ cd ~tux1/doc        (home)

$ cd      (Go to your home directory)
$ cd ..   (Go one directory up)
$ cd -    (Go to previous directory)
```

© Copyright IBM Corporation 2012

Figure 4-16. Change current directory

LX027.2

Notes:

Using the **cd** command with nothing after it automatically returns you to your home directory. This is the directory into which you are usually placed when you log in.

Create directories

- With the **mkdir** (Make Directory) command:

— \$ **mkdir** dir_name

```
$ mkdir /home/tux1/doc (full pathname)

$ cd /home/tux1
$ mkdir doc           (relative pathname)
```

© Copyright IBM Corporation 2012

Figure 4-17. Create directories

LX027.2

Notes:

The **mkdir** command creates one or more new directories specified by the dir_name parameter. Each new directory contains the standard entries . (dot) and .. (dot dot).

The -m option can be used with the **mkdir** command to specify the directory being created with a particular set of permissions.

Removing directories

- With the **rmdir** (Remove Directory) command:
 - \$ rmdir dir_name

```
$ pwd  
/home/tux1  
$ rmdir doc test  
rmdir: doc: Directory not empty  
$
```

directory must be empty!

© Copyright IBM Corporation 2012

Figure 4-18. Removing directories

LX027.2

Notes:

The **rmdir** command lets you delete directories. You get no message if the command is successful. It never hurts to follow a command such as this with an **ls**, which is discussed on the next page, to make sure that you have accomplished what you set out to do.

It is also practical to check if you are in the correct directory with **pwd**, before you try to remove something.

Working with multiple directories

- Create and remove multiple directories simultaneously with the -p flag.

```
$ mkdir -p dir1/dir2/dir3  
$ rmdir -p dir1/dir2/dir3
```

© Copyright IBM Corporation 2012

Figure 4-19. Working with multiple directories

LX027.2

Notes:

The **mkdir** `dir1/dir2/dir3` command generates an error message if neither `dir1` nor `dir2` exists. To overcome this problem, you could use the `-p` option with **mkdir**. If `dir1` and `dir2` already exist, only `dir3` is created.

The `-p` option with **rmdir** first removes `dir3`, then `dir2`, and finally the `dir1` directory. If a directory is not empty, you are in it, or you do not have the right permissions to it when it is removed, the command terminates.

List the contents of directories

- With the **ls** command:

— `ls [dir/file]`

```
$ ls /home  
tux1 tux2 tux3
```

Important options:

- l long listing (more information)
- a lists all files (including hidden)
- t lists files sorted by change date
- R lists contents recursively

© Copyright IBM Corporation 2012

Figure 4-20. List the contents of directories

LX027.2

Notes:

The **ls** command is used to list the contents of a directory, and has many useful options. If no file or directory name is given as an argument, the current directory is used.

By default, the **ls** command displays the information in alphabetic order. When the **ls** command is executed it does not display any file names that begin with a dot (.), unless the -a option is used. These files are generally referred to as hidden files, for this reason.

The touch command

- The **touch** command creates an empty file or updates the modification time of an existing file.

```
$ ls -l  
-rw-rw-r-- 1 tux1 penguins 512 Jan 1 11:10 docs  
  
$ touch docs  
$ ls -l  
-rw-rw-r-- 1 tux1 penguins 512 Jan 1 15:37 docs  
  
$ touch new  
$ ls -l  
-rw-rw-r-- 1 tux1 penguins 512 Jan 1 15:37 docs  
-rw-rw-r-- 1 tux1 penguins 0 Jan 1 15:38 new
```

© Copyright IBM Corporation 2012

Figure 4-21. The touch command

LX027.2

Notes:

The **touch** command serves two purposes:

- If the file specified by the file name does not exist, a zero length (empty) file is created. This is useful because some programs in Linux only use the fact that a file exists to perform a certain action or not. The contents are then not important. An example of this is a lockfile. Suppose you have a special program that can only run once on a system. It is not allowed to run two instances. To ensure that this program is only started once, the program first checks if a certain file exists. If it does exist, it terminates itself. If it does not yet exist, it creates the file and starts working. When the program is terminated by the user, it also deletes the file. This effectively ensures that the program can only be started once.
- If the file does exist, the last modification time (displayed with `ls -l`) is updated to reflect the current date and time. This can be useful to force a backup of a file for instance, when only incremental backups are made.

The `-t` parameter allows you to specify a time and date. This makes it possible to give a file any date and time you like.

Copying files (1 of 2)

- The **cp** command copies files.
 - `cp source[s] [target]`

```
Copying one file to another:
```

```
$ cp .bashrc bashrc.old
```

```
Copying multiple files into a target directory:
```

```
$ cp doc/mon_report doc/walrus /tmp
```

© Copyright IBM Corporation 2012

Figure 4-22. Copying files (1 of 2)

LX027.2

Notes:

Pay attention to the two ways you can use the **cp** command. The first syntax copies a file from one directory to another directory. The second syntax is if you enter more than two parameters to **cp**. This means that the first n parameters (which represent files) are copied to the last parameter, which represents a directory.

When using the **cp** command, if the file specified as the target file already exists, then the copy operation writes over the original contents of the file without warning. To avoid this, use `cp -i` (interactive copy).

If you are copying more than one file in one operation, the specified target must be a directory. If the target is a directory, the copy has the same name as the original file.

`cp -R` can be used to recursively copy all files, subdirectories, and the files in those directories to a new directory.

Copying files (2 of 2)

- **cp** can recursively copy directories with the **-R** flag.

```
$ cp -R /home/tux1/doc /tmp
```

To prevent **cp** from overwriting existing files, use:

```
$ cp -R -i /home/tux1/doc /tmp
cp: overwrite `/tmp/doc/walrus`?
```

© Copyright IBM Corporation 2012

Figure 4-23. Copying files (2 of 2)

LX027.2

Notes:

When using the **cp** command, if the file specified as the target file already exists, the copy operation writes over the original contents of the file without warning. To avoid this, use **cp -i** (interactive copy).

If you are copying more than one file in one operation, the specified target must be a directory. If the target is a directory, the copy has the same name as the original file.

cp -R can be used to recursively copy all files, subdirectories, and the files in those directories to a new directory.

Moving and renaming files (1 of 2)

- With the **mv** command:

— `mv source[s] [target]`

To move a file do another directory:

```
$ mv doc/walrus ../../tmp
```

To rename a file:

```
$ mv doc documents
```

Use the **-i** option to prevent **mv** from overwriting existing files!

© Copyright IBM Corporation 2012

Figure 4-24. Moving and renaming files (1 of 2)

LX027.2

Notes:

The **mv** command is used to move files from one directory to another. The syntax is `mv source target`. The **mv** command can also be used to rename files. The source can be a file or a list of files. If the source is a list of files, then the target must be a directory.

The target can be a file or a directory. **Caution:** If the target is the name of a file that already exists and if you have the correct permissions set for that file and directory, you overwrite the file and never get an error message. To avoid this, use `mv -i`, an interactive move, which prompts you if there are duplicate names. As a result of the **mv** you still have the same number of files as you did before. Furthermore, all the attributes remain the same.

Moving and renaming files (2 of 2)

- Moving and renaming files can be combined by **mv**.

```
$ cd  
$ pwd  
/home/tux1  
$ mv /tmp/walrus      ./test/walrus2
```

To move a directory:

```
$ mv ./test /tmp
```

mv is recursive by default

© Copyright IBM Corporation 2012

Figure 4-25. Moving and renaming files (2 of 2)

LX027.2

Notes:

If the source is a directory instead of a file, **mv** attempts to move this whole directory to the new location, with all the files in it. Earlier versions of **mv** would only do this if the source and target directory were on the same file system. Current versions of **mv** now perform a **cp** and **rm** automatically if you try to move directories between file systems.

The main difference between **mv** and **cp** is in what happens with the update time, inode number, and so forth. **mv** retains the inode number and the inode. **cp** creates an entirely new file with new inode information, changing only the file name and access time.

Removing files

- You can move files with the **rm** command.

```
$ rm test/walrus2
$ ls test/walrus2
ls: rob: No such file or directory

If unsure, use -i option
$ rm -i test/walrus2
rm: remove `test/walrus2`?

To remove files and directories recursively:
$ rm -ir test/
```

© Copyright IBM Corporation 2012

Figure 4-26. Removing files

LX027.2

Notes:

The **rm** command removes the entries for the specified file or files from a directory. Note that the **rm** command by default does not require confirmation from the user². For the interactive version of the command, use the **-i** option.

The **-r** option permits recursive removal of directories and their contents if the directory is specified. Be careful when using this option, as it does not require the directory to be empty for this option to work.

The **-f** (force) option prevents error messages and does not ask the user for confirmation.

Listing file contents

- With the **cat** (Concatenate) command:

```
$ cat file1 file2 ...  
  
$ cat walrus  
"The time has come", the walrus said,  
"To talk of many things:  
Of shoes - and ships - and sealing wax -  
Of cabbage - and kings -  
And why the sea is boiling hot -  
And whether pigs have wings."  
$
```

© Copyright IBM Corporation 2012

Figure 4-27. Listing file contents

LX027.2

Notes:

If the output of the **cat** command is longer than a screen, the file scrolls until the bottom of the file is reached. Thus, you may only be able to read the last full screen of information. The **cat** command can be used to copy two files into one file. The syntax is:

```
cat file1 file2 > new_file
```

Displaying files page by page

- With the **more** or **less** commands:

```
$ less walrus
"The time has come", the walrus said,
"To talk of many things:
Of shoes - and ships - and sealing wax -
Of cabbage - and kings -
And why the sea is boiling hot -
And whether pigs have wings."
/tmp/test/walrus 1-6/6 (END)
```

© Copyright IBM Corporation 2012

Figure 4-28. Displaying files page by page

LX027.2

Notes:

The **more** and **less** commands reads the file names specified and displays the contents of the files one page at a time. Use the space bar to view the next page and the **b** key to view previous one.

To search for patterns in the file which is displayed, use the / (forward slash) key. To repeat a search, use **n**. The advantage of **less** over **more** is that **less** can also scroll backwards if its input is received from a pipe. Another difference is that **less** **clears** the screen when done, while **more** keeps the content of the last page on screen.

Use the **q** to end the **more** and **less** commands. (**less** does not terminate itself when the end of the file is reached.)

Displaying binary files

- With the **od** command:

```
$ od /usr/bin/passwd
0000000 042577 043114 000401 000001 000000 000000 000000 000000
0000020 000002 000003 000001 000000 107300 004004 000064 000000
0000040 051430 000000 000000 000000 000064 000040 000006 000050
$
```

- With the **strings** command:

```
$ strings /usr/bin/passwd
/lib/ld.so.1
__gmon_start__
__deregister_frame_info
__register_frame_info
...
$
```

© Copyright IBM Corporation 2012

Figure 4-29. Displaying binary files

LX027.2

Notes:

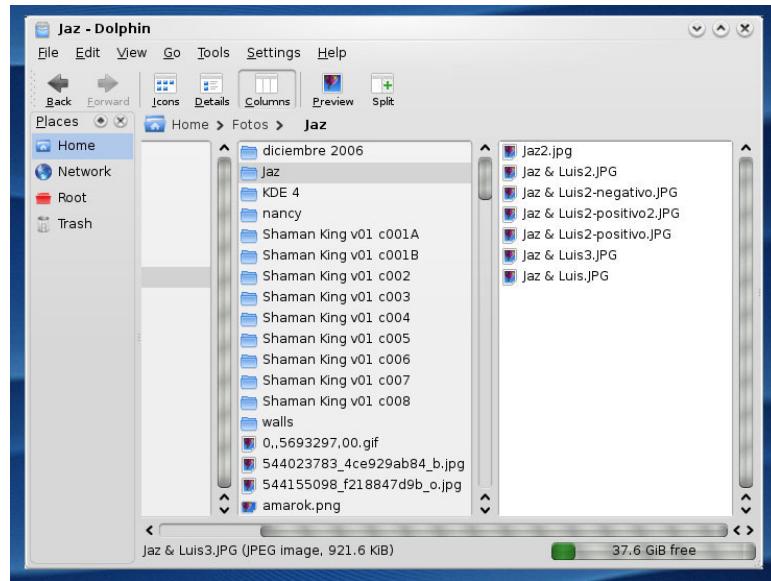
od (Octal Dump) allows you to view the contents of a binary file byte by byte. Without any option, **od** shows the contents of the file in octal format, but you can also specify the **-c** option for decimal display, or the **-h** option for hexadecimal display. Other output options also exist.

strings is also a very handy tool to take a peek at binary files. It displays only all the strings that occur in a binary file. (A string is a combination of at least four consecutive, contiguous, printable ASCII characters.)

An alternative for **od** is **hexdump**. It basically does the same thing, but in a slightly more readable format. It is not available on all distributions by default, however.

File managers

- Linux also offers different graphical file managers.
 - Nautilus (GNOME)
 - Konqueror (KDE)



© Copyright IBM Corporation 2012

Figure 4-30. File managers

LX027.2

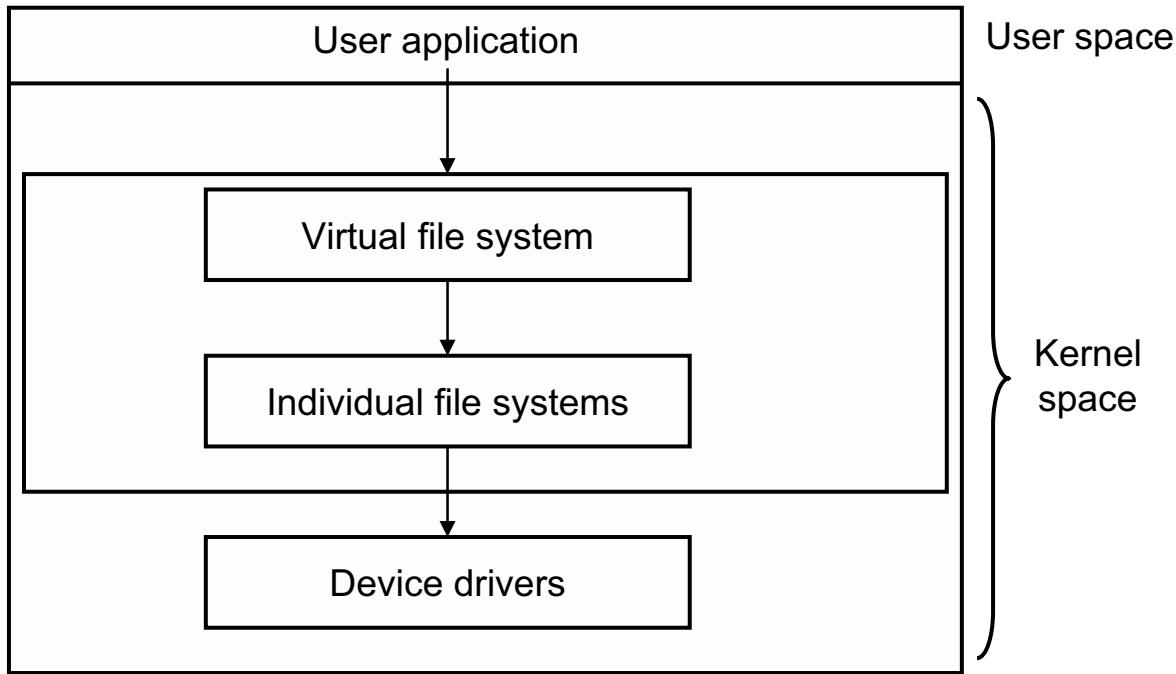
Notes:

When you are in a graphical environment, you do not need to use these commands (although they are not that hard if you are used to them), but you can use one of the several graphical file managers that are available for your desktop environment to perform file operations.

Both the GNOME and KDE file managers have the capability to recognize the type of file that you are working with and, if appropriate, allow you to do something with the content as well. If you click a .tar.gz file (a compressed file archive), for instance, it automatically uncompresses the file and opens the archive. You can then copy the files out of the archive.

Virtual unified file system (1 of 2)

- Linux does not use drive letters (A:, C:, D:) to identify drives and partitions, but creates a virtual, unified file system.



© Copyright IBM Corporation 2012

Figure 4-31. Virtual unified file system (1 of 2)

LX027.2

Notes:

The virtual file system (VFS) is the primary interface to the underlying file systems. This component exports a set of interfaces and then abstracts them to the individual file systems, which may behave very differently from one another. Each individual file system implementation, such as ext2, JFS, and so on, exports a common set of interfaces that is used (and expected) by the VFS.

The VFS layer provides added flexibility that allows individual file systems to be dynamically added or removed on a Linux system.

Virtual unified file system (2 of 2)

- Different drivers and partitions are mounted on a mountpoint.
- Mounting associates a storage device with a file system.

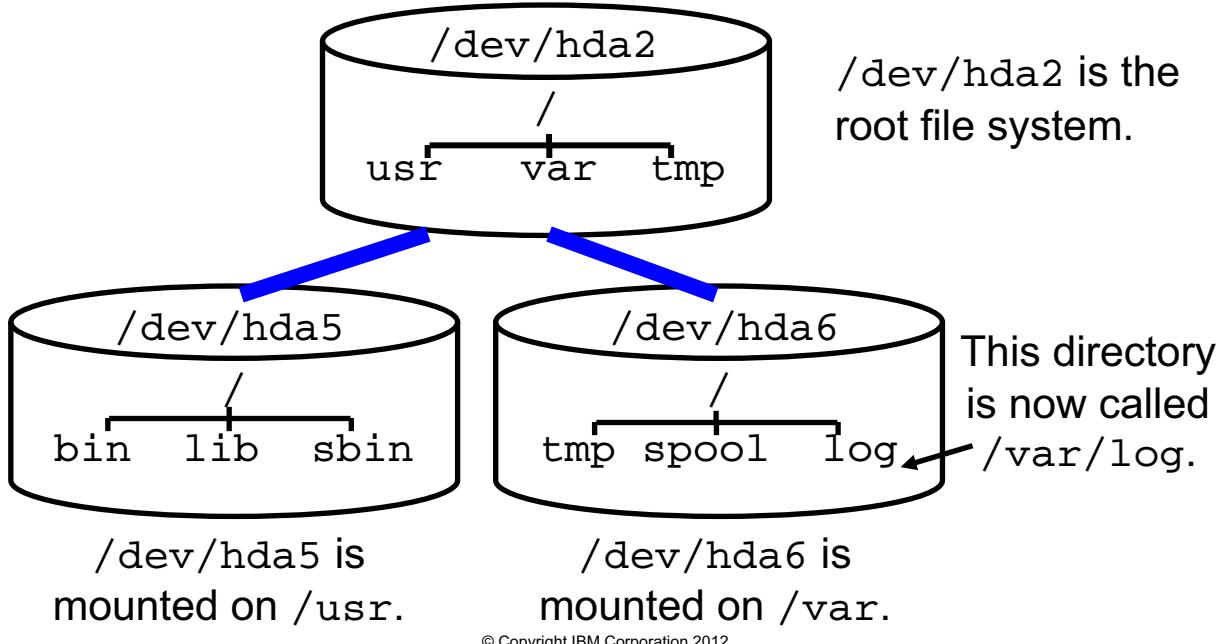


Figure 4-32. Virtual unified file system (2 of 2)

LX027.2

Notes:

The virtual unified file system model that Linux uses allows all known file systems to be *mounted* on top of each other in a huge, virtual file system. *Mounting* associates a storage device with a file system. This offers transparency to users, makes system administration easier, and makes it possible to support far more than 26 different file systems simultaneously.

The first file system is called the *root file system* and is mounted by the kernel itself, when the kernel starts. In addition to regular directories and data, this file system also contains a number of empty directories, which are used as mount points for other file systems. As an example, take a look at `/dev/hda6` in the visual. It is a fully contained file system, with its own directories and files. One of the directories is called `log`. When this file system is mounted on the `var` directory in the root file system, the `log` directory now becomes available in our virtual file system hierarchy as `/var/log`. But you also could have mounted `/dev/hda6` on, let's say, the mount point (empty directory) `variable` in the root file system (`/dev/hda2`). Then all of a sudden, the `log` directory would have become available as `/variable/log`.

There are several reasons for creating multiple file systems, and then mounting these on top of each other:

- It makes it easier to do partial and incremental backups.
- It allows you to set different disk space quota per file system.
- It allows you to split your data over multiple disks and makes migration of data between disks easier.
- It allows you to apply different security settings (such as read-only) to different file systems.
- It allows you to mount certain file systems over the network.

Therefore, on important servers, you typically see multiple (sometimes even hundreds) of file systems, all mounted on top of each other.

There are only a few directories in the root file systems that cannot be a separate file system:

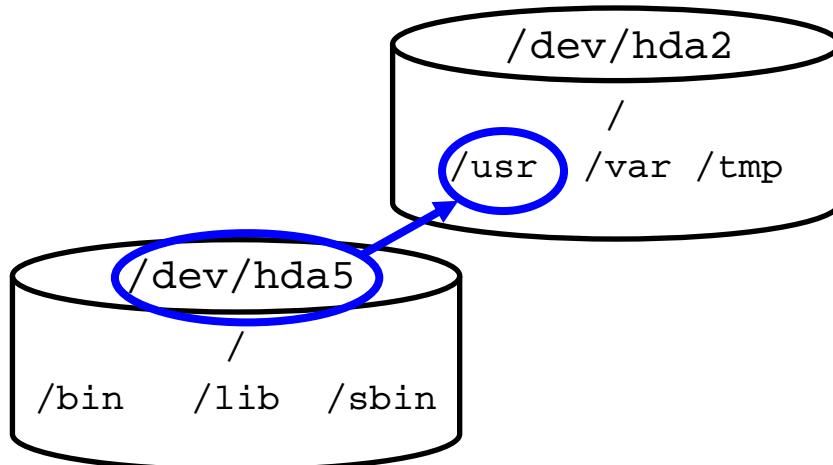
- /root
- /bin
- /lib
- /sbin
- /etc
- /dev

All other directories are candidates to become a separate file system. To determine which scheme is useful, you need to attend the System Administration class.

The mount command

- The **mount** command mounts a file system.
 - Makes it part of the unified file system structure
 - `mount [-t type] [-o opts] device mountpnt`

```
# mount /dev/hda5 /usr
```



© Copyright IBM Corporation 2012

Figure 4-33. The mount command

LX027.2

Notes:

The **mount** command is used to mount a file system on a mount point. It can best be compared to attaching a branch to a tree. After a file system (a disk partition, a CD-ROM device or a floppy disk, for instance) has been mounted, it has become part of the unified file system and can be accessed.

The syntax of the mount command is:

```
mount [-t type] [-o options] device mountpoint
```

The *type* specifies the file system type, for example ext2, vfat or iso9660. The *options* might indicate read-only, for instance. The *device* identifies the device name, which contains the file system to be mounted. The *mount point* identifies the (empty) directory where the file system should be mounted.

As an example, suppose you want to mount the `/dev/hda5` partition on the `/usr` mountpoint. The command to use then becomes:

```
mount /dev/hda5 /usr
```

The umount command

- The **umount** command unmounts a file system.
 - It takes it out of the unified file system structure.
 - The file system should not be busy.
 - `umount {device|mountpnt}`

```
# umount /dev/hda5
- OR -
# umount /usr
```

© Copyright IBM Corporation 2012

Figure 4-34. The umount command

LX027.2

Notes:

Unmounting a file system is done with the **umount** command. This command needs only one argument, which is either the device name or the mount point of the file system to be unmounted.

Unmounting a file system can only be done if the file system is no longer in use. A file system is in use when one of the following three conditions is true:

- A user currently has a file opened on the file system.
- A user is currently running a program from that file system.
- A user uses a directory on that file system as its current working directory.

The /etc/fstab file

- /etc/fstab lists all known file systems on the system.
- Syntax:
 - device mountpoint type options dump fsck
- File systems with the noauto option are not mounted automatically but can be used as templates for mount.

```
# cat      /etc/fstab
/dev/hda1  /mnt/winC      vfat    defaults        0  0
/dev/hda2  /           ext3    defaults        1  1
/dev/hda5  /usr          ext3    defaults        1  2
/dev/hda6  /var          ext3    defaults        1  2
/dev/cdrom /media/cdrom  iso9660 noauto,owner,ro  0  0
/dev/fd0   /media/floppy auto    noauto,owner    0  0
none      /proc          proc    defaults        0  0
none      /dev/pts       devpts  gid=5,mode=620   0  0
```

Note: Some distributions use file system labels instead of device names!

© Copyright IBM Corporation 2012

Figure 4-35. The /etc/fstab file

LX027.2

Notes:

The /etc/fstab file lists all known file systems on the system. On a large server, this might add up to dozens of file systems, each with their own mountpoint and special options. We're not going to cover the file in great depth, but you need to be aware of it at least. The file lists every file system known to the system on a single line, where each line consists of six columns:

- The first column identifies the device name.
- The second column identifies the mount point.
- The third column identifies the file system type.
- The fourth column identifies the options.
- The fifth column is a parameter for the **dump** program.
- The sixth column is a parameter for the **fsck** program.

All file systems listed in this file will be mounted automatically at the corresponding mountpoint when the system boots, except for the file systems which have the **noauto** option. As you can see in the visual, this applies to /mnt/cdrom and /mnt/floppy.

Mounting and unmounting removable media

- Most distributions configure /etc/fstab so that the console user is allowed to mount removable media (floppy, CD-ROM) on a predetermined mountpoint and with predetermined options (for security).
- Always unmount media before ejecting.
- The GUI typically mounts media automatically or nearly so.

```
$ whoami
tux1
$ mount /media/cdrom
$ mount
.
/dev/cdrom on /media/cdrom type iso9660 (ro,nosuid,nodev,user=tux1)
.
$ ls /media/cdrom
.

$ umount /media/cdrom
```

© Copyright IBM Corporation 2012

Figure 4-36. Mounting and unmounting removable media

LX027.2

Notes:

As you have seen in other visuals, both the floppy and CD-ROM can be predefined in the /etc/fstab file, but have the noauto option set, so they're not automatically mounted when the system boots. These entries are nevertheless useful, because the **mount** command uses this file to complete the information about a file system if only a partial **mount** command is given: The only command needed to mount a CD-ROM is:

```
mount /media/cdrom
```

which, based on the /etc/fstab file, will be expanded by the **mount** command to

```
mount -t iso9660 -o owner,ro /dev/cdrom /media/cdrom
```

The owner option by the way allows a regular user to perform the mount command. When this option is set, you don't need to be root on the system to mount that particular device at that particular mountpoint. It's enough to be logged in on the console.

Hard links and soft (symbolic) links

- A *hard link* associates another file with an existing inode.
 - You cannot create a hard link for a directory or across file systems.
 - The file is not removed until all hard links to the file are removed.
- *Soft links* are like shortcuts to files or directories.
 - You can link to directories and across file systems.
 - They becomes useless when you remove the target file.

```
$ ln FileA FileB
$ ls -il FileA FileB
8986669 -rw-r--r-- 2 test test 200 2010-04-22 15:15 FileA
8986669 -rw-r--r-- 2 test test 200 2010-04-22 15:15 FileB

$ ln -s FileB FileC
$ ls -il FileB FileC
8986669 -rw-r--r-- 2 test test 200 2010-04-22 15:15 FileB
8986670 lrwxrwxrwx 1 test test      5 2010-04-22 15:16 FileC -> FileB
```

© Copyright IBM Corporation 2012

Figure 4-37. Hard links and soft (symbolic) links

LX027.2

Notes:

Linux has a few helpful tools that allow files to reference other files. This is useful for shortcuts, organization, etc. These tools are called file linking. There are two types of file links: *hard link* and *soft or symbolic links*.

Hard links: Hard links create a new file entry that actually points to an existing inode, (of an existing file). The new file can have a different name, but it will share an inode and inode number with the existing file. This means it is essentially the same file, but with maybe a different name and location. The actual file data is not removed from the system until all references to it are removed as well.

Symbolic links: Soft or symbolic links allows you to create a new small file, this time with its own inode and inode number that contains the name and path of another file the system knows about. The new file becomes a link or shortcut to the existing file.

Hard links cannot be used to link to a directory and cannot be linked across file systems. Symbolic links can link to directories and to file across other file systems. Symbolic links, however, become useless when their target file is deleted.

Links are created using the **ln** command.

Hard links use the syntax:

```
ln targetFile linkedFile
```

Symbolic links use the syntax:

```
ln -s targetFile symLinkFile
```

Unit review

- There are three types of files.
 - Ordinary
 - Directory
 - Special
- The Linux file system structure is a hierarchical tree.
- Files are accessed using either full or relative path names. A full path name always begins with a forward slash (/).
- The following commands can be used with directories: **pwd**, **cd**, **mkdir**, **rmdir**, **touch**, and **ls**.
- The following commands can be used with files: **cat**, **more**, **less**, **cp**, **mv**, **rm**, **touch**, **od**, and **strings**.

© Copyright IBM Corporation 2012

Figure 4-38. Unit review

LX027.2

Notes:

Checkpoint

1. True or False: Linux imposes an internal structure on a regular file (not a directory or special file).

2. Which of the following is not a legal file name?
 - a. ~tux1/mydocs.tar.gz
 - b. /home/tux1/mydoc(1)
 - c. /var/tmp/.secret.doc
 - d. /home/./home/tux1/one+one

3. What command would you use to copy the file /home/tux1/mydoc to /tmp and rename it at the same time to tempdoc?

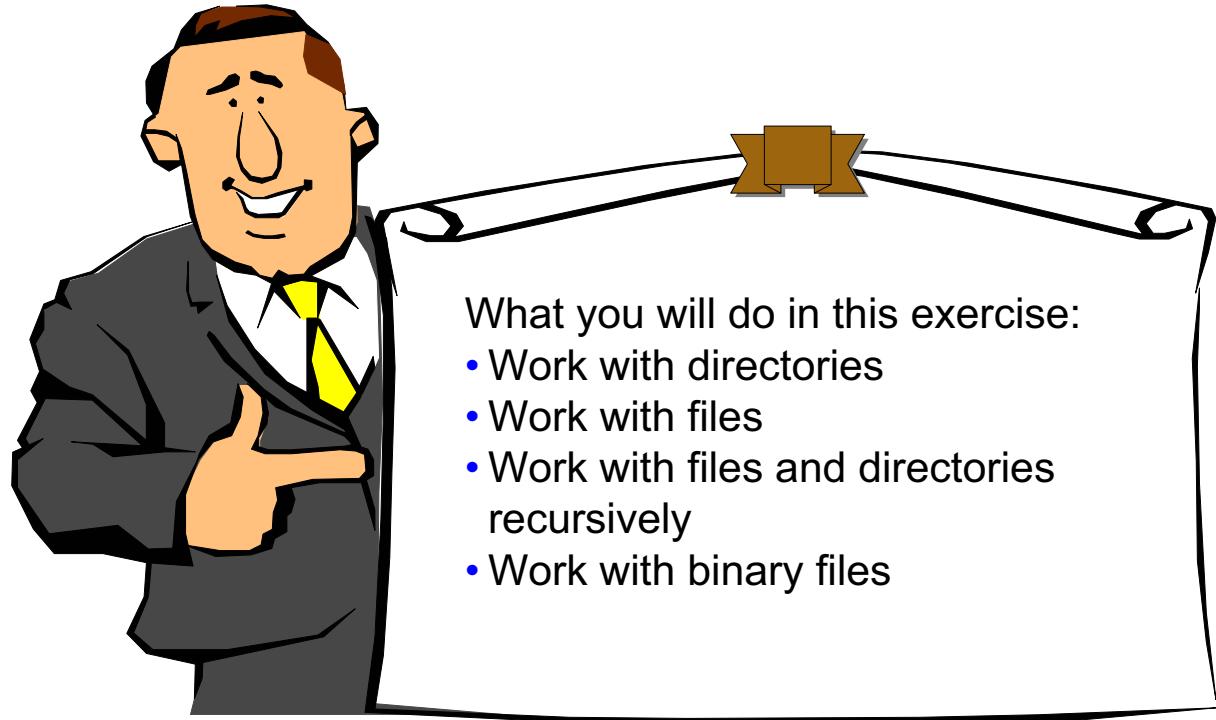
© Copyright IBM Corporation 2012

Figure 4-39. Checkpoint

LX027.2

Notes:

Exercise: Working with files and directories



© Copyright IBM Corporation 2012

Figure 4-40. Exercise: Working with files and directories

LX027.2

Notes:

Unit summary

Having completed this unit, you should be able to:

- Describe the different file types
- Describe file and path names
- Create, delete, copy, move, and list directories
- Create, delete, copy, and move files
- View the content of both text and binary files

© Copyright IBM Corporation 2012

Figure 4-41. Unit summary

LX027.2

Notes:

Unit 5. File and directory permissions

What this unit is about

This unit describes how permissions are used, lists permissions required to perform common commands, demonstrates how to change permissions using symbolic and octal notation, and explains how default permissions are calculated.

What you should be able to do

After completing this unit, you should be able to:

- Describe how permissions are used
- List the permissions required to perform several common commands
- Change permissions using symbolic and octal notation
- Describe how default permissions are calculated

How you will check your progress

- Checkpoint questions
- Machine exercises

Unit objectives

After completing this unit, you should be able to:

- Describe how permissions are used
- List the permissions required to perform several common commands
- Change permissions using symbolic and octal notation
- Describe how default permissions are calculated

© Copyright IBM Corporation 2012

Figure 5-1. Unit objectives

LX027.2

Notes:

Users and groups

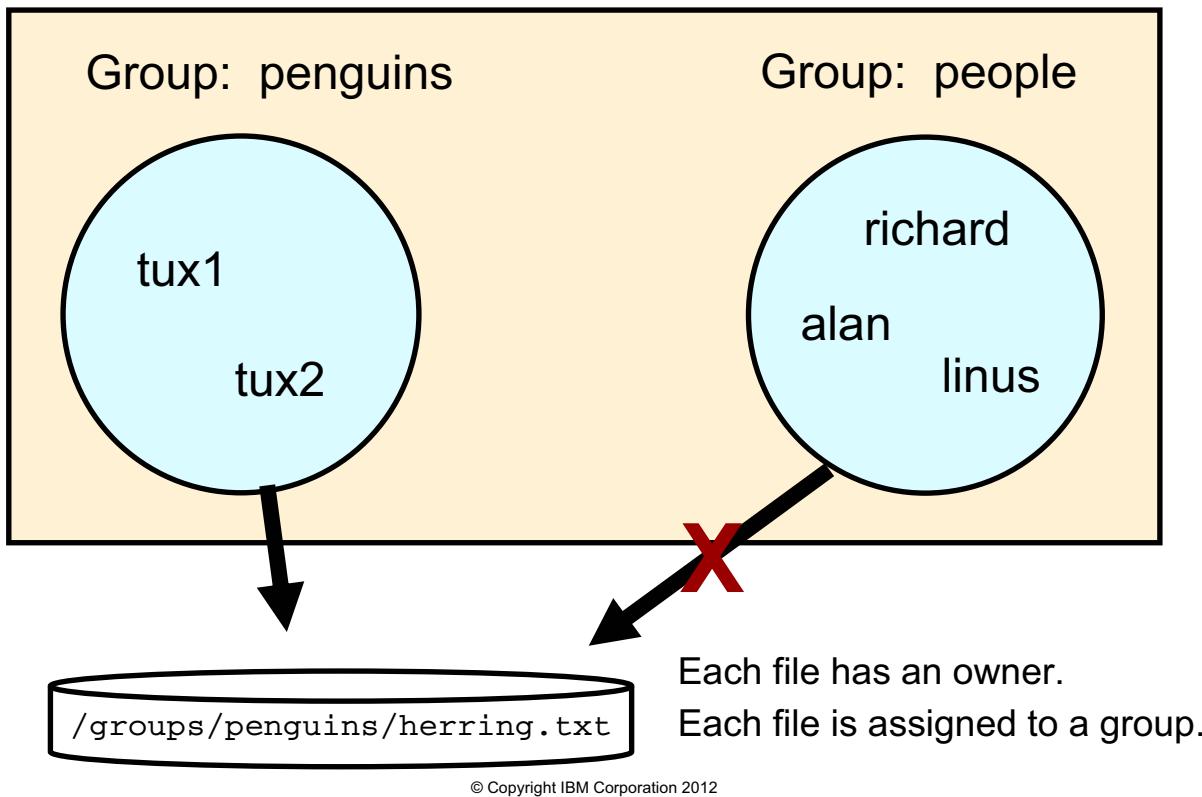


Figure 5-2. Users and groups

LX027.2

Notes:

To protect your files from other users, Linux allows you to set permissions on files and directories. As an example, you might want to protect your own files from all other users on the system, or make sure that certain system files can only be read by users, and not written to.

Permissions also work on a group level, allowing you to create files that are accessible only to users that are members of a particular group, while members of other groups would not have access.

Permissions

- File permissions are assigned to:
 - The owner of a file
 - The members of the group the file is assigned to
 - All other users
- Permissions can only be changed by the owner and root!



© Copyright IBM Corporation 2012

Figure 5-3. Permissions

LX027.2

Notes:

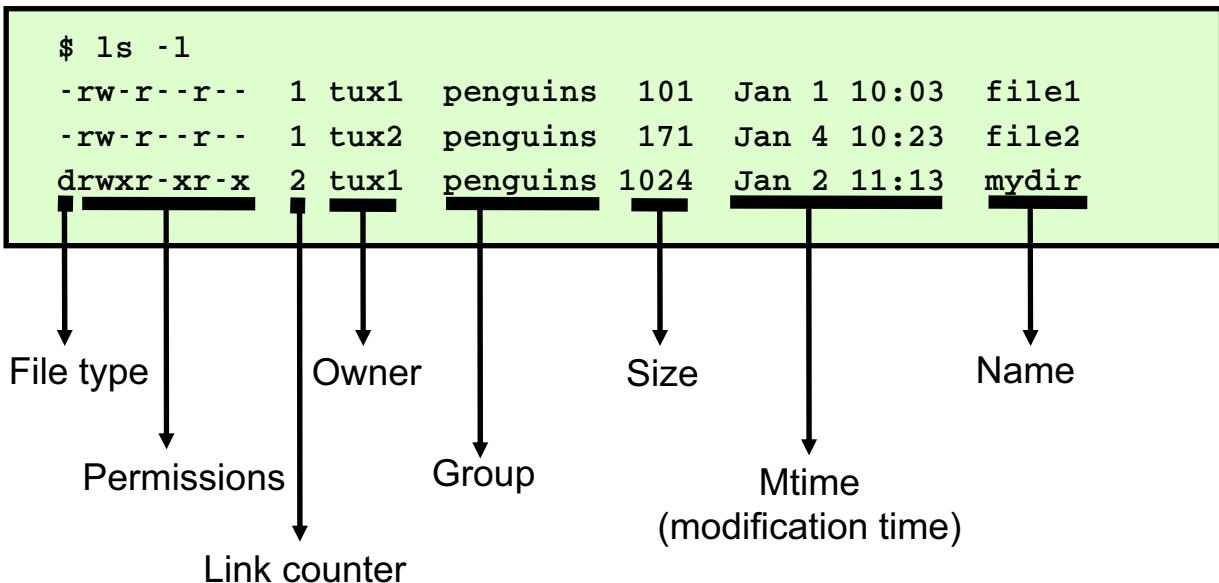
Permissions under Linux are configured for each file and directory. There are three levels of permissions:

1. The permissions that apply to the owner of the file. The owner of a file is by default the user that created the file.
2. The permissions that apply to all members of the group that is associated with the file.
3. The permissions that apply to all other users on the system.

Permissions can only be changed by the file owner and root.

Viewing permissions (command line)

- To show the permissions of a file, use the **ls** command with the **-l** option.



© Copyright IBM Corporation 2012

Figure 5-4. Viewing permissions (command line)

LX027.2

Notes:

To view the permissions that are currently applied to a file, use the **ls -l** command. Note that, when viewing the permissions on a directory, you cannot use the command **ls -l <directoryname>** because that command lists the contents of the directory and not the directory itself. Instead, use the command **ls -ld <directoryname>**.

Some distributions, including Red Hat, have "Access Control Lists" (ACLs) enabled. This allows more extended permission groups than just user/group/others. If the ACL functionality is enabled in your distribution, all permission bits will be followed by either a "." (dot) or a "+" (plus). A dot means that this particular file/directory does not have any additional access controls, while a plus means that it has. You can then view the additional access controls with the **getfacl** command. ACLs will be covered at the end of this unit.

Permissions notation

rwxrwxrwx	<table border="0"> <tr> <td>r</td> <td>Read</td> </tr> <tr> <td>w</td> <td>Write</td> </tr> <tr> <td>x</td> <td>Execute</td> </tr> </table>	r	Read	w	Write	x	Execute
r	Read						
w	Write						
x	Execute						
owner group other							

Regular files

- r File is readable.
- w File is writeable.
- x File is executable (if in an executable format).

Directories

- r Contents of directory can be listed (**ls**).
- w Contents can be modified add/delete files).
- x Change into directory is possible (**cd**).

© Copyright IBM Corporation 2012

Figure 5-5. Permissions notation

LX027.2

Notes:

For a file, these permissions mean the following:

- **Read:** Allow the user to read the contents of the file, for instance, with **cat** or **less**.
- **Write:** Allow the user to modify the contents of the file, for instance, with **vi**.
- **Execute:** Allow the user to execute the file as a program, provided that the file is indeed an executable program (such as a shell script).

For a directory, these permissions have a slightly different meaning.

- **Read:** Allow the user to view the contents of the directory, for instance, with **ls**.
- **Write:** Allow the user to modify the contents of the directory. In other words, allow the user to create and delete files and to modify the names of the files. **Note:** Having write permissions on a directory thus allows you to delete files, even if you have no write permissions on that file!
- **Execute:** Allow the user to use this directory as its current working directory. In other words, allow the user to **cd** into it.

Required permissions

Command	Source directory	Source file	Target directory
<code>cd</code>	x	N/A	N/A
<code>ls</code>	x, r	N/A	N/A
<code>mkdir, rmdir</code>	x, w	N/A	N/A
<code>cat, less</code>	x	r	N/A
<code>cp</code>	x	r	x, w
<code>cp -r</code>	x, r	r	x, w
<code>mv</code>	x, w	None	x, w
<code>vi</code>	x, r	r, w	N/A
<code>rm</code>	x, w	None	N/A

© Copyright IBM Corporation 2012

Figure 5-6. Required permissions

LX027.2

Notes:

The visual shows the permissions that are required for certain common Linux commands. Note that you do not need write permissions on a file for moving or removing that file. This sounds strange at first, but is easily explained: the name of a file is actually stored in a directory. So if you have write permissions on a directory, you can change the name of all files in that directory, or remove files from that directory.

Who can change permissions?

- The owner of the file or directory
- The root user

```
$ ls -ld /home/tux1
drwx----- 4 tux1 penguins 1024 Jan 5 12:43 /home/tux1
```

© Copyright IBM Corporation 2012

Figure 5-7. Who can change permissions?

LX027.2

Notes:

You can see who the owner is by looking at the output of the `ls -l` command. The third column lists the owner's name. If you are working as the listed user (or root) you are allowed to change permissions on the listed file (or directory).

Changing permissions (1 of 2)

- To change the permission of a file use the **chmod** command
- Syntax: **chmod <MODE> <FILE [S]>**
- Mode can be symbolic.

```
$ chmod go-rx /home/tux1
$ ls -ld /home/tux1
drwx----- 4 tux1    penguins 1024 Jan 5 12:43 /home/tux1
```

- Mode can also be octal.

```
$ chmod 700 /home/tux1
$ ls -ld /home/tux1
drwx----- 4 tux1    penguins     1024 Jan 5 12:43 /home/tux1
```

© Copyright IBM Corporation 2012

Figure 5-8. Changing permissions (1 of 2)

LX027.2

Notes:

Changing permissions is done with the **chmod** command. This command supports two different ways of writing down the required permissions: symbolic and octal. Symbolic notation describes the permissions using the following syntax:

chmod <who operator what> <filename(s)>

<who> can be:

- *u* for the owner (user) of the file
- *g* for the group assigned to the file
- *o* for all other users
- *a* for all (owner+group+others)

<operator> can be:

- A plus sign (+) to add permissions
- A minus sign (-) to delete permissions
- An equals sign (=) to clear all permissions and set to the permissions specified

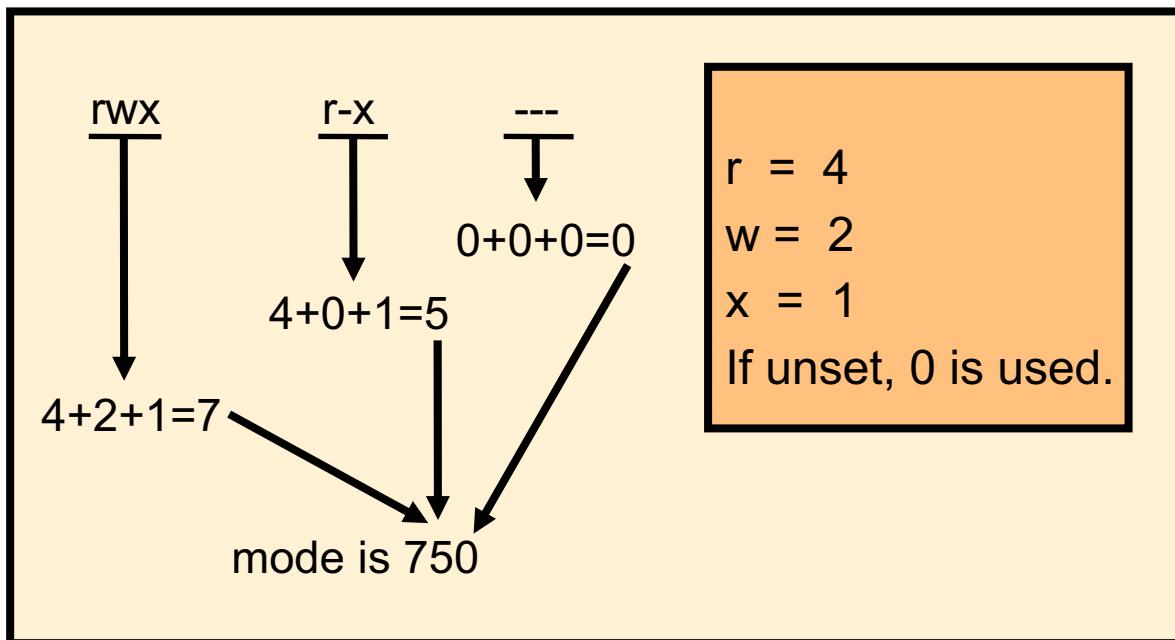
<what> can be any combination of *r*, *w*, and *x*.

Legal commands are, for instance:

- `chmod u=rw,go=r <file>` gives the user read-write permissions and gives read permissions to the group and others. All previous permissions are discarded.
- `chmod a+x` gives everybody execute permissions in additions to the permissions they already had.

With octal notation, permissions are identified with an octal number.

Changing permissions (2 of 2)



© Copyright IBM Corporation 2012

Figure 5-9. Changing permissions (2 of 2)

LX027.2

Notes:

The visual shows how octal numbers are calculated. Octal permissions might seem incredibly complex when compared to symbolic notation; however, in actual practice there are only a few permission combinations that make sense.

When applying these permissions, experienced Linux users do not have to think about them anymore, but apply them blindly. These combinations are:

- 600 (rw-----): For private files
- 700 (rwx-----): For private programs and directories
- 644 (rw-r--r--): For files that you want to be readable by others
- 755 (rwxr-xr-x): For programs and directories you want to be readable/executable by others
- 666 (rw-rw-rw-): For public writable files (does not happen often)
- 777 (rwxrwxrwx): For public writable directories, such as /tmp

umask

- New files should not be created with 666! To avoid this problem, a permission mask exists.

Regular files:

Default permissions	rw-rw-rw	666
umask (-)	----w--w-	022
Resulting permissions	rw-r--r--	644

Directories:

Default permissions	rwxrwxrwx	777
umask (-)	----w--w-	022
Resulting permissions	rwxr-xr-x	755

Syntax: umask 022

© Copyright IBM Corporation 2012

Figure 5-10. umask

LX027.2

Notes:

It is obviously important to know with what permissions new files and directories are created. Under Linux, it is not really easy to tell, since the default permissions can be modified by setting a umask (with the **umask** command).

If no umask were set (which never happens, by the way), a file would always be created with permissions 666 (rw-rw-rw-) and a directory would get 777 (rwxrwxrwx). In actual practice, however, a umask is set, and this number is subtracted from these permissions.

Therefore, with a umask of 022, the default permissions for a file becomes 644 (rw-r--r--, 666-022) and the default permissions for a directory becomes 755 (rwxr-xr-x, 777-022). The default umask depends on your distribution, and whether your distribution uses something called User Private Groups.

- Red Hat assigns a umask of 002 to regular users and 022 to root.
- SUSE assigns a umask of 022 to all users, including root.

User Private Groups and the reasons for the different umasks are beyond the scope of this course. They are covered in the LX03 course.

Access Control Lists

- Access Control Lists allows more permission bit groups than user/group/others
- ACL functionality needs to be enabled by the distribution
 - Can be enabled per-filesystem if needed
- Set ACLs with **setfacl**
 - For instance **setfacl -m u:tux2:rw ourfile**
- List ACLs with **getfacl**
- Various other utilities (for example **ls**) have been modified to work with ACLs

© Copyright IBM Corporation 2012

Figure 5-11. Access Control Lists

LX027.2

Notes:

At some systems you might also be confronted with Access Control Lists (ACLs). ACLs are a way of assigning permissions to more groups than just one user, one group and others. With ACLs, you can assign different permissions to multiple users and multiple groups.

Setting ACLs is done with the **setfacl** command, while displaying ACLs is done with the **getfacl** command. In addition to this, tools such as **ls** have been modified to work with ACLs. On an ACL-enabled system (such as Red Hat 6), the output of **ls** will look like this:

```
$ ls -l
drwxr-xr-x. 2 tux tux 4096 Dec 14 12:45 Desktop
-rw-rw-r--+ 1 tux tux    12 Dec 14 15:28 ourdocument
```

The "." (dot) after the permissions on the Desktop directory signify that there are no further ACLs associated with this directory (other than the default permissions/ACLs).

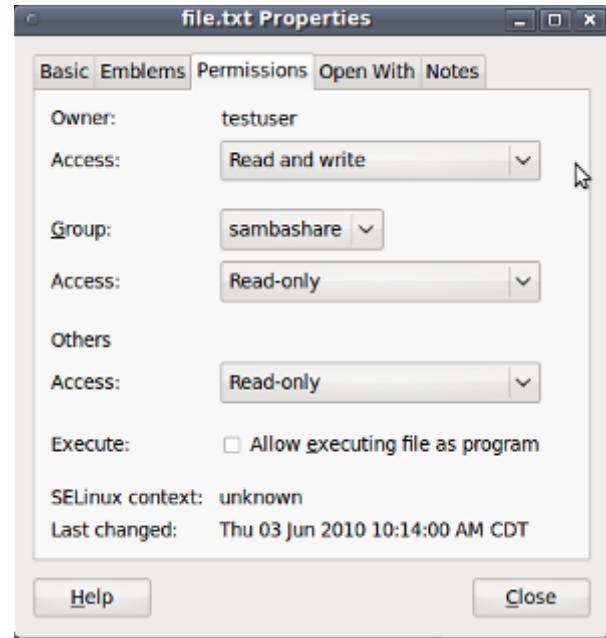
The "+" (plus) after the permissions on the ourdocument file mean that there are further ACLs assigned to this file. To view these ACLs, use **getfacl**:

```
$ getfacl ourdocument
# file: ourdocument
# owner: tux
# group: tux
user::rw-
user:tux2:rw-
group::rw-
mask::rw-
other::r--
```

As ACLs are typically not used by regular users, coverage of them is outside the scope of this course.

Permissions in the GUI

- You can also check file permissions using the GUI.
- The Properties tab allows you to check permissions, set permissions, and modify group and execution privileges all in one window.



© Copyright IBM Corporation 2012

Figure 5-12. Permissions in the GUI

LX027.2

Notes:

You can also view and edit file permissions using the GUI interface; however, the method might vary depending on the desktop environment. Typically, you would right-click on a file and choose **Properties** from the menu to bring up a window of properties; then click on the **Permissions** tab.

Unit review

- Permissions determine whether a user is able to do something with a file or directory.
- Permissions can be set for the user, the group, and all others.
- Three base permissions exist: read, write, and execute.
- To view the permissions, use `ls -l`.
- Permissions can be changed only by the owner of the file or directory and by root.
- The umask determines the default permissions on a file.

© Copyright IBM Corporation 2012

Figure 5-13. Unit review

LX027.2

Notes:

Checkpoint

```
$ pwd  
/groups/  
$ ls -l  
drwxrwxr-x  2  root  penguins 1024  Jan 1 10:03  penguins  
$ ls -l penguins  
-rw-r--r--  1  tux1  penguins  544  Jan 1 10:15  hello.c  
-rw-r--r--  1  tux1  penguins  544  Jan 1 10:15  task.c  
-rw-r--r--  1  tux1  penguins  544  Jan 1 10:15  zip.c
```

- Can tux2 (who is also a member of the penguins group) successfully execute the following commands?
 1. cd /groups/penguins
 2. mkdir /groups/penguins/mydir
 3. cp /groups/penguins/task.c ~/task.c
 4. vi /groups/penguins/zip.c
 5. vi /groups/penguins/newfile.c
 6. rm /groups/penguins/hello.c

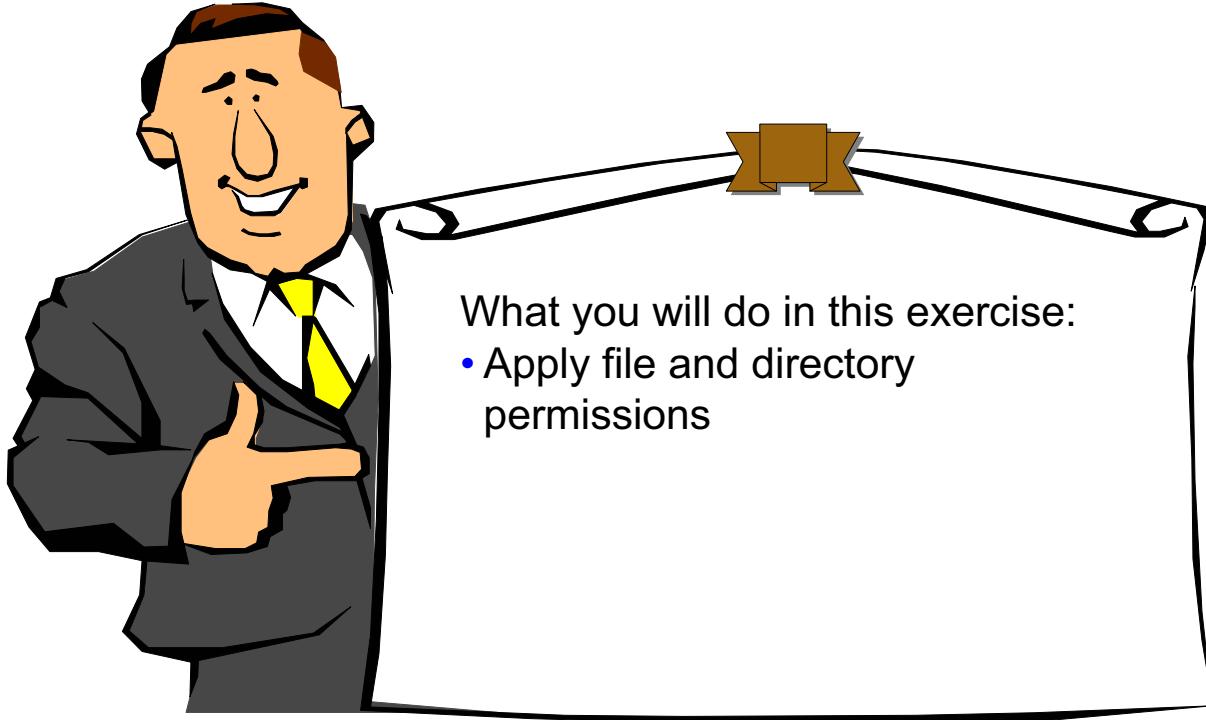
© Copyright IBM Corporation 2012

Figure 5-14. Checkpoint

LX027.2

Notes:

Exercise: File and directory permissions



© Copyright IBM Corporation 2012

Figure 5-15. Exercise: File and directory permissions

LX027.2

Notes:

Unit summary

Having completed this unit, you should be able to:

- Describe how permissions are used
- List the permissions required to perform several common commands
- Change permissions using symbolic and octal notation
- Describe how default permissions are calculated

© Copyright IBM Corporation 2012

Figure 5-16. Unit summary

LX027.2

Notes:

Unit 6. Linux documentation

What this unit is about

This unit describes the use of the **man** and **info** commands and the HOWTO documentation. This unit also explains the importance of the Internet for gathering information about Linux.

What you should be able to do

After completing this unit, you should be able to:

- Use the **man** command to view information about Linux commands
- Describe the use of **info**
- Describe the HOWTO documentation
- Explain the importance of the Internet for gathering information about Linux

How you will check your progress

- Checkpoint questions
- Machine exercises

Unit objectives

After completing this unit, you should be able to:

- Use the **man** command to view information about Linux commands
- Describe the use of **info**
- Describe the HOWTO documentation
- Explain the importance of the Internet for gathering information about Linux

© Copyright IBM Corporation 2012

Figure 6-1. Unit objectives

LX027.2

Notes:

The man command

- With the **man** command, you can read the manual page of commands.
- Manual pages are stored in /usr/share/man.
- The manual page consists of:
 - Name:** The name of the command and a one-line description
 - Synopsis:** The syntax of the command
 - Description:** Explanation of how the command works and what it does
 - Options:** The options used by the command
 - Files:** The files used by the command
 - Bugs:** Known bugs and errors
 - See also:** Other commands related to this one

© Copyright IBM Corporation 2012

Figure 6-2. The man command

LX027.2

Notes:

The **man** command shows the manual page of the commands and subroutines given as an argument to the **man** command. Most manual pages consist of the following:

- Name:** The title and a one-line description of the command
- Synopsis:** The syntax of the command
- Description:** Many pages of information about the function and usage of the command
- Options:** An explanation of the options
- Files:** Any system files associated with the command
- Bugs:** Any information about the behavior and performance of the command in unusual circumstances
- See also:** Other commands that are related to the same topic (viewing them can tell you more about the working of this particular command)

You can search for a pattern in a manual page with the forward slash (/) key.

man example (1 of 2)

```
$ man finger
FINGER(1)          BSD General Commands Manual      FINGER(1)

NAME
    finger — user information lookup program

SYNOPSIS
    finger [-lmsp] [user ...] [user@host ...]

DESCRIPTION
    The finger displays information about the system
    users.

    Options are:

        -s      Finger displays the user's login name, real
    Manual page finger(1) line 1
```

© Copyright IBM Corporation 2012

Figure 6-3. man example (1 of 2)

LX027.2

Notes:

This is only the first screen of the manual page of the **finger** command. You can now use the **less** commands (spacebar, b, q, and so forth) to browse the page.

man example (2 of 2)

- The -k option of the **man** command or the **apropos** command prints out a description of all entries that match the given keyword.

```
$ man -k print
arch (1) - print machine architecture
date (1) - print or set the system date and time
logname (1) - print user's login name
lpc (8) - line printer control program
lpd (8) - line printer spooler daemon
lpr (1) - off line print
lprm (1) - remove jobs from the line printer queue
```

© Copyright IBM Corporation 2012

Figure 6-4. man example (2 of 2)

LX027.2

Notes:

The **man -k** command shows the commands that have manual pages that contain any of the given keywords in their title.

The **apropos** command can also be used and is equivalent to using the **man -k** command.

To allow the use of **man -k** and **apropos**, the superuser (root) must have run the **/usr/sbin/makewhatis** command to create the **/var/cache/man/whatis** file. Typically, a distribution or an administrator sets up a cron job so that this is done each night. This is covered in the LX03 course.

man sections

- The collection of manual pages is divided into nine sections:
 1. Executable or shell commands
 2. System calls
 3. Library calls
 4. Special files (usually found in /dev)
 5. File formats and conventions
 6. Games
 7. Miscellaneous (macro packages and so on)
 8. System administration commands
 9. Kernel routines (non-standard)
- Certain subjects appear in multiple sections.
- To select correct section, add section number:
 - `man 1 passwd` (about the **passwd** command)
 - `man 5 passwd` (about the **passwd** file)

© Copyright IBM Corporation 2012

Figure 6-5. man sections

LX027.2

Notes:

Manual pages are stored in nine different sections. The first eight of them are standard across UNIX, and Section 9 is used for Linux kernel documentation. In some cases, a single subject might appear in multiple sections. As an example, **passwd** is both a command and a file; so a man page appears in two different sections. To retrieve a manual page from a specific section, specify the section number as the first argument to **man**.

The info command

- The **info** command is sometimes a replacement for manual pages.
- It is widely used by the GNU project.
- Information for **info** is stored in /usr/share/info.
- Some **info** commands include:

<space>	next screen of text
 or <bs>	previous screen of text
n	next node
p	previous node
q	quit info

© Copyright IBM Corporation 2012

Figure 6-6. The info command

LX027.2

Notes:

Another tool for viewing documentation is the **info** command.

The syntax of the info command is `info cmd_name`.

To view the documentation of the **info** command, enter `info info`. This displays the **info** documentation about the **info** command.

The **info** command works with entities named nodes. A node is one piece of information about a command or function. In **info**, you navigate through nodes to find and read information. The main difference between **info** and **man** is that these nodes can contain hyperlinks to other info pages, similar to the World Wide Web.

info has a lot of commands that help you navigate through the documentation. Some of these commands are:

- **<space>**: Next screen of text
- **** or **<bs>**: Previous page of text
- **N**: Next node
- **P**: Previous node
- **U**: Go to up node
- **B**: To top of node
- **E**: To end of node
- **S**: Search for a string in the current node
- **?**: Go to the help
- **L**: Leave the help and go back to the node
- **Q**: Quit info
- **<tab>**: Jump to the next cross reference
- **F**: Follow this cross reference (this brings you to another node)
- **M**: Pick menu item specified by name.
- **<Ctrl-H>**: Refresh screen

Cross references are indicated by an asterisk (*) on a line. With the **Tab** key, you can jump to this cross reference. Pressing the **f** key makes **info** follow the cross reference and show you another node.

Another way of moving through **info** is by specifying menu items. A node only contains a menu when you see * **Menu**: in the text. Menu items are also indicated by a *. Again use the **Tab** key to jump to a menu item. Then press the **m** key and **info** asks you what menu item you want to go to. Just pressing **Enter** makes you follow the link for the currently selected menu item. You could also enter another menu item, to follow its link.

The **info** command can be used to obtain the up-to-date information when a manual page starts with a sentence such as: “This documentation is no longer being maintained and may be inaccurate or incomplete. The Texinfo documentation is now the authoritative source.”

info example

```
# info pwd
```

```
File: coreutils.info,  Node: pwd invocation,  Next: stty invocation, \
Up: Working context

19.1 `pwd': Print working directory
=====
`pwd' prints the name of the current directory. Synopsis:

  pwd [OPTION]...

The program accepts the following options. Also see *note Common
options::.

`-L'
`--logical'
    If the contents of the environment variable `PWD' provide an
    absolute name of the current directory with no `.' or `...'
    components, but possibly with symbolic links, then output those
--zz-Info: (coreutils.info.gz)pwd invocation, 38 lines --Top-----
```

© Copyright IBM Corporation 2012

Figure 6-7. info example

LX027.2

Notes:

The **info** command is invoked with an argument that is the command of which you want to view the documentation. On the screen, you see the following:

- **File:** The file that contains the node you are looking at
- **Node:** The current node
- **Next:** The next node. You can use the **n** command to jump to this node
- **Up:** Besides a next node, a node can also have an up node. Use the **u** command to jump to the up node.

The node you are viewing:

- **Lines:** The total number of lines for this node
- **Position:**
 - **ALL:** You see all the lines of the node.
 - **TOP:** You are at the top of the node.
 - **BOT:** You are at the bottom of the node.
 - **75%:** You are at 75% of the node.

The --help option

- This is another way of getting help about a command.
- Help is built into the command itself (if supported).

```
$ cat --help
Usage: cat [OPTION]... [FILE]...
Concatenate FILE(s), or standard input, to standard output.

-A, --show-all      equivalent to -vET
-b, --number-nonblank number nonempty output lines
-e                equivalent to -vE
-E, --show-ends    display $ at end of each line
-n, --number       number all output lines
-s, --squeeze-blank suppress repeated empty output lines
-t                equivalent to -vT
-T, --show-tabs   display TAB characters as ^I
-u                (ignored)
-v, --show-nonprinting use ^ and M- notation, except for LFD and TAB
--help      display this help and exit
--version   output version information and exit
```

© Copyright IBM Corporation 2012

Figure 6-8. The --help option

LX027.2

Notes:

As we already saw, the **man** and **info** commands can be used to obtain information about the working of a command. This information is stored in a separate file in **/usr/share/man** or **/usr/share/info**. Obviously, this manual page has to be installed.

Another way of getting help about a command is using the **--help** option of the command itself. This option shows you a brief explanation of the synopsis of the command and the options that can be used with the command. The information shown is part of the command itself, and does not require the presence of a separate file.

The visual shows some lines of the help the **who --help** command would give you. The actual output probably does not fit on your screen. To read the complete help, issue **who --help | less**, which shows the output by page.

Note that not all commands support the **--help** option. Conveniently, for most commands that do not support **--help**, invoking them with **--help** will produce an invalid parameter error message, and the “usage” information will be printed anyway.

HOWTO documents

- These are documents that describe in detail a certain aspect of configuring or using Linux.
- They include detailed information about how to perform a given task.
 - DHCP support
 - Kernel compilation
 - Dual boot with other operating systems
- HOWTO documents are text files in /usr/share/doc/HOWTO.
 - Need to be installed manually
- On the Internet:
 - <http://www.tldp.org/index.html>

© Copyright IBM Corporation 2012

Figure 6-9. HOWTO documents

LX027.2

Notes:

Linux HOWTOs are documents that describe in detail a certain aspect of configuring or using Linux. For example, there is the installation HOWTO, which gives instructions on installing Linux, and the Mail HOWTO, which describes how to set up and configure mail under Linux. Other examples include the NET-3 HOWTO and the Printing HOWTO. HOWTOs are comprehensive docs, much like an FAQ but generally not in question-and-answer format; however, many HOWTOs contain an FAQ section at the end. There are several HOWTO formats available: plain text, PostScript, DVI, and HTML. In addition to the HOWTOs, there are a multitude of mini-HOWTOs on short, specific subjects. They are only available in plain text and HTML format.

HOWTO example

```
$ zless /usr/share/doc/HOWTO/en-txt/UPS-HOWTO.gz
UPS HOWTO

Eric Steven Raymond
[http://www.catb.org/~esr/] Thyrsus Enterprises

Nick Christenson

Revision History
Revision 2.2          2007-05-22      Revised by: esr
An Uninterruptible Power Supply (UPS) is an important thing to have if
you live in an area where power outages are at all common, especially if
you run a mail/DNS/Web server that must be up 24/7. This HOWTO will
teach you things you need to know to select a UPS intelligently and

/usr/share/doc/HOWTO/en-txt/UPS-HOWTO.gz
```

© Copyright IBM Corporation 2012

Figure 6-10. HOWTO example

LX027.2

Notes:

The example on the slide shows you the HOWTO on how to install and configure XFree86 on your system.

XFree86 is the graphical environment of a Linux system.

The example on the visual does not show the complete HOWTO. You see only the first 12 of 792 lines.

Other documentation

- Certain programs also offer other kinds of documentation.
 - HTML
 - PDF
 - PostScript
 - Plain text
- These are usually stored in
`/usr/share/doc/<package_name>`.

© Copyright IBM Corporation 2012

Figure 6-11. Other documentation

LX027.2

Notes:

When a programmer creates a program, the programmer usually includes the standard documentation, such as manual or info pages or both, and implements the --help option. But most programmers also write some non-standardized pieces of documentation. These are typically README files, with up-to-date release information, or CHANGELOGS, which list the changes since the previous versions. Other programmers might write large amounts of HTML-based documentation, or Postscript-based installation instructions, and so forth.

A typical distribution leaves this documentation intact and stores it in
`/usr/share/doc/<programname>`.

In practice, the value of this documentation varies greatly. There are programmers who only use the standardized tools (man, info) and as a consequence, `/usr/share/doc/<programname>` is virtually empty. Other programmers have created a whole Web site about their program, consisting of more than 20 HTML pages with supporting graphics, example configuration files, and so forth. So your mileage might vary here.

Internet

- All Linux documentation available on the Internet
- Google: <http://www.google.com/linux>
- Other sites:
 - www.tldp.org
 - www.linux.org
 - www.redhat.com
 - www.novell.com/linux
 - www.fedoraproject.org
 - www.kernel.org
 - www.lwn.net
 - www.ibm.com/developerworks/linux
 - Many more



© Copyright IBM Corporation 2012

Figure 6-12. Internet

LX027.2

Notes:

All information about Linux can also be found on the Internet. There are scores of Web pages on Linux. For more personal and up-to-date help, you can also go to Usenet news and other forums.

Unit review

- The **man** command can be used from the command line to view the proper syntax of Linux commands.
- Some commands have more complete documentation available by using the **info** command.
- Specific system administration tasks are described in the HOWTO documents.
- The Internet is the place for the latest information about Linux.

© Copyright IBM Corporation 2012

Figure 6-13. Unit review

LX027.2

Notes:

Checkpoint

1. True or False: A HOWTO document is the best source of documentation if you want up-to-date information about a specific command.

2. The main Linux documentation Web site is:
 - a. <http://www.tldp.org>
 - b. <http://www.linux.org>
 - c. <http://www.lwn.net>
 - d. <http://www.kernel.org>

3. In which sections are manual pages divided?

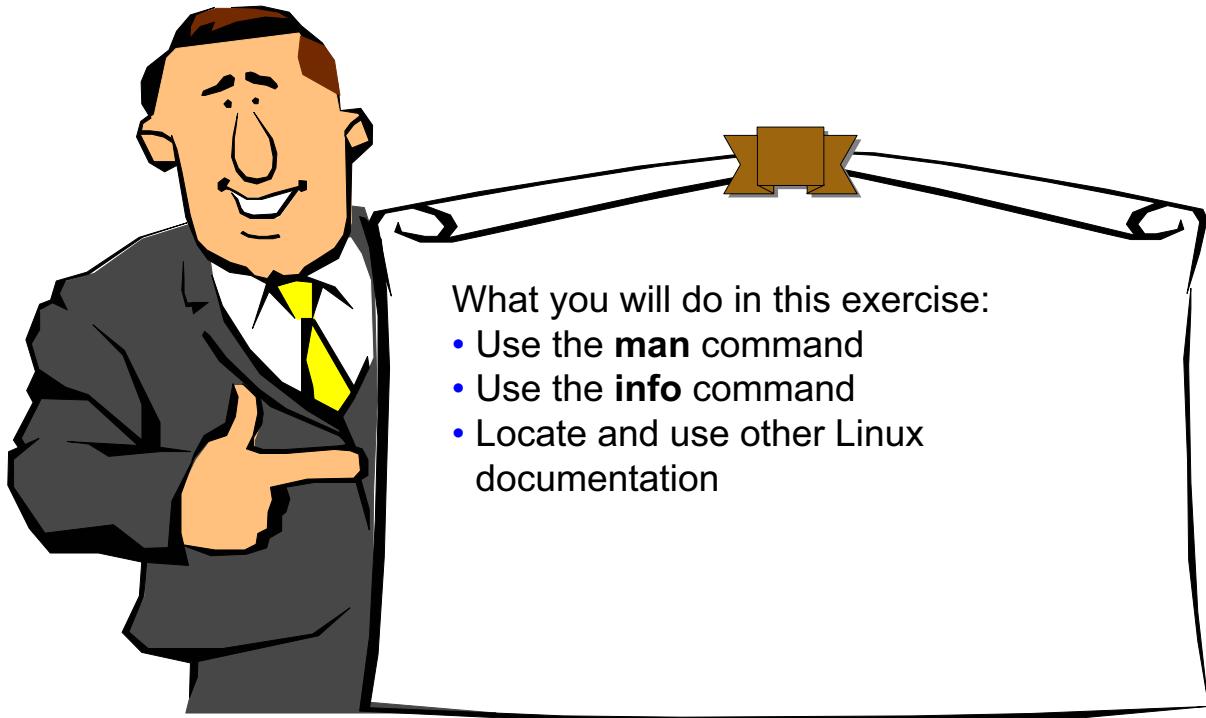
© Copyright IBM Corporation 2012

Figure 6-14. Checkpoint

LX027.2

Notes:

Exercise: Linux documentation



What you will do in this exercise:

- Use the **man** command
- Use the **info** command
- Locate and use other Linux documentation

© Copyright IBM Corporation 2012

Figure 6-15. Exercise: Linux documentation

LX027.2

Notes:

Unit summary

Having completed this unit, you should be able to:

- Use the **man** command to view information about Linux commands
- Describe the use of **info**
- Describe the HOWTO documentation
- Explain the importance of the Internet for gathering information about Linux

© Copyright IBM Corporation 2012

Figure 6-16. Unit summary

LX027.2

Notes:

Unit 7. Editing files

What this unit is about

This unit shows how to use the **file** command to determine the type of file, illustrates how to edit text files with Vi, and discusses other text file editors as well as ways that non-text files can be edited.

What you should be able to do

After completing this unit, you should be able to:

- Determine the type of file using **file**
- Edit text files with Vi
- Discuss other text file editors, such as KEDIT
- Discuss the ways non-text files can be edited

How you will check your progress

- Checkpoint questions
- Machine exercises

Unit objectives

After completing this unit, you should be able to:

- Determine the type of file using **file**
- Edit text files with **Vi**
- Discuss other text file editors, such as **KEDIT**
- Discuss the ways non-text files can be edited

© Copyright IBM Corporation 2012

Figure 7-1. Unit objectives

LX027.2

Notes:

Determining file content

- Use the **file** command to determine the content of a file

```
$ file /etc/passwd  
/etc/passwd: ASCII text  
$ file /usr/bin/passwd  
/usr/bin/passwd: ELF 32-bit LSB executable
```

- To edit text files, use a **text editor** such as **vi**
- Non-text files can only be changed using the application that created them, or with a **hex editor**
- But most configuration files in Linux are text files

© Copyright IBM Corporation 2012

Figure 7-2. Determining file content

LX027.2

Notes:

When we are editing a file, we are changing the content of the file. For this, we need an editor.

Since Linux does not impose any structure on the contents of the file, there is no editor under Linux which can edit any file available. Different files need to be edited using different editors.

To determine the file type, the **file** program is used. This program reads the first few bytes of the file and compares it to a database of known file types. If there is a match, then the type of the file is displayed.

If the file turns out to be a text file, then you can edit this file with a text editor. On a typical Linux system, there's usually a large assortment of text editors available, including **vi**, **kedit**, **emacs** and so forth. Non-text files usually need to be edited through the application that created them, or with a so-called "hex-editor", an editor which displays the file in hexadecimal format and thus is able to display and modify non-printable characters.

Most configuration files on a Linux system are text files. This makes it possible to perform most system administration tasks with just a simple text editor.

The Vi text editor

- Vi is the default editor in all UNIX operating systems.
- It is usually the only editor available in emergencies.
- It is relatively hard to learn, but it is very powerful.
- As a Linux user, you should be able to use Vi for basic editing tasks.
 - But OK if you prefer another editor for daily work
- Vi in Linux is usually Vim (Vi Improved).
 - Syntax highlighting
 - Arrow keys, Del, BS work in insert mode
 - Multi level undo
 - Mouse support

© Copyright IBM Corporation 2012

Figure 7-3. The Vi text editor

LX027.2

Notes:

Vi, which officially stands for visual interpreter, is the most commonly available editor in all UNIX operating systems. The reason for this is that it was the first, and for a long time only editor which was capable of editing a file in full-screen mode. Before Vi, all editors were line-based: they could only display and edit one line at a time.

Considering today's standards, Vi is relatively hard to learn. Editors like KEDIT for example have a graphical interface and are therefore much easier to use for novices. But if you need to do system management on a remote system, connected via a slow network or modem connection, KEDIT is not an option.

The same goes for emergency situations. When a system crashes and won't boot in the normal fashion anymore, you need to fall back to some sort of rescue mode. This rescue mode is usually started from CD-ROM or over the network, and makes only the minimum amount of tools available to get the system up and running again. In such a rescue mode, the only full-screen editor available is usually Vi.

The last reason for learning Vi is that certain programs (such as mail and news readers) use an external editor if the user needs to type more than a few words (the body of your

e-mail message for instance). In all but a few cases, the external editor called is Vi. So it is important to learn Vi, at least enough to make simple changes to text files. That doesn't mean that Vi has to be your preferred editor, and that you need to learn all features.

If that's what you want, that's fine. But if you prefer KEDIT or Emacs (to name two ends of the spectrum) for your daily work, that's fine too.

When executing the vi command in Linux, in most distributions, the program actually started is Vim (Vi Improved). Vim is downwards compatible with Vi, but offers a large number of advantages and improvements over traditional Vi. The visual lists some of them.

For a complete list, start Vi (or Vim) and execute the command :help vi_diff.txt.

Vi modes

- Vi knows three modes of operation.
 - Command mode (for simple, one-letter commands)
 - Edit mode (insert text)
 - ex mode (for complicated commands)
 - You can easily change between modes.

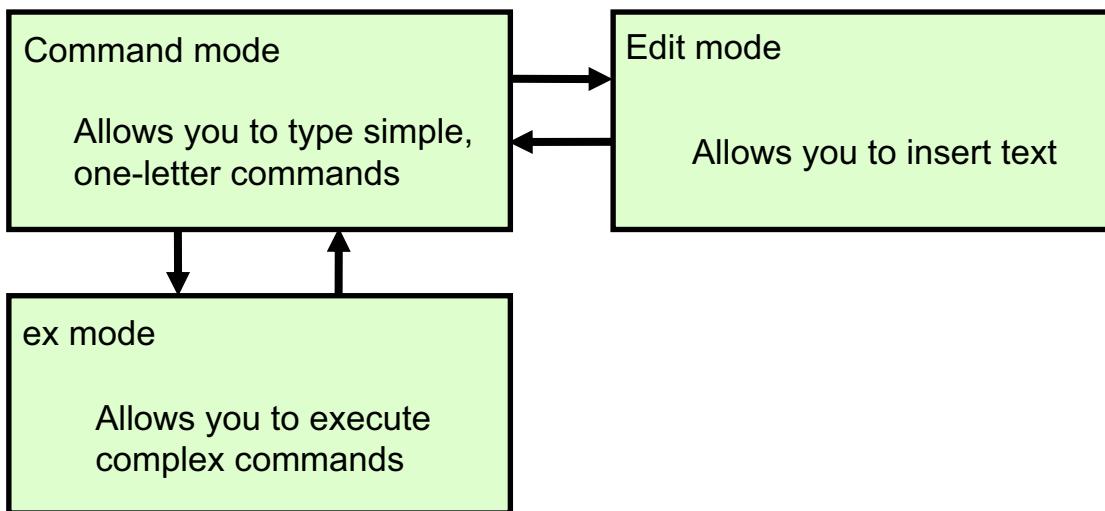


Figure 7-4 Vi modes

1 X037.3

Notes:

The apparent complexity of Vi is largely due to the concept of different modes that Vi uses, and that you constantly need to change between them.

The first mode that you need to know about is the command mode. When Vi is in this mode, you can type simple, one-letter commands that do specific things. There are as many commands as there are letters in the alphabet, where the lowercase and uppercase letter (and sometimes the control-sequence too) have a different meaning.

The second mode is the edit mode. This mode allows you to enter characters that show up in the file.

The third mode is the ex mode. The ex line editor was the direct predecessor of Vi. It was really powerful, but did not have full-screen capabilities. Most of the powerful ex commands have been integrated in Vi, as the ex mode.

Starting Vi

- \$ vi myfile.txt

```
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
"myfile.txt" [New File]      0, 0 - 1          All
```

© Copyright IBM Corporation 2012

Figure 7-5. Starting Vi

LX027.2

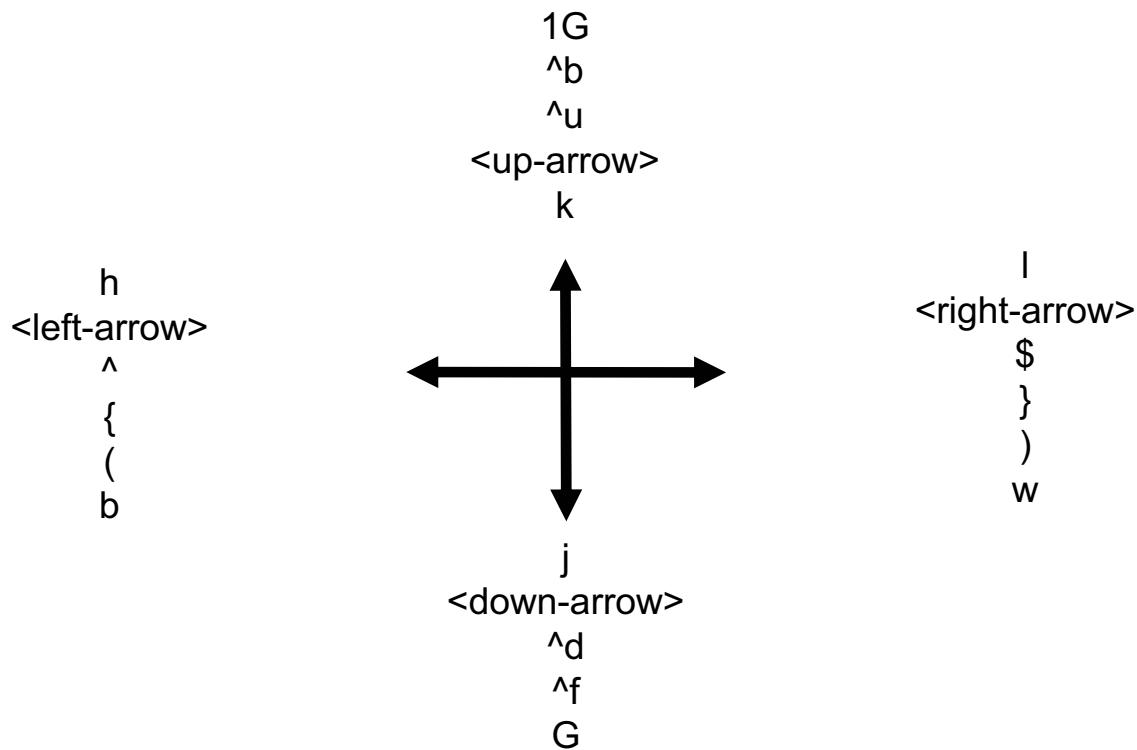
Notes:

Vi does editing in a buffer. When a session is initiated, one of two things happen:

- If the file to be edited exists, a copy of the file is put into a buffer in /tmp by default.
- If the file does not exist, an empty buffer is opened for this session.

Tildes (~) represent empty lines that are not part of the file you are editing. The editor starts in command mode.

Cursor movement in command mode



© Copyright IBM Corporation 2012

Figure 7-6. Cursor movement in command mode

LX027.2

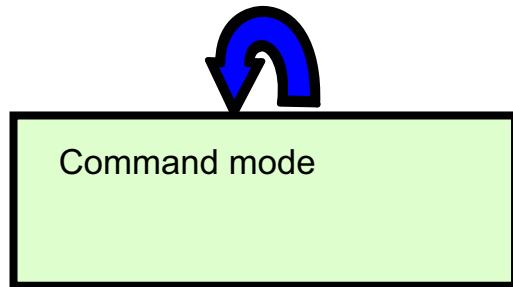
Notes:

To move about in your file, make sure you are in command mode.

- **<left arrow> or h:** One character left
- **<right arrow> or l:** One character right
- **B or b:** One word left
- **W or w:** One word right
- **^:** Move to beginning of line
- **\$:** Move to end of line
- **():** One sentence left or right
- **{ }:** One paragraph left or right
- **<up-arrow> or k:** One line up
- **<down-arrow> or j:** One line down
- **1G:** Go to the first line of the file
- **G:** Go to the last line of the file
- **<ctrl-u> or <ctrl-d>:** One half-page up or down
- **<ctrl-b> or <ctrl-f>:** One page back or forward

Editing text in command mode

- To delete a single character under cursor: **x**
- To delete a single character left of cursor: **X**
- Undo the last change: **u**
- To repeat last command: **.**
- To join two lines together: **J**



© Copyright IBM Corporation 2012

Figure 7-7. Editing text in command mode

LX027.2

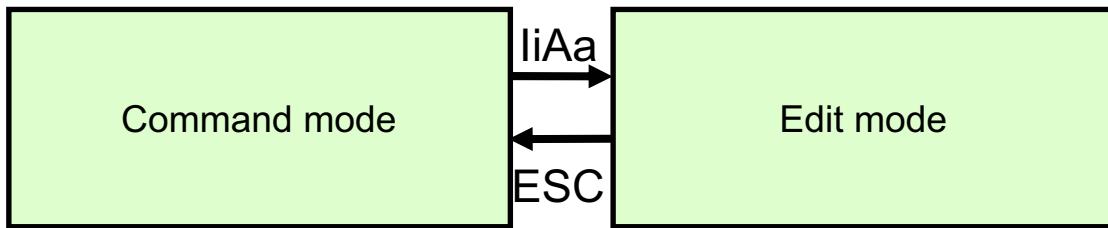
Notes:

To execute one of the illustrated commands, you must be in command mode.

There are several different ways to perform the delete functions.

Switching to edit mode

- To insert text at beginning of line: **I**
- To insert text before cursor: **i**
- To append text after cursor: **a**
- To append text at end of line: **A**
- To go back to command mode: <**ESC**>



© Copyright IBM Corporation 2012

Figure 7-8. Switching to edit mode

LX027.2

Notes:

There are a number of ways to get from command mode to edit mode:

- **I (capital I):** Insert text at beginning of current line.
- **i:** Insert text before current cursor position.
- **a:** Append text after current cursor position.
- **A:** Append text at end of line.

To exit the edit mode, press the **Esc** key.

Adding text in edit mode

```
This file contains some lines.
```

```
Line 2.
```

```
And this is line 3.
```

```
Line 4 follows line 3.
```

```
The last line is line 5.
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
-- INSERT --
```

```
3,8
```

```
All
```

- Keystroke **i** switches Vi to edit mode. New characters can be inserted at the current position of the cursor.

© Copyright IBM Corporation 2012

Figure 7-9. Adding text in edit mode

LX027.2

Notes:

After starting Vi, you are in the command mode. If you want to type some text, you have to change to the edit mode of Vi.

To enter edit mode, enter the **i** command. This places you in edit mode. Look at the last line in Vi because it should state now that you are in edit mode (it shows “-- INSERT --”).

To exit from input mode, press the **<esc>** key. The bottom line should no longer show “-- INSERT --”.

Exiting the edit mode

```
This file contains some lines.  
Line 2.  
And this for example is line 3.  
Line 4 follows line 3.  
The last line is line 5.  
~  
~  
~  
~  
~  
~
```

3,8

All

- Keystroke **ESC** leaves the edit mode.

© Copyright IBM Corporation 2012

Figure 7-10. Exiting the edit mode

LX027.2

Notes:

To leave the edit mode, use the **<esc>** key.

Searching for patterns

- To search for a pattern (in command mode): /<pattern>
- To repeat the previous search: **n**

```
This file contains some lines.  
Line 2.  
And that for example is line 3.  
Line 4 follows line 3.  
The last line is line 5.  
~  
~  
~  
~  
~  
~  
/line
```

© Copyright IBM Corporation 2012

Figure 7-11. Searching for patterns

LX027.2

Notes:

Vi can also search for patterns. This is done with the / (slash) command. To repeat a previous search, use the **n** command.

Replacing patterns

- Advanced search and replace can be done in ex mode.
- To replace old text with new text use the following command:
:1,\$s /old/new/g

```
This file contains some lines.
Line 2.
And that for example is line 3.
Line 4 follows line 3.
The last line is line 5.
~
~
~
~
~
~
: 1,$s/this/that/g
```

© Copyright IBM Corporation 2012

Figure 7-12. Replacing patterns

LX027.2

Notes:

Advanced search and replace functions are available in ex mode. The command :1,\$s /old/new/g for instance replaces all occurrences of the word *old* with *new*.

Let's break down this command a little:

- The colon (:) switches to **ex** mode.
- 1,\$ means that our command is going to apply to all lines, starting with line 1 and ending with the last line of the file. You could also specify 1,5 to limit your search to lines 1 through 5, inclusive. Other possibilities are .,\$, for the current line through to the end, or %, which also means the whole file.
- s means that you are going to execute a search and replace. There are a lot of other possibilities here, but d is perhaps the most common: it allows you to delete lines.
- The first slash (/) is the start delimiter of the search phrase. The second slash (/) is the end delimiter of the search phrase, and the start of the replacing phrase. The third / is the end delimiter of the replacing phrase.

You are free to choose your delimiter character, but / is the most common. If your

delimiter character happens to be part of the search or replace phrase, pick a delimiter character which is not (usually a colon (:)) is used instead), or make sure you escape the delimiter character with a backslash.

- The **g** means a global replace. Normally only the first occurrence of the search phrase on a particular line is replaced.

There are far more possible commands in ex mode. See the manual page of Vi.

Cut, copy, and paste

- To cut a whole line into buffer: **dd**
- To copy a whole line into buffer: **yy**
- To cut a word from the current cursor position to its end: **dw**
- To paste contents of buffers here: **p**
- To cut or copy multiple lines, proceed command by number:
3dd, 8yy

© Copyright IBM Corporation 2012

Figure 7-13. Cut, copy, and paste

LX027.2

Notes:

The **dd** and **yy** commands allow you to cut and copy a single line into a buffer. The **p** command then retrieves the buffer contents and adds it after the current line.

To cut or copy multiple lines at once, precede the command by the number of lines you want to cut or copy.

Cut and paste

```
This file contains some lines.  
Line 2.  
And that for example is line 3.  
Line 4 follows line 3.  
The last line is line 5.
```

- Cut line three by typing **dd**.

```
This file contains some lines.  
Line 2.  
Line 4 follows line 3.  
The last line is line 5.  
And that for example is line 3.
```

- Insert it after line four by typing **p**.

© Copyright IBM Corporation 2012

Figure 7-14. Cut and paste

LX027.2

Notes:

The **dd** and the **p** commands are used to move (a number of) lines. Use the proper keys (that is, **H**, **J**, **K** or **I**) to go to the line you want to move. Now give the **dd** command. This removes the line you were in and place it in a buffer.

Use the move keys again to go to the place where you want the line to reappear. Use the **p** command if you want the line to reappear under the line where the cursor is in. Use the **P** command if you want the line to reappear above the line the cursor is in.

You can also delete a couple of lines with the **dd** command. To delete 12 lines, enter **12dd**. The **u** command can undo your last command if you make an error. If you delete something in error, immediately type the **u** command to retrieve it.

Copy and paste

```
This file contains some lines.  
Line 2.  
And that for example is line 3.  
Line 4 follows line 3.  
The last line is line 5.
```

- Copy line three by typing **yy**.

```
This file contains some lines.  
Line 2.  
And that for example is line 3  
Line 4 follows line 3.  
The last line is line 5.  
And that for example is line 3.
```

- Insert it after line five by typing **p**.

© Copyright IBM Corporation 2012

Figure 7-15. Copy and paste

LX027.2

Notes:

To copy and paste text, you can use the same steps you used with cut and paste. The only difference is the **yy** command instead of the **dd** command.

Vi options

- Options entered in ex mode change the behavior of the Vi editor.
 - :set all
 - :set autoindent/noautoindent
 - :set number/nonumber
 - :set list/nolist
 - :set showmode/noshowmode
 - :set tabstop=x
 - :set ignorecase/noignorecase
 - :set wrapmargin=x
 - :set hlsearch/nohlsearch
 - :syntax on/off
 - :set fileformat=dos/unix
- To make an option available to all Vi sessions, put it into a `.exrc` or `.vimrc` file in your home directory.

© Copyright IBM Corporation 2012

Figure 7-16. Vi options

LX027.2

Notes:

Vi has many settings for operation. Some of these affect the way text is presented, while others make editing easier for novice users.

- **:set all:** Display all settings
- **:set autoindent:** Sets autoindent on
- **:set ai:** idem
- **:set noai:** Turns autoindent off
- **:set number:** Enables line numbers
- **:set nu:** idem
- **:set nonu:** Turn line numbers off
- **:set list:** Display non-printable characters
- **:set nolist:** Hide non-printable characters
- **:set showmode:** Show the current mode of operation (default on)
- **:set noshowmode:** Hide the mode of operation
- **:set tabstop=4:** Set tab to 4 character jumps
- **:set ts=4:** idem

Beware of tab stop settings. They apply only to your local display. What is inserted in the file is actually the tab character itself, which is expanded to spaces whenever the file is read. The number of spaces depend on the terminal settings at that moment, not at the terminal settings from the time the file was created.

- **:set ignorecase:** Ignore case-sensitive
- **:set ic:** idem
- **:set noic:** Case-sensitive
- **:set wrapmargin=5:** Set the margin for automatic word wrapping from one line to the next.
- **:set wrapmargin=0:** Turn off word wrapping
- **:set hlsearch:** Set highlighting of search results on
- **:set nohlsearch:** Set highlighting of search results off
- **:syntax on/off:** Set syntax highlighting on/off
- **:set fileformat=dos:** Set MS-DOS style line endings (CR/LF)
- **:set fileformat=unix:** Set UNIX style line endings (LF)

You might want to create a **.exrc** or **.vimrc** file in your home directory containing these commands (without the leading colon). The settings in this file are then read and applied when starting Vi (or Vim).

Exiting Vi

- To save in ex mode
 - :w
- To forcefully save file in ex mode
 - :w!
- To quit without saving in ex mode
 - :q
- To forcefully exit in ex mode (without saving changes)
 - :q!
- To save and exit in ex mode (recommended)
 - :wq
- To save and exit in ex mode, shorter
 - :x
- To save and exit in command mode
 - ZZ

© Copyright IBM Corporation 2012

Figure 7-17. Exiting Vi

LX027.2

Notes:

There are a number of ways to exit Vi, both in command mode and ex mode.

The :w! command is useful if you are editing a read-only file but want to write it nevertheless. Vi then tries to remove the read-only attribute, write the file, and set the read-only attribute again. Obviously, if Vi cannot remove the read-only attribute (because the user is not the owner, for instance), Vi reports an error.

The :q! command is useful if you want to exit Vi without saving your changes.

Vi cheat sheet

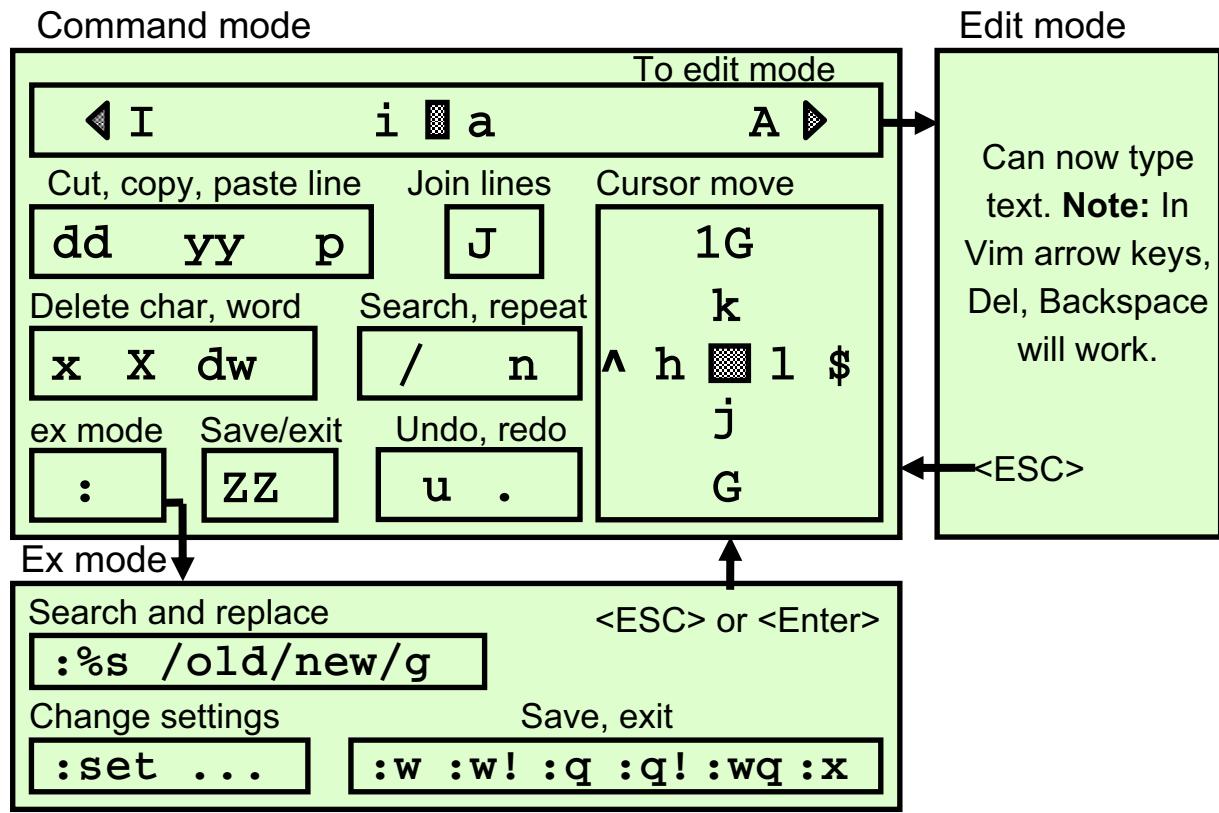


Figure 7-18. Vi cheat sheet

LX027.2

Notes:

You can use this page as a quick reference for some of the most used commands in Vi. There are many cheat sheets or helper pages on the Web. Do a Google-Linux search for Vi and cheat sheet and use the one you like.

Other editors

- A typical Linux distribution comes with a large number of editors.
- Text mode editors:
 - Pico (really simple), Joe
 - Original Vi
 - Emacs (even more powerful and complicated than Vi)
- Graphical mode editors:
 - KVim, KEDIT, KWrite
 - gVim, gedit
- Hex editors allow you to change non-text files if you know the internal structure.
 - KHexEdit
 - Emacs (in hexl-mode)

© Copyright IBM Corporation 2012

Figure 7-19. Other editors

LX027.2

Notes:

If you do not want to use Vi, or cannot use Vi on the file you want to edit, there are usually more editors available. The visual just shows a shortlist. If you do not like any of the editors that are available, you are free to write your own (that is the main reason that there are so many editors around in the first place!)

Unit review

- The most common editor on any UNIX is Vi.
- Vi has three modes of operation: command mode, edit mode, and ex mode.
- Vi makes a copy of the file you are editing in an edit buffer. The contents are not changed until you save the changes.
- A typical Linux distribution comes with a lot of other editors as well.

© Copyright IBM Corporation 2012

Figure 7-20. Unit review

LX027.2

Notes:

Checkpoint

1. True or False: You need to learn Vi because Vi is the best editor for any job.

2. What does the **file** command do?
 - a. It looks at the extension to determine the type of file.
 - b. It looks at the first few characters of the file and compares this to a database of known file types.
 - c. It asks the kernel for information about the file.
 - d. It makes a wild guess.

3. What is a hex editor?

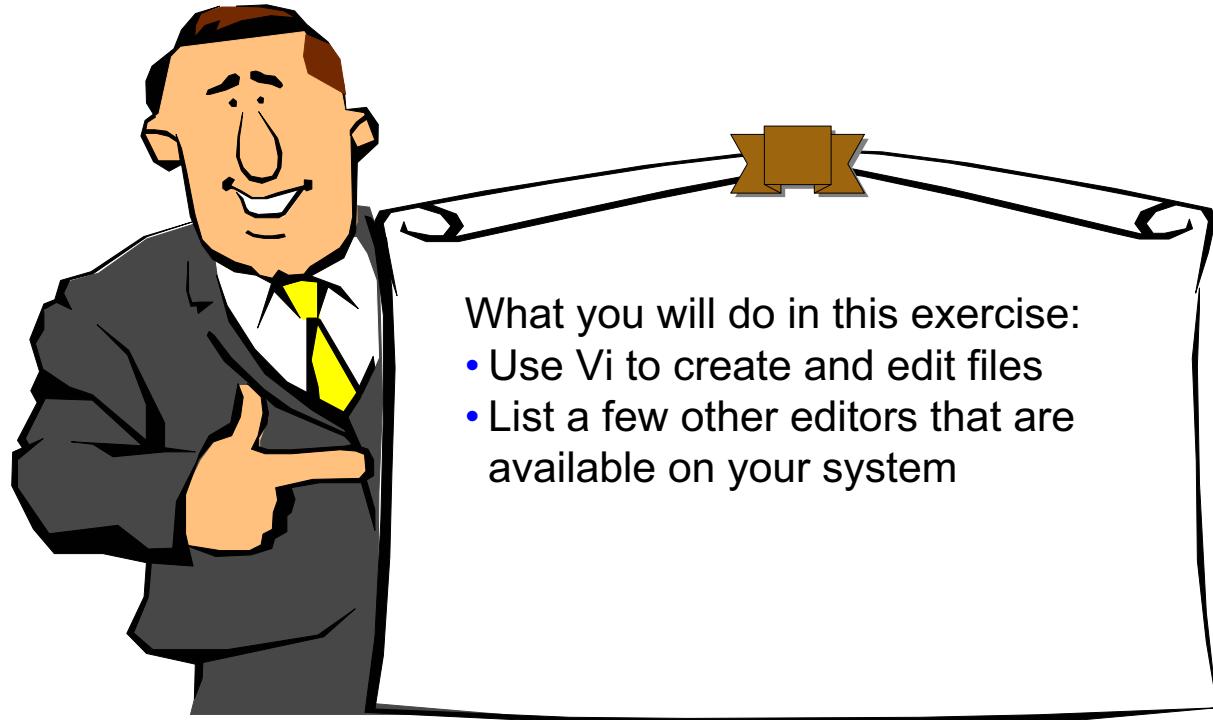
© Copyright IBM Corporation 2012

Figure 7-21. Checkpoint

LX027.2

Notes:

Exercise: Editing files



What you will do in this exercise:

- Use Vi to create and edit files
- List a few other editors that are available on your system

© Copyright IBM Corporation 2012

Figure 7-22. Exercise: Editing files

LX027.2

Notes:

Unit summary

Having completed this unit, you should be able to:

- Determine the type of file using **file**
- Edit text files with **Vi**
- Discuss other text file editors, such as KEDIT
- Discuss the ways non-text files can be edited

© Copyright IBM Corporation 2012

Figure 7-23. Unit summary

LX027.2

Notes:

Unit 8. Shell basics

What this unit is about

This unit describes the function of the shell, metacharacters and reserved words, the use of wildcards, using redirection and pipes, using command substitution, common filters, grouping commands, working with shell variables, using aliases, and applying quoting.

What you should be able to do

After completing this unit, you should be able to:

- Explain the function of the shell
- Discuss metacharacters and reserved words
- Use wildcards to access files with similar names
- Use redirection and pipes
- Use command substitution
- Describe and use the most common filters
- Group commands to control their execution
- Work with shell variables
- Use aliases
- Apply quoting

How you will check your progress

- Checkpoint questions
- Machine exercises

Unit objectives

After completing this unit, you should be able to:

- Explain the function of the shell
- Discuss metacharacters and reserved words
- Use wildcards to access files with similar names
- Use redirection and pipes
- Use command substitution
- Describe and use the most common filters
- Group commands to control their execution
- Work with shell variables
- Use aliases
- Apply quoting

© Copyright IBM Corporation 2012

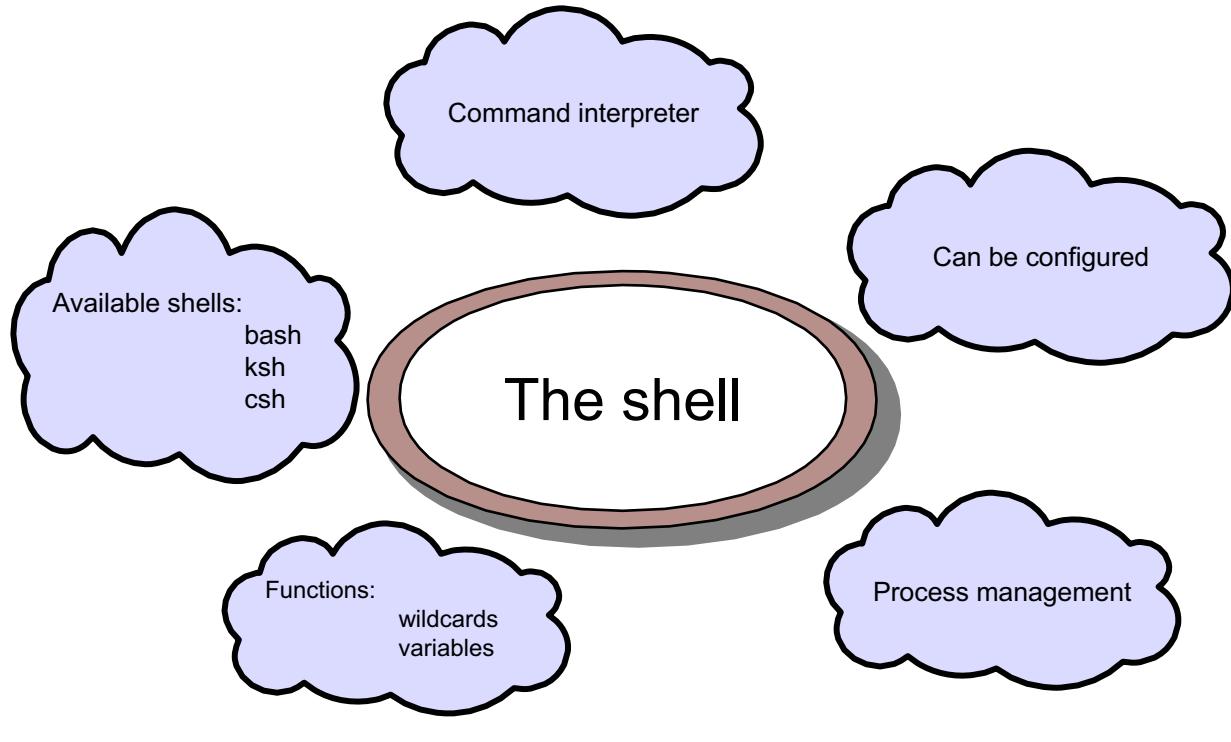
Figure 8-1. Unit objectives

LX027.2

Notes:

The shell

- The *shell* is the command line user interface to Linux.



© Copyright IBM Corporation 2012

Figure 8-2. The shell

LX027.2

Notes:

The shell is a special program in any UNIX operating system, including Linux, because it allows the user to interact with the operating system.

It does this by allowing the user to type a command and subsequently interprets and executes this command. The shell also has a number of commands and control structures built-in, which allow it to be used as a comprehensive programming language.

A huge number of shells have been written by various people since the first shell, simply called sh was released. A lot of these shells are also available for Linux.

In most distributions the default shell is bash, which stands for Bourne Again Shell.

Tcsh is another fairly popular shell, which is similar, but also has many syntactic differences. It will not be covered.

Shell features

- When the user types a command, various things are done by the shell before the command is actually executed.
 - Wildcard expansion * ? []
 - Input/output redirection < > >> 2>
 - Command grouping { com1 ; com2; }
 - Line continuation \
 - Shell variable expansion \$var
 - Alias expansion dir -> ls -l
 - Shell scripting #!/bin/bash
- For example, the ls *.doc command could be expanded to /bin/ls --color=tty mydoc.doc user.doc before execution (depending on settings and files present).

© Copyright IBM Corporation 2012

Figure 8-3. Shell features

LX027.2

Notes:

As said, the shell allows you to type your command, after which it interprets it and executes the corresponding program. However, before it executes the program it performs certain transformations on the command you just entered. If you know what these transformations are, and how to use them, it makes your life much simpler.

Some of these transformations are listed in the visual. We cover them in this unit. But these are not the only things that the shell can do. Just like when we talked about Vi, we are only going to scratch the surface here. For more in-depth information, read the book *Learning the bash Shell from O'Reilly* (ISBN 1-6592-347-2), or go to any of IBM's other Linux courses, in particular the AL32 (*Bash Shell Programming*).

Metacharacters and reserved words

- Metacharacters are characters that the shell interprets as having a special meaning.
 - Examples: < > | ; ! ? * \$ \ ` ' " ~ [] () { }
- Reserved words are words that the shell interprets as special commands.
 - Examples: case, do, done, elif, else, esac, for, fi, function, if, in, select, then, until, while

© Copyright IBM Corporation 2012

Figure 8-4. Metacharacters and reserved words

LX027.2

Notes:

For the shell to make a distinction between the actual command and the actual parameters that you typed, and the hints you want to give to the shell to let it do something, there has to be some sort of agreement on what the shell can touch in your command and what not. As part of this agreement, a number of metacharacters and reserved words have been defined.

Metacharacters are individual characters that have a special meaning to the shell. They can appear anywhere in your command and are always handled by the shell itself, *before* the command is executed. Reserved words are words that the shell interprets as special commands. They only have a special meaning if they appear as a single word, surrounded by whitespace¹. *Function* for instance is a reserved word, but *functions* is not.

You should never name your file or program after a reserved word, nor should your file names ever contain a reserved character. (Technically, it is possible, but it only gets you or somebody else in trouble later, when you or they need to work with the file by name and have trouble typing it.)

¹ Whitespace is the beginning of the line, one or more spaces or tabs, or the end of the line.

Basic wildcard expansion

- When the shell encounters a word that contains a wildcard, it tries to expand this to all matching file names in the given directory.

```
$ ls -a
. . . .et .w few myfile ne nest net new test1 test1.2
test1.3

? matches a single character
$ ls ne?
net new
$ ls ?e?
few net new

* matches any string, including the null string
$ ls n*
ne nest net new
$ ls *w
few new
```

© Copyright IBM Corporation 2012

Figure 8-5. Basic wildcard expansion

LX027.2

Notes:

One of the first things the shell does after it has read your command is to try to perform wildcard expansion. This means that it starts looking in your command for words that contain (or solely consist of) one or more wildcards. It then looks in the filesystem to see if it can expand that word to one or more file names that match the pattern.

The most often used wildcards are the asterisk (*) and question mark (?). The asterisk (*) matches zero or more arbitrary characters. The question mark (?) matches exactly one arbitrary character.

Note that file names that start with a dot (.) are considered “hidden” by the shell and are not included in wildcard expansion unless a dot is explicitly included in the expression, like “.*”. And don’t just blindly try .* either, since that also matches ., the current directory, and .., the higher-level directory. So if you execute `rm -fr /tmp/ .*`, you are wiping out your whole system and not just the contents of /tmp.

Wildcard expansion is done for each and every command that is entered by the user, even if the program that started does not accept any filenames as parameters at all! As an example, the command `userdel *` is entirely legal, but does not delete all user accounts on your system. What does the command `userdel *` attempt to do?

Advanced wildcard expansion

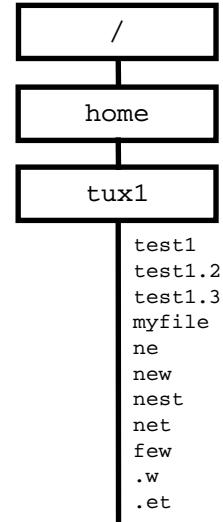
- The wildcards [,], -, and ! match inclusive lists.

```
$ ls ne[stw]
net new

$ ls *[1-5]
test1 test1.2 test1.3

$ ls [!tn]*
few myfile

$ ls ?[!y]*[2-5]
test1.2 test1.3
```



© Copyright IBM Corporation 2012

Figure 8-6. Advanced wildcard expansion

LX027.2

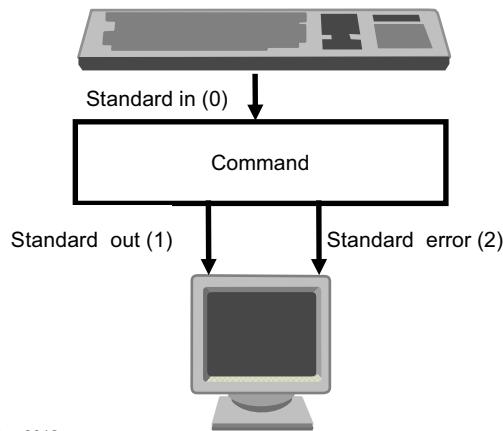
Notes:

At certain times, the ? and * wildcards give a match that is too broad. You might for instance only want filenames that start with the letters *a* or *b*. In that case, inclusive lists can be used. An inclusive list is defined with square brackets ([and]), which contain the letters to match. The dash sign can be used to specify a range, and the exclamation mark can be used to invert the list (all characters match except the ones listed). See the visual for examples.

File descriptors

- Every program has a number of file descriptors associated with it.
- Three descriptors are assigned by the shell when the program starts (STDIN, STDOUT, and STDERR).
- Other descriptors are assigned by the program when it opens files.

Standard in	STDIN	<	0
Standard out	STDOUT	>	1
Standard error	STDERR	2>	2



© Copyright IBM Corporation 2012

Figure 8-7. File descriptors

LX027.2

Notes:

Three files are automatically opened for each process in the system. These files are referred to as standard input (STDIN), standard output (STDOUT), and standard error (STDERR).

Standard input is where a command expects to find its input, by default the keyboard. Standard out and is where a command sends its normal, expected output. Standard error is where the command sends its error, unexpected output. The user's terminal session is the default for both standard output and standard error. These defaults can be changed using redirection.

Input redirection

- Default standard input:

```
$ cat
Atlanta
Atlanta
Chicago
Chicago
<Ctrl-d>
```

- STDIN redirected from file:

```
$ cat < cities
Atlanta
Chicago
$
```

© Copyright IBM Corporation 2012

Figure 8-8. Input redirection

LX027.2

Notes:

The symbol < tells **cat** to take input from the file instead of the keyboard.

The file table for the redirection example looks like the following:

Descriptor 0 (standard input) :

- default: keyboard
- for "cat < cities" command: "cities" file

Descriptor 1 (standard output) :

- default: screen
- for "cat < cities" command: screen

Descriptor 2 (standard error) :

- default: screen
- for "cat < cities" command: screen

Output redirection

- Default standard output: /dev/tty

```
$ ls  
file1 file2 file3
```

- Redirect output to a file:

```
$ ls > ls.out
```

- Redirect and append output to a file:

```
$ ls >> ls.out
```

- Create a file with redirection:

```
$ cat > new_file  
Save this line  
<Ctrl-d>
```

© Copyright IBM Corporation 2012

Figure 8-9. Output redirection

LX027.2

Notes:

Redirection allows standard output to go to somewhere other than the screen (default). In the example, standard output has been redirected with `>` to go the file named `ls.out`.

The file descriptor table in this example holds the following values:

Descriptor 0 (standard input) :

- default: keyboard
- for “`cat < cities`” command: keyboard

Descriptor 1 (standard output) :

- default: screen
- for “`cat < cities`” command: file “`ls.out`”

Descriptor 2 (standard error) :

- default: screen
- for “`cat < cities`” command: screen

Using ordinary redirection overwrites an existing file. To avoid this, use the `>>` (no space between them) to append output to an existing file.

Error redirection

- Default standard error: /dev/tty

```
$ cat fileA
cat: fileA: No such file or directory
```

- Redirect error output to a file:

```
$ cat fileA 2> error.file
$ cat error.file
cat: fileA: No such file or directory
```

- Redirect and append errors to a file:

```
$ cat fileA 2>> error.file
```

- Discard error output:

```
$ cat fileA 2> /dev/null
```

© Copyright IBM Corporation 2012

Figure 8-10. Error redirection

LX027.2

Notes:

Standard error can be independently redirected, using **2>**. There can be no space between the **2** and the **>**. The special file **/dev/null** is a bottomless pit where you can be redirect unwanted data. All data sent there is just thrown away.

/dev/null has the unique property of always being empty. It is commonly referred to as the *bit bucket*. The file descriptor table for the first error redirection example contains the following:

Descriptor 0 (standard input) :

- default: keyboard
- when command runs: keyboard

Descriptor 1 (standard output) :

- default: screen
- when command runs: screen

Descriptor 2 (standard error) :

- default: screen
- when command runs: file "error.file"

Combined redirection

- Combined redirects

```
$ cat < cities > cities.copy 2> error.file  
$ cat >> cities.copy 2>> error.file < morecities
```

- Association

- This redirects STDERR to where STDOUT is redirected.

```
$ cat cities > cities.copy 2>&1  
...writes both stderr and stdout to cities.copy  
  
be careful about this:  
$ cat cities 2>&1 > cities.copy  
...writes stderr to /dev/tty and stdout to cities.copy
```

© Copyright IBM Corporation 2012

Figure 8-11. Combined redirection

LX027.2

Notes:

With the association examples, the order in which redirections are specified is significant. In the first example, file descriptor 1 is associated with the file specified, outfile. Then the example associates descriptor 2 with the file associated with file description 1, outfile.

If the order of the redirection is reversed, the errors are redirected to the same place as standard out. But standard out at this point has not been redirected yet, so the default value is used, which is the screen. So, the error messages are redirected to the screen. Remember that the default error messages are sent to the screen.

In bash, using the sequence &> is roughly identical to appending 2>&1:

```
$ cat >output 2>&1
```

Is equivalent to:

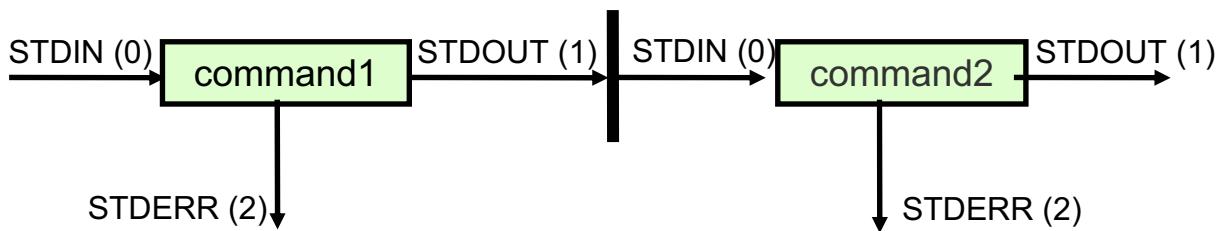
```
$ cat &>output
```

Pipes

- A sequence of two or more commands separated by a vertical bar (|) is called a *pipe* or *pipeline*.

```
$ ls -l | wc -l
```

- The standard output of command1 becomes the standard input of command2.



© Copyright IBM Corporation 2012

Figure 8-12. Pipes

LX027.2

Notes:

Two or more commands can be separated by a pipe or vertical bar (|) on a single command line. The requirement is that any command to the left of a pipe must send output to standard output. Any command to the right of the pipe must take its input from standard input.

Note that everything **ls** sends to standard out can now be counted by **wc**.

Filters

- A *filter* is a command that reads from standard in, transforms the input in some way, and writes to standard out. They can, therefore, be used at intermediate points in a pipeline.

```
$ ls | grep .doc | wc -l  
4
```

© Copyright IBM Corporation 2012

Figure 8-13. Filters

LX027.2

Notes:

A command is referred to as a filter if it can read its input from standard input, alter it in some way, and write its output to standard output. A filter can be used as an intermediate command between pipes.

A filter is commonly used with a string of piped commands, as in the example above. The **ls** command lists all the files in the current directory and then pipes this information to the **grep** command. The output of **grep** is piped to the **wc -l** command. The result is that the command counts the number of files in our current directory that contain *.doc*. In this example, the **grep** command is acting as a filter.

Common filters

- **grep:** Only displays lines that match a pattern
- **sed:** Allows string substitutions
- **awk:** Pattern scanning and processing
- **fmt:** Insert line wraps so that text looks pretty
- **expand, unexpand:** Change tabs to spaces and vice versa
- **tr:** Substitute characters
- **nl:** Number lines
- **pr:** Format for printer
- **sort:** Sort the lines in the file
- **tac:** Display lines in reverse order

© Copyright IBM Corporation 2012

Figure 8-14. Common filters

LX027.2

Notes:

Many programs can be used as a filter on an average UNIX system. The list in the visual shows the more common ones.

expand expands all **TAB** characters to spaces, ensuring that everything is aligned properly. The number of tab stops can be specified as a parameter. **unexpand** does exactly the opposite.

sed allows string substitutions. It works like the global editing facilities in **vi**, but doesn't edit a file. Instead, it filters standard input to standard output, executing the specified substitutions on the go. A short example: `sed s/old/new/ < oldfile > newfile` copies `oldfile` to `newfile`, changing all occurrences of `old` to `new`.

awk is a pattern scanning and processing language. **awk** scans each line of text and applies the necessary procedures to that line. A short example: `awk { print $1 } < infile` only prints the first field of each line of `infile` to STDOUT.

sed and **awk** are really powerful commands -- so powerful that some people refer to them as programming languages. O'Reilly has a book about them in case you're interested. They are also covered in the *AL32 (Linux Bash Programming)* course.

fmt is a text formatter. It takes unformatted text and formats them so that it looks pretty, inserting for instance line breaks, spaces and so forth where necessary.

tr converts individual characters. Here's how you convert all uppercase characters to lowercase: `tr ' [A-Z] ' '[a-z]'`.

grep scans individual lines for a pattern and only displays them if there is a match. It is very useful for filtering specific things out of a large file.

nl numbers all lines.

pr formats your output for a printer, adding headers, footers, page numbers and page breaks in the process.

sort sorts the lines in the file.

tac (inverse of **cat**) displays the lines in reverse order, last line first.

Split output

- The **tee** command reads standard input and sends the data to both standard output and a file.

```
$ ls | wc -l  
3  
  
$ ls | tee ls.save | wc -l  
3  
  
$ cat ls.save  
file1  
file2  
file3
```

© Copyright IBM Corporation 2012

Figure 8-15. Split output

LX027.2

Notes:

The **tee** command can be used to capture a snapshot of information going through a pipe. **tee** puts a copy of the data in a file as well as passing it to standard output to be used by the next command. **tee** does not alter the data flowing through the pipe.

Command substitution

- Command substitution allows you to use the output of a command as arguments for another command.
- Use backticks -- ` -- or \$() notation.

```
$ rm -i `ls *.doc | grep tmp`  
  
$ echo There are $(ps ax | wc -l) processes running.
```

© Copyright IBM Corporation 2012

Figure 8-16. Command substitution

LX027.2

Notes:

Command substitution allows you to use the output (STDOUT) of a command as arguments for another command. One common use is where you have a command which delivers a series of file names or user names, and need to use these file names or user names as arguments.

Two notations are possible: backticks (`) and the \$() notation.

Each example in the chart uses a different notation. They work as follows:

- The **ls** command generates a list of files which end in **.doc** in the current directory.
- The **grep** command filters the list of files, and only shows all lines (filenames) that contain the word **tmp**.
- The resulting list is used as arguments to the **rm** command.

In short, the command listed removes all files with names ending in **.doc**, and containing the string **tmp**.

The second example:

- The **ps** command generates a detailed list of processes running on the system, one per line.
- The **wc -l** command counts the lines (processes) from the output of the **ps** command
- The resulting output of the **wc** command is substituted in the string argument to the **echo** command.

Command grouping

- Multiple commands can be entered on the same line, separated by a semicolon (;).

```
$ date ; pwd
```

- Commands can be grouped into one input/output stream by putting curly braces ({}) around them.

```
$ { echo Print date: ; date ; cat cities; } | lpr
```

- Commands can be executed in a subshell by putting round braces () around them.

```
$ ( echo Print date: ; date ; cat cities ) | lpr
```

© Copyright IBM Corporation 2012

Figure 8-17. Command grouping

LX027.2

Notes:

Placing multiple commands separated by a semicolon (;) on a single line produces the same result as entering each command on a separate command line. There is no relationship between the commands, nor is any input or output redirection being done.

Commands can be grouped into one input/output stream by putting curly braces ({} and {}) around them. This combines their input/output streams into one.

You can also group commands into one input/output stream by putting round braces ((and)) around them. In this case, the grouped commands are executed in a subshell.

Shell variables

- Variables are part of the shell you are running.
- A variable has a unique name.
- The first character must not be a digit.
- To assign a value to a variable use `variable=value`.

```
$ var1="Hello class"  
$ var2=2
```

© Copyright IBM Corporation 2012

Figure 8-18. Shell variables

LX027.2

Notes:

Another feature of the shell is the expansion of shell variables. Shell variables can be set using the `variable=value` command (note that it is not allowed to put a space between the variable name and the equal sign). They are always stored as strings, even if the value is an integer. Their length is unlimited.

Referencing shell variables

- To reference the value of a variable, use \$variable.

```
$ echo $var1  
Hello class
```

```
$ echo $var2  
2
```

© Copyright IBM Corporation 2012

Figure 8-19. Referencing shell variables

LX027.2

Notes:

To reference a variable, use the \$variable expression.

Exporting shell variables

- The **export** command is used to pass variables from a parent to a child process by putting it in the environment of the child process.
- Changes made to variables in a child process do not affect the variables in its parent.

```
$ export x=4
$ bash
$ echo $x
4
$ x=100
$ echo $x
100
$ exit
$ echo $x
4
```

© Copyright IBM Corporation 2012

Figure 8-20. Exporting shell variables

LX027.2

Notes:

Variables by default are local to the shell they are defined in, which means that programs (including subshells) that are running as child process of this shell cannot reference the variables.

Only when a variable is exported, is it made available for all subsequent child processes (including subshells) too.

The **export** command exports a variable or lists exported variables if no parameters are provided.

If you change the value of a variable in a subshell, that change does not affect the parent process.

Standard shell variables

- The shell uses several shell variables internally.
- These variables are always written in uppercase.
- Examples include the following:
 - **\$**: PID of current shell
 - **PATH**: Path which is searched for executables
 - **PS1**: Primary shell prompt
 - **PS2**: Secondary shell prompt
 - **PWD**: Current working directory
 - **HOME**: Home directory of user
 - **LANG**: Language of user
- Overwriting these and other system variables by accident can cause unexpected results.
- Use lowercase variables in your shell scripts to avoid conflicts.

© Copyright IBM Corporation 2012

Figure 8-21. Standard shell variables

LX027.2

Notes:

The shell always defines and uses a large number of shell variables itself. These variables are almost always written with uppercase letters.

The most important shell variables for us are:

- **\$**: The Process ID of the shell.
- **PATH**: The search path for programs to be executed. If the user types a command and this command is not a built-in command and contains no indication of where it might be stored (such as `./command`, which indicates that the command is stored in the current directory), then all directories in the variable `$PATH` are searched (in order of appearance).

Note that if the current directory (indicated with a dot) is not part of the `$PATH` variable, then the current directory is not searched. This is considered a safety feature and therefore the default in most distributions.

- **PS1**: The primary command prompt. This prompt is shown when the shell is able to accept a command.

Special character sequences exist which can be used in this prompt, and which is expanded to, for instance, the current username, directory or time of the day. See the manual page of **bash** for details.

- **PS2:** The secondary command prompt. This prompt is shown for instance after the user ended a line with the line continuation character (backslash).
- **PWD:** The current working directory.
- **HOME:** The home directory of the user.
- **LANG:** The current language of the user. This variable is used for instance when sorting data (some countries list the é between e and f, and other countries list it after the z), when generating error messages and a lot of other things.

If you experience strange results, try `LANG=C`, which gives the default ASCII sorting order.

There are far more shell variables that the shell uses internally. If you overwrite them accidentally, you might experience strange problems. It is therefore a good idea to use lowercase variables in your own shell scripts.

Return codes from commands

- A command returns a value to the parent process. By convention, zero means success and a non-zero value means an error occurred.
- A pipeline returns a single value to its parent.
- The environment variable question mark (?) contains the return code of the previous command.

```
$ whoami  
tux1  
$ echo $?  
0  
$ cat fileA  
cat: fileA: No such file or directory  
$ echo $?  
1
```

© Copyright IBM Corporation 2012

Figure 8-22. Return codes from commands

LX027.2

Notes:

After a program exits, a return code is sent to the parent process. This number can thus be used to tell the parent the termination status of the child process. By convention, a return code of 0 means that the process ran correctly. A return code other than 0 indicates that something went wrong during the execution of the command.

After a program has run, the shell makes the return code available by using the shell variable \$?.

Quoting metacharacters

- When you want a metacharacter *not* to be interpreted by the shell, you need to quote it.
- Quoting a single character is done with the backslash (\).

```
$ echo The amount is US\$5
The amount is US$5
```

- Quoting a string is done with single ('') or double ("") quotes.
 - Double quotes allow interpretation of the dollar sign (\$), backtick (`), and backslash (\).

```
$ amount=5
$ echo 'The amount is $amount'
The amount is $amount
$ echo "The amount is $amount"
The amount is 5
```

© Copyright IBM Corporation 2012

Figure 8-23. Quoting metacharacters

LX027.2

Notes:

When you want a certain metacharacter not to be interpreted by the shell, you need to quote it. Quoting prevents the shell from interpreting metacharacters.

There are three ways a metacharacter can be quoted:

- The first method is by putting a backslash (\) directly in front of the character to be quoted. This ensures that the next character is passed onto the command without being interpreted. (The backslash itself is a regular metacharacter, which can be quoted too.)
- The second method is by using single quotes (''). This assures that any metacharacter within the quotes is being passed onto the command without being interpreted. It is useful if you've got a larger number of characters to quote.
- The last method is by using double quotes (""). This passes on any metacharacter within the quotes onto the command, except for the dollar (\$), the backtick (`) and the backslash (\). This allows you to use variables, command interpolation, and quoting non-metacharacters within a string.

Quoting non-metacharacters

- The backslash can also be used to give a special meaning to a non-metacharacter (typically used in regular expressions).
 - **\n**: New line
 - **\t**: Tab
 - **\b**: Bell
- A backslash followed directly by **Enter** is used for line continuation.
 - The continued line is identified with the \$PS2 prompt (default: >).

```
$ cat/home/tux1/mydir/myprogs/data/information/letter\
> /pictures/logo.jpg
```

© Copyright IBM Corporation 2012

Figure 8-24. Quoting non-metacharacters

LX027.2

Notes:

The backslash character can also be used to give a special meaning to a non-metacharacter. This is typically used in regular expressions, for instance when you do a **grep**. Some examples:

- **\n** is expanded to a new line.
- **\t** is expanded to the tab character.
- **\b** is expanded to the bell sign.

A backslash directly followed by the **Enter** key is used for line continuation.

Aliases

- The **alias** command allows you to set up aliases for often-used commands.
- For example:

```
$ alias ll='ls -l'  
$ alias rm='rm -i'  
To show all currently defined aliases:  
$ alias  
To delete an alias:  
$ unalias ll  
$ ll  
bash: ll: command not found
```

© Copyright IBM Corporation 2012

Figure 8-25. Aliases

LX027.2

Notes:

The shell also supports aliases. An alias is typically a short letter combination which expands into a far larger command. By making this large command an alias, you save yourself a lot of typing, especially if you use that command a lot.

Aliases are defined with the **alias** command. To show all aliases, use the **alias** command without an argument. To remove an alias, use the **unalias** command.

Most distributions define a number of aliases by default.

Unit review

- The shell is the command interpreter of Linux.
- The default shell in Linux is bash.
- A shell has a number of additional features, such as wildcard expansion, alias expansion, redirection, command grouping, and variable expansion.
- Metacharacters are a number of characters that have a special meaning to the shell.
- Reserved words are words that have a special meaning to the shell.

© Copyright IBM Corporation 2012

Figure 8-26. Unit review

LX027.2

Notes:

Checkpoint

1. True or False: A filter is a command that reads a file, performs operations on this file and then writes the result back to this file.

2. The output of the **ls** command is:

one two three four five
six seven eight nine ten

What will the output be if you run the **ls ?e*** command?

- a. three seven ten
- b. seven ten
- c. one three five seven eight nine ten
- d. ?e*

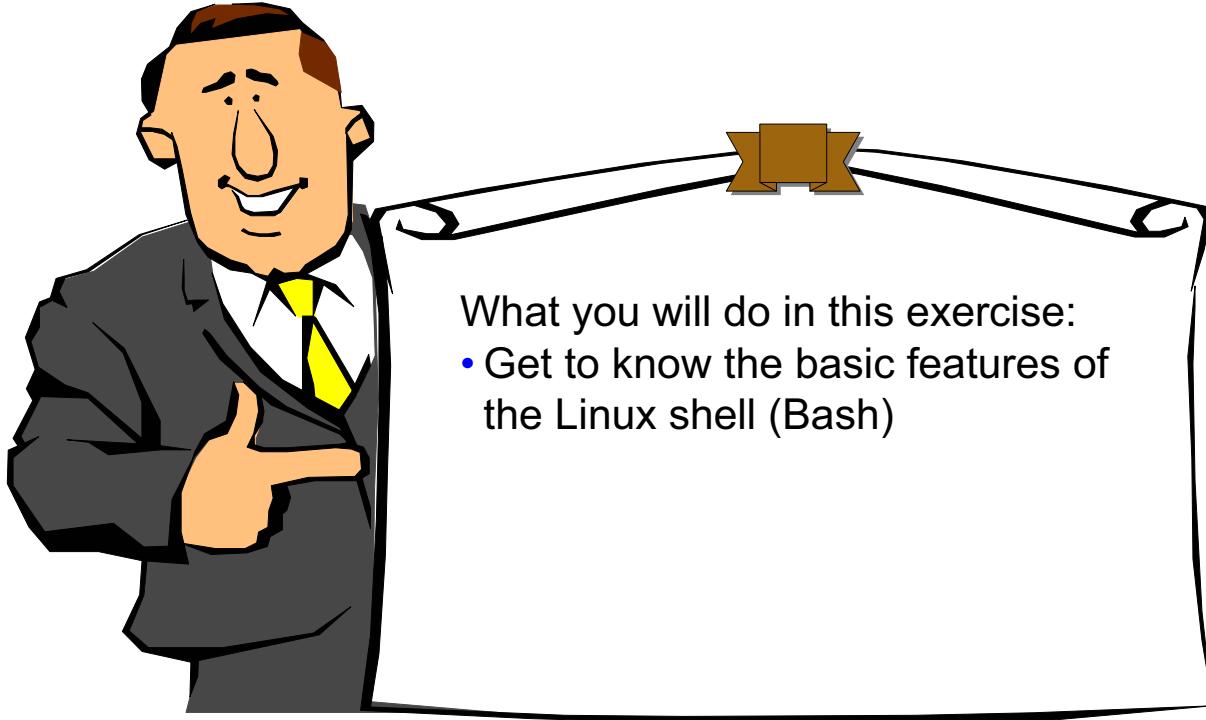
© Copyright IBM Corporation 2012

Figure 8-27. Checkpoint

LX027.2

Notes:

Exercise: Shell basics



© Copyright IBM Corporation 2012

Figure 8-28. Exercise: Shell basics

LX027.2

Notes:

Unit summary

Having completed this unit, you should be able to:

- Explain the function of the shell
- Discuss metacharacters and reserved words
- Use wildcards to access files with similar names
- Use redirection and pipes
- Use command substitution
- Describe and use the most common filters
- Group commands to control their execution
- Work with shell variables
- Use aliases
- Apply quoting

© Copyright IBM Corporation 2012

Figure 8-29. Unit summary

LX027.2

Notes:

Unit 9. Working with processes

What this unit is about

This unit describes defining a Linux process, the relationship between parent and child processes, the purpose of a shell, foreground and background processes, the concept of signals and how they are used to terminate processes, and the concept of priorities.

What you should be able to do

After completing this unit, you should be able to:

- Define a Linux process
- Describe the relationship between parent and child processes
- Explain the purpose of a shell
- Start foreground and background processes
- Explain the concept of signals and use them to terminate processes
- Explain the concept of priorities and manage them

How you will check your progress

- Checkpoint questions
- Machine exercises

Unit objectives

After completing this unit, you should be able to:

- Define a Linux process
- Describe the relationship between parent and child processes
- Explain the purpose of a shell
- Start foreground and background processes
- Explain the concept of signals and use them to terminate processes
- Explain the concept of priorities and manage them

© Copyright IBM Corporation 2012

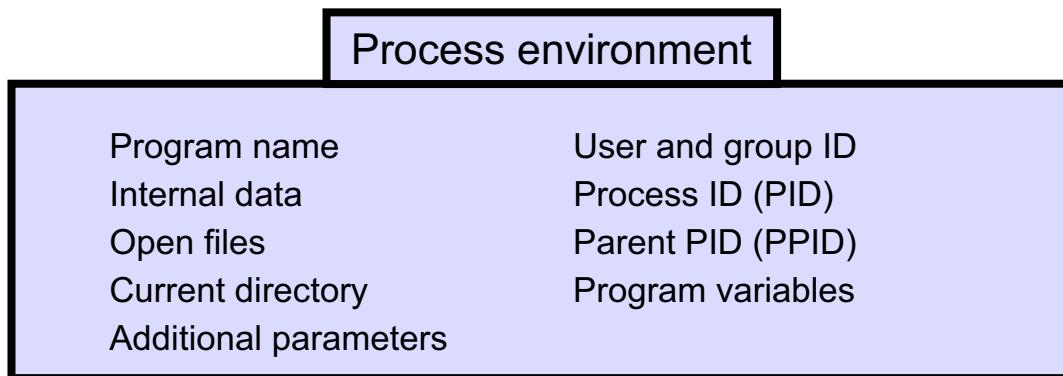
Figure 9-1. Unit objectives

LX027.2

Notes:

What is a process?

- A program is an executable file.
- A process is a program that is being executed.
- Each process has its own environment.



- To see the PID of your current shell process, type:
`$ echo $$`

© Copyright IBM Corporation 2012

Figure 9-2. What is a process?

LX027.2

Notes:

A program or a command that is actually running on a system is referred to as a process. Linux can run a number of different commands at the same time as well as many occurrences of the same program (such as Vi) at the same time.

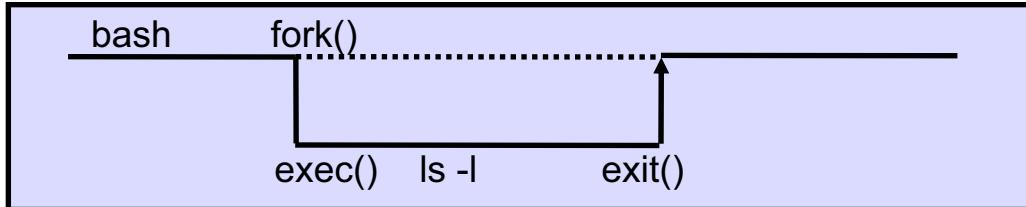
The Linux kernel holds an internal table, called the process table, in which the information about running processes is kept.

A shell is a special process that is able to read user commands and can start the appropriate program. One of the built-in commands of the shell is **echo**, which displays something on the screen, and one of the built-in shell variables is \$\$, which displays the process ID (PID) of the shell.

Starting and stopping a process

- All processes are started by other processes.
 - This is called a parent/child relationship.

```
$ ls -l
```



- A process can be terminated for two reasons.
 - The process terminates itself when done.
 - The process is terminated by a signal from another process.

© Copyright IBM Corporation 2012

Figure 9-3. Starting and stopping a process

LX027.2

Notes:

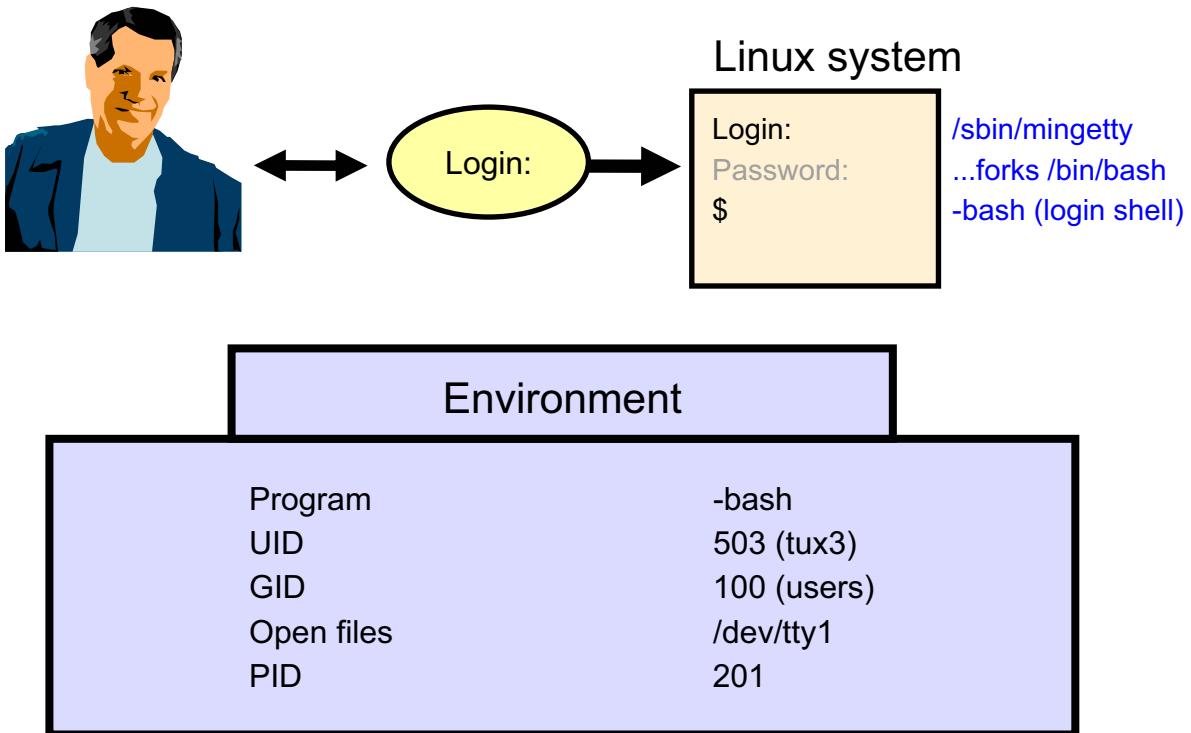
All processes in a Linux system are started by another process, so for each and every process you can identify the parent (the process that started this particular process) and the children (the processes that were started by this particular process), if any.

There is one exception to this. The **init** process is started by the kernel itself, and always has process ID 1. (Actually, there are a few more processes that are started by the kernel. These processes usually have PID 2, 3, and so forth, and their name usually starts with the letter *k*. The amount of processes and their names vary from kernel version to kernel version.)

Processes do not run forever. They can be terminated because of two reasons:

- Because the process terminates itself, either automatically (when the work has been done) or based on user input (such as a user entering ZZ in Vi).
- When another process sends a “signal” to the process.

Login process environment



© Copyright IBM Corporation 2012

Figure 9-4. Login process environment

LX027.2

Notes:

When a user approaches a Linux system and wants to start working with it, he or she is greeted with the login prompt. This login prompt is generated by the login process. This is not entirely true. The first login prompt is generated by getty, telnetd, sshd or another program that opens that particular tty. When the user types in his username, the login program is started with that username as parameter. The login program then asks for the password, and if authentication fails, it displays the second login prompt. Complicated, isn't it?) The user types the login name, and the login program asks for a password. If the user also types in the correct password, then the login program looks up the favorite shell of the user and starts this shell program. This first shell is called the login shell.

In a graphical environment, things work differently. The graphical login prompt is generated by a display manager¹. When users correctly authenticates himself, his window manager is started. The window manager can then start a terminal window, which in turn starts a shell. Because in a graphical environment more than one terminal window can be opened, a user can run multiple shells simultaneously.

¹ xdm, kdm, and gdm are the most common display managers.

Parents and children

```
$ echo $$  
561  
  
$ bash  
$ echo $$  
675  
  
$ date  
Mon Jan 1 22:28:21 UTC 2007  
$ <ctrl-d>  
  
$ echo $$  
561
```

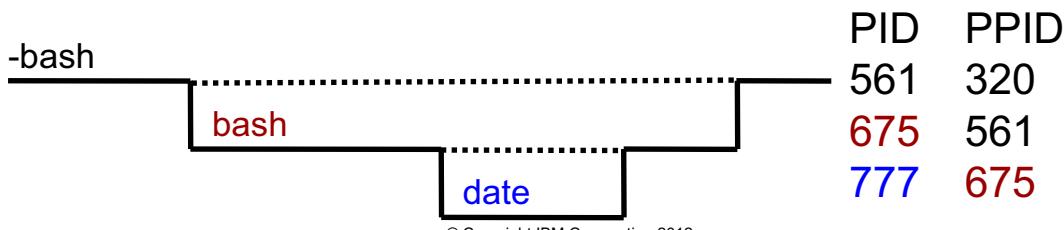


Figure 9-5. Parents and children

LX027.2

Notes:

The PID is the process identification number used by the kernel to distinguish the different processes. The PPID is the parent process identification number, or in other words, the PID of the process which started this one.

The special environment variable \$\$ identifies the PID of the current shell.

The **echo** command is built into the shell, so it doesn't need to create a subshell to be run.

In the example above, a second bash shell is started as a way to illustrate the parent/child relationship with processes. As another example, a second, different shell could be started (for example csh) to run specific shell scripts or programs.

Monitoring processes

- The **ps** command displays process status information.

```
$ ps aux
USER     PID %CPU %MEM   VSZ RSS   TTY STAT START TIME COMMAND
root      1  0.0  0.0 1336 436 ? S Jan1  0:05 init
root      2  0.0  0.0     0   0 ? SW Jan1  0:00 [keventd]
root      3  0.0  0.0     0   0 ? SW Jan1  0:05 [kapmd]
root      4  0.0  0.0     0   0 ? SW Jan1  0:05 [kswapd]
...
root 10248  0.0  0.1 2852 884 pts/2 R 13:47 0:00 ps aux
```

- ps** supports a large number of options; you typically use **ps aux**.
 - a:** All processes attached to a terminal
 - x:** All other processes
 - u:** Provides more columns

© Copyright IBM Corporation 2012

Figure 9-6. Monitoring processes

LX027.2

Notes:

ps prints process information. When no options are given, it only prints the processes that were started on your current terminal.

ps supports a large number of options. The most common invocation is **ps aux**, which displays all processes, with and without a tty (**a** and **x**) in a user-oriented format (**u**).

Viewing process hierarchy

- **pstree** shows process hierarchy.

```
$ pstree
init---apmd
|-atd
|-crond
|-gpm
|-httpd---10* [httpd]
|-inetd
|-kattraction.kss
|-kdm---X
|   `---kdm---kwm---kbgnwmm
|       |-kfm
|       |-kpanel
|       |-krootwm
|       |-kvt---bash---man---sh---gunzip
|           `---less
|       `---startkde---autorun
|-kflushd
```

© Copyright IBM Corporation 2012

Figure 9-7. Viewing process hierarchy

LX027.2

Notes:

pstree is a very simple tool which allows you to view the process hierarchy. It also supports a number of options that allow you to include the PID, for instance.

Controlling processes

- Processes can be controlled in two ways.
 - From the shell that started it, using its job number
 - From anywhere on the system, using its PID
- The following actions can be performed on a running process:
 - Terminate
 - Kill
 - Stop/continue
- These actions are performed by sending signals.

© Copyright IBM Corporation 2012

Figure 9-8. Controlling processes

LX027.2

Notes:

When a process has started, it can be controlled in two ways:

- From the shell that started the process, by referring to its job number
- From anywhere else on the system, by using its Process ID

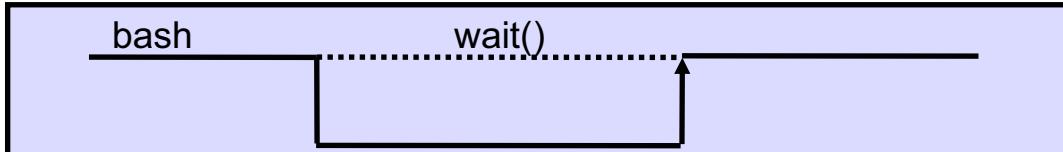
Various actions can be performed on a process, including terminate, kill, stop, and continue.

If a process was started from a regular, interactive shell, then the notion of foreground and background comes into play as well. A foreground process is a process that receives any keyboard input that is typed into the terminal. A background process does not receive any keyboard input. Only one process can run in the foreground at once.

Starting processes

- Foreground processes
 - Foreground processes are invoked by simply typing a command at the command line.

```
$ find / -name README
```



- Background processes
 - Background processes are invoked by putting an ampersand (&) at the end of the command line.

```
$ find / -name README &
```



© Copyright IBM Corporation 2012

Figure 9-9. Starting processes

LX027.2

Notes:

Processes that are started from and require interaction with the terminal are called foreground processes. As long as a foreground process runs, you are not able to run another command in the system using this shell. Processes that are run independently of the initiating terminal are referred to as background processes.

Background processes are most useful with commands that take a long time to run and do not need to interact with the user. A background process is started by ending the command line with a &. Normally, this must be the last character on the command line.

When a background process is started, you can see two numbers.

[1] 417

This means:

- [1] is the first process you are running in the background.
- 417 is the process ID of this process.

Job control in the bash shell

- <Ctrl-z>: Suspends foreground task
- **jobs**: Lists background or suspended jobs
- **fg**: Resumes suspended task in the foreground
- **bg**: Resumes suspended task in the background
- Specify a job number for bg, fg, and kill using %job

© Copyright IBM Corporation 2012

Figure 9-10. Job control in the bash shell

LX027.2

Notes:

You can stop a foreground process by pressing <Ctrl-z>. This does not terminate the process; it suspends it so that you can subsequently restart it.

To restart suspended processes in the background, use the **bg** command. To bring a suspended or background process into the foreground, use the **fg** command.

To find out what suspended/background jobs you have, issue the **jobs** command. This command shows you the job number of a process.

The **bg**, **fg**, and **kill** commands can be used with a job number. For instance, to kill job number 3, you can issue the command `kill %3`.

The **jobs** command does not list jobs that are started with the **nohup** command if the user has logged off and then logged back into the system. On the other hand, if a user invokes a job with the **nohup** command and then issues the **jobs** command without logging off, the job is listed.

Job control example

```
$ find / -name README &
[1] 2863
$ jobs
[1]+    running          find / -name README &
$ fg %1
/usr/share/doc/packages/grub/README
....
<Ctrl-z>
[1]+    stopped          find / -name README
$ bg 2863
$ jobs
[1]+    running          find / -name README &
$ kill %find
```

© Copyright IBM Corporation 2012

Figure 9-11. Job control example

LX027.2

Notes:

It is not mandatory to use job numbers with **fg**, **bg**, and **kill**. Look at the **bg** and **kill** commands to see how to use process IDs and process names.

Kill signals

- Several signals can be sent to a process.
 - Using keyboard interrupts (if foreground process)
 - Using the kill command
 - Synopsis: `kill -signal PID`
 - Using the killall command to kill all named apps
 - Synopsis: `killall -signal application`
- The following table lists the most important signals.

Signal	Keyboard	Meaning	Default action
01		Hangup	End process
02	Ctrl-C	Interrupt	End process
03	Ctrl-\	Quit	End process and core dump
09		Kill	End process - cannot be redefined - handled by kernel
15		Terminate	End process

© Copyright IBM Corporation 2012

Figure 9-12. Kill signals

LX027.2

Notes:

If you want to control a process from outside the shell (or other process) that started it, you need to use signals. Signals are the UNIX way of “nudging” a process into doing something.

When a process is running in the foreground, you can use keyboard interrupts (Ctrl-key) to send a signal. Otherwise, you need to use the **kill** or **killall** command to send a signal.

Most signals are delivered to the application itself. Technically, this means that the programmer of an application can write a special subroutine (called a signal handler) that is executed when a signal arrives. If the programmer did not write these special signal handlers, then the kernel performs the default action for that signal, which in most cases means that the application is terminated.

For us, only a few signals are important.

The hangup (01) signal is sent to a process if its parent dies, for example, if you log off when a background process is running. Most daemons (discussed later) redefine this signal to mean re-read configuration file.

The interrupt signal (02) is generated when the user presses the interrupt key (usually <Ctrl-c>) on the keyboard. The key is in different places depending upon the system and the terminal type.

Users pressing the quit key (usually <Ctrl-\>) generates the quit signal (03). Again, this is in different places on different systems.

The difference between <Ctrl-c> (signal 02) and <Ctrl-\> (signal 03), from a programmer standpoint is that <Ctrl-\> by default generates a so-called “core dump”. This is a file, usually called core, which contains the state of the program at the moment <Ctrl-\> was pressed. By using a debugger, such as **gdb**, the programmer can then use this core dump to figure out what is going on in the program. In all but a very few cases, a core dump is not interesting for a regular user and can be deleted.

The most powerful signal you can send to a process is a signal 9, because this signal is never delivered to the process, but handled by the Linux kernel immediately. A process can thus never redefine this signal. Processes which refuse to be killed by other signals can thus be killed with the `kill -9` command. There is a drawback to this, however: because the process is killed right away, it has no chance of writing data to disk and closing files correctly. This might lead to corrupted data. Use `kill -9` therefore only as a last resort.

The **kill** command by default sends signal 15 to a process. This is the regular terminate signal.

To list all the signals supported use the `kill -1` command. This list also shows you the names of signals. Instead of the signal number you could also use the signal name. `kill -9 <PID>` equals `kill -SIGKILL <PID>`.

Note that the number of the signal bears no resemblance to its strength or priority.

The **killall** command was created in Linux because most users do not know the PID of the process they want to kill, but only the name. This meant that they first needed to do a `ps` command to determine the PID of the process before they could issue the `kill` command. To short-circuit this, the **killall** command was invented. This command does not accept a PID, but rather expects the name of a running process as arguments. It then sends the signal to ALL processes with that name.

As an example, some misbehaving programs may leave child processes running when the master process ends. Assuming all of these child processes run using the same name as the master process, all of these child processes can be killed with a single command, such as `killall java`. Note the danger in a command like `killall bash`.

Running long processes

- The **nohup** command stops a process from being killed if you log off from the system before it completes by intercepting and ignoring the SIGHUP and SIGINTR (hangup and interrupt) signals and redirecting STDOUT and STDERR to a file.

```
$ nohup find / -name README &
nohup: appending output to `nohup.out'
$ logout
```

© Copyright IBM Corporation 2012

Figure 9-13. Running long processes

LX027.2

Notes:

nohup tells the process to ignore signals 01 and 03 (hangup and quit). This allows the process to continue if you log off the system.

Since all processes need to have an associated parent process, commands that start with **nohup** get the init process as the parent when you log off the system.

nohup is designed to be used for background processes as it has little meaning when used with a foreground process.

Managing process priorities

- Processes are scheduled according to priority.

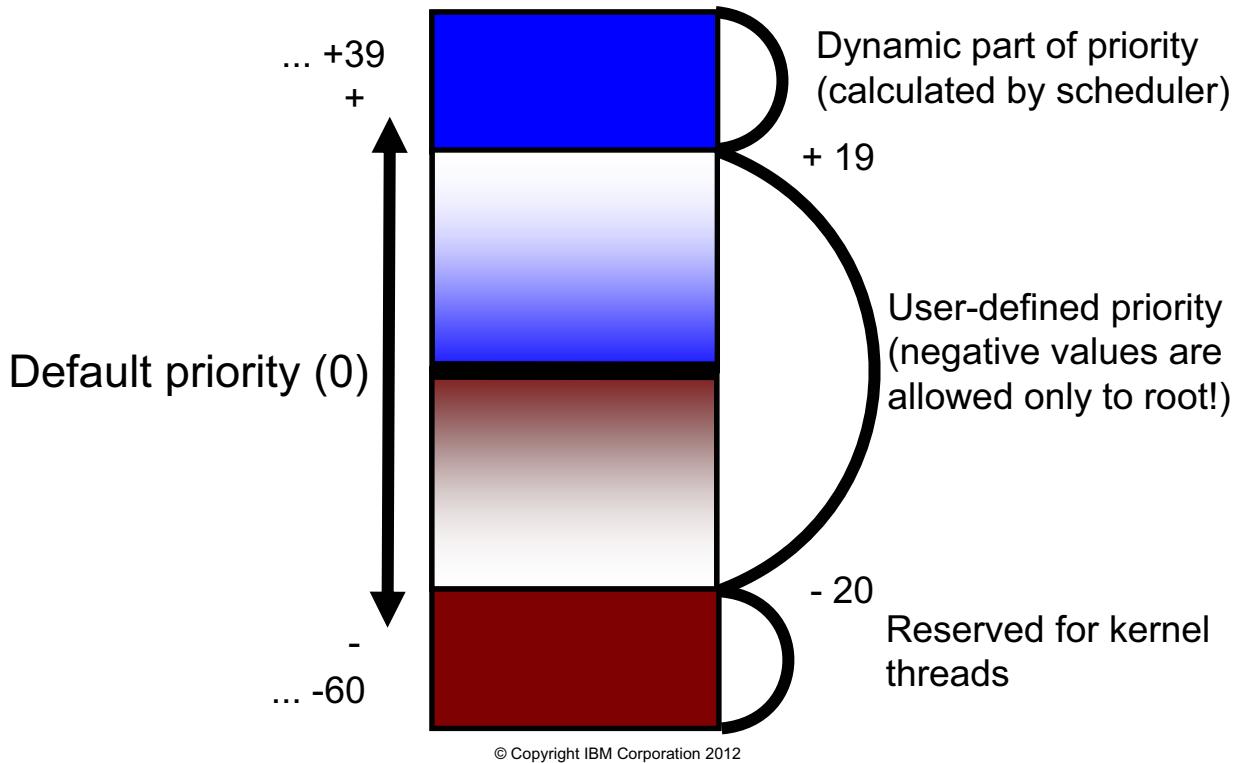


Figure 9-14. Managing process priorities

LX027.2

Notes:

Processes on a Linux system are scheduled according to a dynamic priority value: when a CPU is free to run a process, it looks through the process table for a process with the lowest priority number which is ready to run. This process then gets a timeslice on the CPU. The priority number of a process is continuously changed. Basically three factors influence this:

- The nice value for the process: The priority number of a process will never decrease below this number.
- After a process has had a certain amount of CPU time, its priority number is increased. This means that the next time a CPU becomes available, the process is less likely to be first in the list.
- After a process has been idle (not using CPU time) for a while (either because it is waiting for something to happen, or because other processes are keeping the CPU busy), the priority number is decreased.

This scheme results in a usage pattern where processes with the same nice value get equal amounts of CPU time. Processes with higher nice values get less CPU time than processes with lower nice values.

In reality, process scheduling is a lot more complex than what was written above. A lot of academic research has gone into optimizing scheduling, and a lot of that academic research has made its way into the kernel. There are several issues that need to be taken into account, in addition to the process priority and its nice value. A few of the more important issues are:

- Real-time processes are processes that need to have immediate access to the CPU as soon as something (e.g. an interrupt) happens. This cannot be delayed, for instance because it needs to control something in the physical world. The logic that controls robot arms is an example: You would not want your robot arm to continue moving beyond its stop point simply because somebody issued a difficult database query via a web server.
- Processes that last run on a specific processor of a multi-processor system should, if possible, run on the same processor the next time again. This preserves the cache contents of that processor and makes the processes generally run faster. This is known as "cache affinity".
- Sometimes processes are dependent on other processes. For instance your web server scripts may be dependent on a single database process. If this database process is delayed, the result is that tens, or possibly hundreds of other processes are also delayed. It therefore makes sense to give this database a higher priority, to reduce the overall latency of the system.

Linux uses a "Completely Fair Scheduler" that is optimized for most workloads, takes issues such as these into account, and uses very few CPU cycles itself.

The nice command

- The **nice** command is used to start a process with a user-defined priority.

```
nice [-n <value>] <original command>
```

```
$ nice -n 10 my_program &
[1] 4862
$ ps 1
F  UID  PID  PRI  NI  VSZ ... COMMAND
0  500 4372    20   0 4860 ... -bash
0  500 4862    30  10 3612 ... my_program
0  00 4863    20   0 1556 ... ps 1
```

© Copyright IBM Corporation 2012

Figure 9-15. The nice command

LX027.2

Notes:

To decrease the priority of a process when starting that process, use the nice command.

By default, nice sets a priority of 10 for a process, but this can be changed with the -n option.

Only root can set negative nice values.

Note that, because of the priority mechanism, even on a busy system, a process with a nice value of 20 gets some CPU time.

The renice command

- The **renice** command is used to change the priority of a currently running process.

```
renice <new_priority> <PID>
```

```
$ renice 15 4862
4862: old priority 10, new priority 15
$ ps 1
F  UID PID          PRI  NI          VSZ ... COMMAND
0  500 4372        20    0  4860 ... -bash
0  500 4862        35   15  3612 ... my_program
0  500 4868        20    0  1556 ... ps 1
```

© Copyright IBM Corporation 2012

Figure 9-16. The renice command

LX027.2

Notes:

When you want to change the priority of a process which is already running, use the **renice** command.

Integrated process management

- Various integrated tools exist for process management.
 - top, gnome-system-monitor, ksysguard
- Their availability depends on distribution.

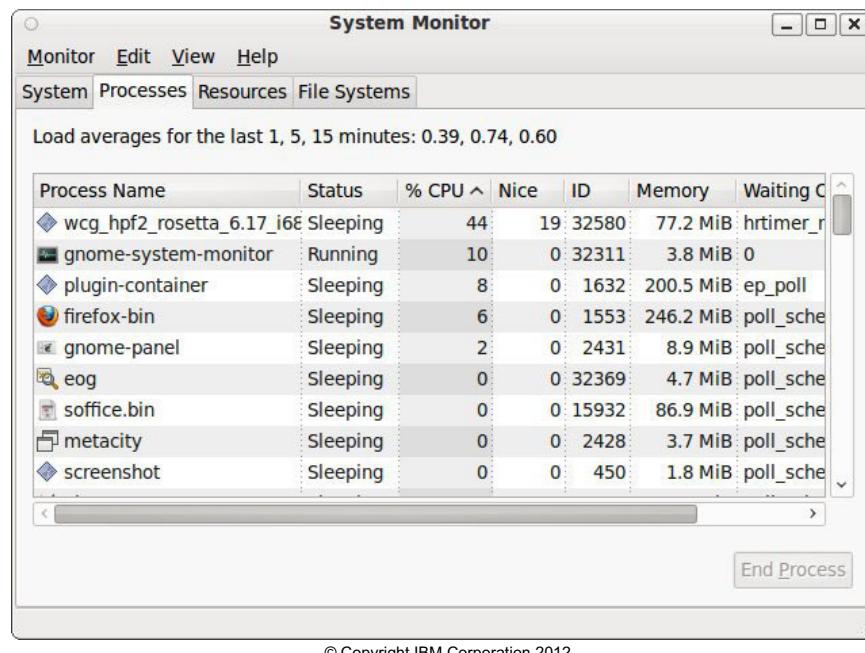


Figure 9-17. Integrated process management

LX027.2

Notes:

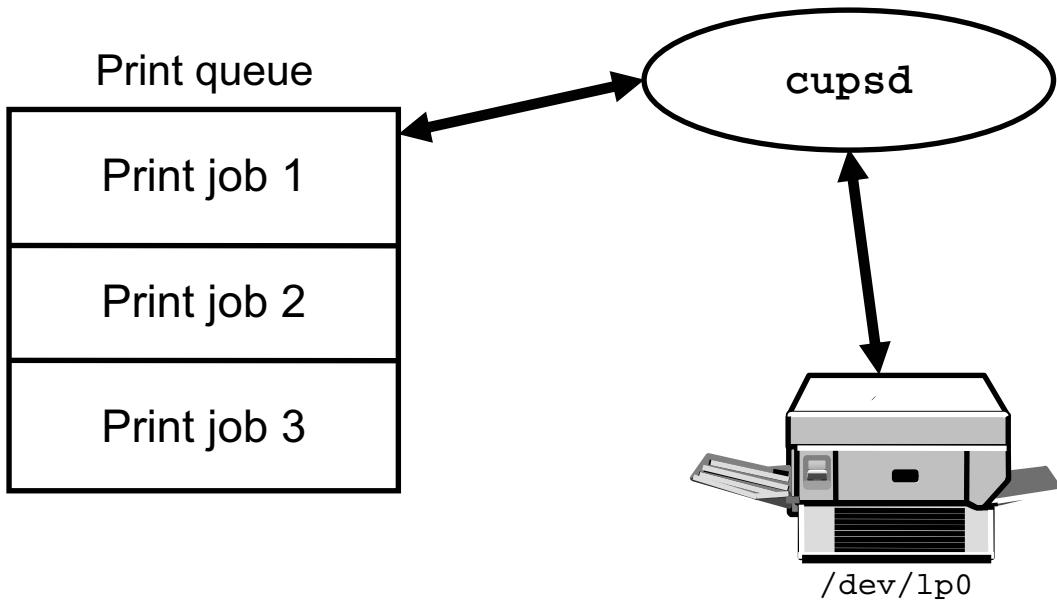
Various tools exist that offer integrated process management. Examples of these are **top** (which runs in a text terminal) and **gnome-system-monitor** and **ksysguard** (which run in a graphical environment).

These tools are configurable: you can select the things you want to see about each process, and you can sort the processes the way you want.

But the biggest advantage is that the display is refreshed every few seconds (the amount of seconds is configurable as well). Together with some generic information about the system (number of users, CPU usage, memory usage), this makes it useful for getting a quick impression of what the system is doing. Some system administrators keep these tools running all day, despite the CPU cost. (Running them typically costs about 2-5% CPU time.)

Daemons

- The word *daemon* refers to a never-ending process, usually a system process, that controls a system resource, such as the printer queue, or performs a network service.



© Copyright IBM Corporation 2012

Figure 9-18. Daemons

LX027.2

Notes:

A daemon (as in a sort of a friendly ghost or spirit, which guards your interests on your behalf) means a background process that typically starts when you start your system and runs until you shut it down. These processes are typically used to control access to a system resource, or to perform a network service.

cupsd (the CUPS daemon or Common UNIX Printing System) is one example of a daemon. **cupsd** tracks print job requests and the printers available to handle them. The **cupsd** daemon maintains queues of outstanding requests and sends them to the proper device at the proper time.

Technically speaking, daemons are nothing more than regular background processes. It's just the purpose that gives them another name.

Unit review

- All processes are started by a parent process (except for init, which is started by the kernel).
- Every process is identified with a process identifier (PID).
- A special process is the shell, which can interpret user commands.
- Processes can terminate by themselves or upon reception of a signal.
- Signals can be sent by the shell, using a keyboard sequence, or by the **kill** and **killall** commands.
- Processes are started with equal priority, but this can be changed using the **nice** and **renice** commands.
- A daemon is a background process that typically controls a system resource or offers a network service.

© Copyright IBM Corporation 2012

Figure 9-19. Unit review

LX027.2

Notes:

Checkpoint

1. True or False: Any user can send a signal to a process of another user and cause that process to halt.

2. If a process is hanging, the proper order of trying to terminate it with the lowest chance of data corruption is:
 - a. kill -1, <Ctrl-C>, kill, <Ctrl-\>
 - b. <Ctrl-Z>, kill, kill -9, kill -15, <Ctrl-C>
 - c. kill -9, kill -15, <Ctrl-C>, <Ctrl-Z>
 - d. <Ctrl-C>, <Ctrl-\>, kill, kill -9

3. What is a daemon?

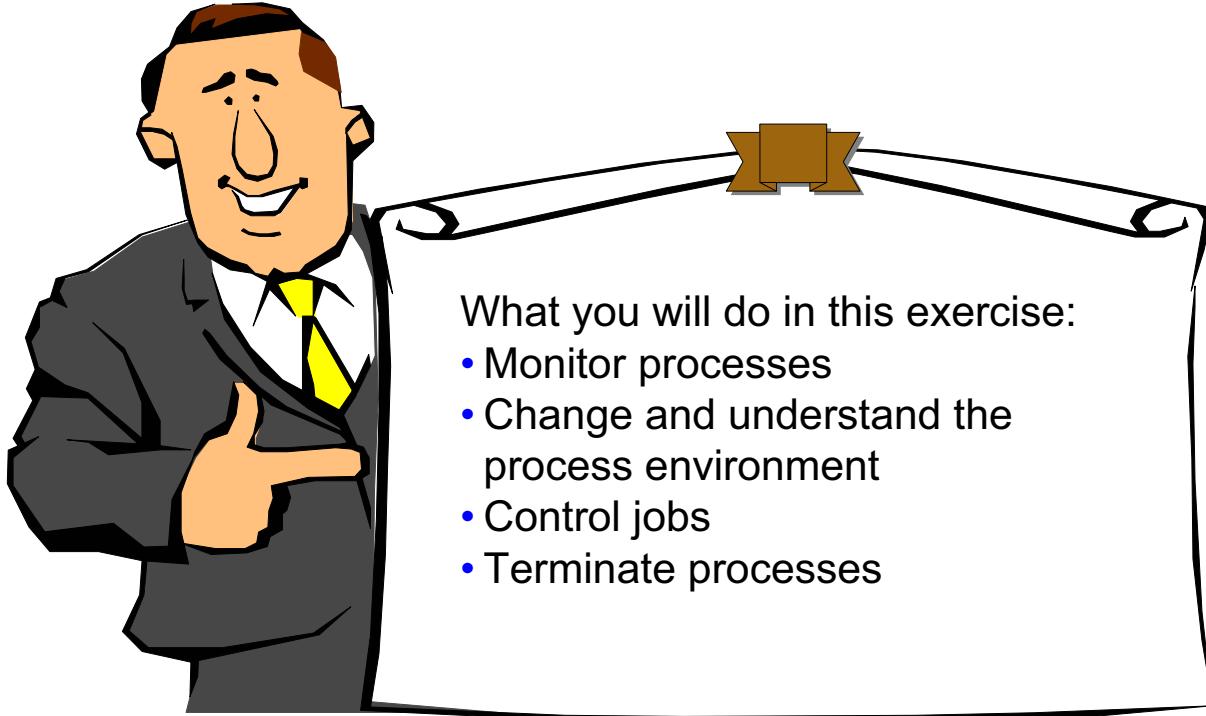
© Copyright IBM Corporation 2012

Figure 9-20. Checkpoint

LX027.2

Notes:

Exercise: Working with processes



© Copyright IBM Corporation 2012

Figure 9-21. Exercise: Working with processes

LX027.2

Notes:

Unit summary

Having completed this unit, you should be able to:

- Define a Linux process
- Describe the relationship between parent and child processes
- Explain the purpose of a shell
- Start foreground and background processes
- Explain the concept of signals and use them to terminate processes
- Explain the concept of priorities and manage them

© Copyright IBM Corporation 2012

Figure 9-22. Unit summary

LX027.2

Notes:

Unit 10. Linux utilities

What this unit is about

This unit discusses the following commands: **find**, **locate**, **grep**, **cut**, **sort**, **head**, **tail**, **type**, **which**, **whereis**, **file**, **join**, **paste**, **gzip**, **gunzip**, **zcat**, **bzip2**, **bunzip2**, and **bzcat**.

What you should be able to do

After completing this unit, you should be able to:

- Use the **find** and **locate** commands to search for files
- Use the **grep** command to search text files for patterns
- Use the **cut** command to list specific columns of a file
- Use the **sort** command to sort the contents of a file
- Use the **head** and **tail** commands to view specific lines in a file
- Use the **type**, **which**, and **whereis** commands to find commands
- Use the **file** command to find out the content of a file
- Use the **join** and **paste** commands to combine files
- Compress and uncompress files with **gzip**, **gunzip**, **zcat**, **bzip2**, **bunzip2**, and **bzcat**

How you will check your progress

- Checkpoint questions
- Machine exercises

Unit objectives

After completing this unit, you should be able to:

- Use the **find** and **locate** commands to search for files
- Use the **grep** command to search text files for patterns
- Use the **cut** command to list specific columns of a file
- Use the **sort** command to sort the contents of a file
- Use the **head** and **tail** commands to view specific lines in a file
- Use the **type**, **which**, and **whereis** commands to find commands
- Use the **file** command to find out the content of a file
- Use the **join** and **paste** commands to combine files
- Compress and uncompress files with **gzip**, **gunzip**, **zcat**, **bzip2**, **bunzip2**, and **bzcat**

© Copyright IBM Corporation 2012

Figure 10-1. Unit objectives

LX027.2

Notes:

The find command

- With the **find** command, you can search one or more directory structures for files that meet specified criteria.
- You can display the names of matching files or execute commands against those files.
- Syntax:
 - \$ find path expression

© Copyright IBM Corporation 2012

Figure 10-2. The find command

LX027.2

Notes:

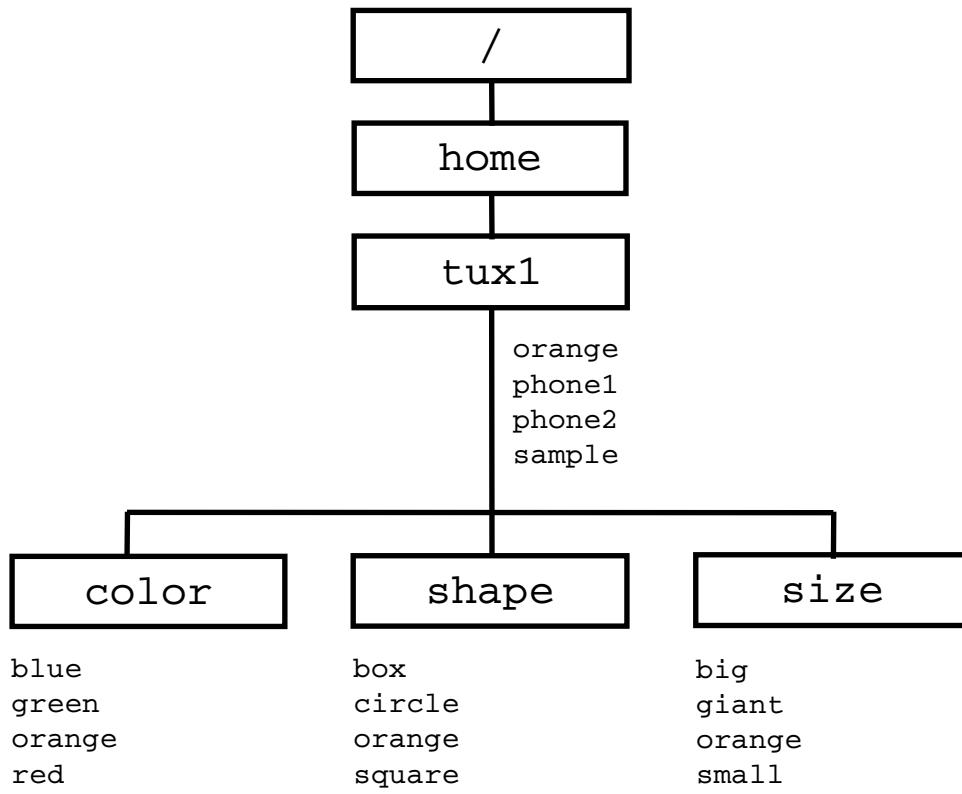
The **find** command recursively searches the directory tree for each specified path, seeking files that match a Boolean expression.

The output of the **find** command depends on the terms specified by the final parameter.

The basic syntax of the command could be written down as:

```
find <from where> <search for> <do something to it>
```

Sample directory structure



© Copyright IBM Corporation 2012

Figure 10-3. Sample directory structure

LX027.2

Notes:

Using find

- Generally, you want to search a directory structure for files with certain names and list the names found.

```
$ cd /home/tux1
$ find . -name orange
./orange
./color/orange
./shape/orange
./size/orange
```

© Copyright IBM Corporation 2012

Figure 10-4. Using find

LX027.2

Notes:

Note that the directory search is recursive meaning that **find** searches the current directory and all the subdirectories underneath it.

If not specified otherwise, the **find** command matches both directories and files.

The examples on the visual search for all files with the name orange, starting in the current directory (.).

Note that the -print option is the default and is not required. This was not always the case. In UNIX versions that have not yet implemented the POSIX standard for the **find** command, the -print option is required for the result to be displayed or used in a pipe.

Executing commands with find

- The -exec option executes a command on each of the file names found.

```
$ find . -name 's*' -exec ls -i {} \;
187787 ./sample
187792 ./shape/square
202083 ./size/small
```

- A set of curly brackets ({}) is a placeholder for each file name. The backslash (\) escapes the following semicolon (;).

© Copyright IBM Corporation 2012

Figure 10-5. Executing commands with find

LX027.2

Notes:

The command following the -exec option, in this case the **ls** command, is executed for each file name found. **find** replaces the {} with the names of the files matched. {} is used as a placeholder for matches.

Note the use of the escaped ";" to terminate the command that **find** is to execute. The \; is hard coded with the find command and is required for use with -exec and -ok options.

`$ find . -name 's*' -exec ls -l {} \;`

is equivalent to

`$ find . -name 's*' -ls`

Interactive command execution

- The -ok option also causes command execution but on an interactive basis.

```
$ find . -name o\* -ok rm {} \;
< rm ... ./orange > ? y
< rm ... ./color/orange > ? y
< rm ... ./shape/orange > ? n
< rm ... ./size/orange > ? y
```

© Copyright IBM Corporation 2012

Figure 10-6. Interactive command execution

LX027.2

Notes:

It is a good idea to use the -ok option rather than the -exec option if there are not a lot of files that match the search criteria.

Additional find options

-type	f d	Ordinary file Directory
-size	+n -n nc	Larger than <i>n</i> blocks Smaller than <i>n</i> blocks Equal to <i>n</i> characters
-mtime	+n -n <i>n</i>	Modified more than <i>n</i> days ago Modified less than <i>n</i> days ago Modified <i>n</i> days ago
-perm	<i>onum</i> <i>mode</i>	Access permissions match <i>onum</i> Access permissions match <i>mode</i>
-user	<i>user</i>	Finds files owned by <i>user</i>
-newer	<i>ref.file</i>	File was modified more recently than <i>ref.file</i>
-o -a		Logical OR Logical AND

© Copyright IBM Corporation 2012

Figure 10-7. Additional find options

LX027.2

Notes:

There are many other options to the **find** command, which are listed in the online manuals. Some options of **find** are:

- **-type**: Allow searches for only files or only directories.
- **-size**: Search for files that exactly match a size (**-size 10**), that are more than a size (**-size +10**) or that are below a certain size (**-size -10**). Size values are expressed in blocks, or with the **c** modifier, in bytes.
- **-mtime**: Search for files that have been modified in the time parameter supplied. The times are in days relative to the current day plus 24 hours. The times can be an exact match, older, or newer than the time specified.
- **-perm**: Search for files that have a certain permission mask (see **chmod**).
- **-newer**: Search for files that are newer than the reference file.
- **-o**: Allow multiple conditions to be matched (find a or b).
- **-a**: Require multiple conditions to be matched (find a and b).

find examples

```
$ find . -perm 777  
./size/giant  
  
File matches expr 1 and expr 2:  
$ find . -name 's*' -type f -a -size +2\  
>-exec ls -i {} \  
187787 ./sample  
187792 ./shape/square  
202083 ./size/small  
  
File matches expr 1 or expr 2:  
$ find . -name circle -o -name 'b*'  
.color/blue  
.size/big  
.shape/box  
.shape/circle
```

© Copyright IBM Corporation 2012

Figure 10-8. find examples

LX027.2

Notes:

The first example finds all the files (files and directories) that have their permissions set as 777 and were modified more than four days ago.

The second example searches for file names, not directory names, which are greater than 1024 bytes (two blocks of 512 bytes). When these have been found, the `ls -i` command is executed on them.

The third example shows you both files and directories that have circle as a name or whose name starts with a `b`.

locate command

- **locate** allows you to quickly find a file on the system based on simple criteria.

```
$ locate passwd
/usr/share/man/man1/passwd.1.gz
/usr/share/man/man5/passwd.5.gz
/etc/passwd
/usr/bin/passwd
```

- This requires that the superuser runs updatedb regularly.
 - Most distributions run updatedb automatically.
 - SuSE does not install locate/updatedb by default.

© Copyright IBM Corporation 2012

Figure 10-9. locate command

LX027.2

Notes:

The **locate** command also searches for files in the directory tree. There are two differences, if you compare **locate** to **find**:

- **locate** can only work with very simple criteria.
- **locate** uses a database which was created earlier.

This means that **locate** is faster in use, but requires a little effort to set up: The superuser has to run the **updatedb** command regularly (preferably every day or so) to keep the database up to date. Most distributions are configured to run **updatedb** every night.

The grep command

- Searches one or more files or standard input for lines matching a pattern
- Simple match or regular expression
- Syntax
 - `grep [options] pattern [file1 ...]`

© Copyright IBM Corporation 2012

Figure 10-10. The grep command

LX027.2

Notes:

The **grep** (Global Regular Expression Print) command searches for the pattern specified and writes each matching line to standard output.

The search can be for simple text, like a string or a name. **grep** can also look for logical constructs, called regular expressions, that use patterns and wildcards to symbolize something special in the text. Only lines that start with an uppercase T, for example.

The command displays the name of the file containing the pattern if more than one file is specified for the search.

grep sample data files

- Phone 1:

Chris	10300	internal
Jan	20500	internal
Lee	30500	external
Pat	40599	external
Robin	50599	external
Terry	60300	internal

- Phone 2:

Chris	1342	internal
Jan	2083	internal
Lee	3139	external
Pat	4200	internal
Robin	5200	internal
Terry	6342	external

© Copyright IBM Corporation 2012

Figure 10-11. grep sample data files

LX027.2

Notes:

This visual shows the sample files used to illustrate the examples that follows.

Basic grep

```
$ grep 20 phone1
Jan      20500      internal

$ grep 20 phone*
phone1:Jan      20500      internal
phone2:Jan      2083       internal
phone2:Pat      4200       internal
phone2:Robin    5200       internal
$ grep -v Jan phone2
Chris      1342       internal
Lee        3139       external
Pat        4200       internal
Robin     5200       internal
Terry      6342       external
```

© Copyright IBM Corporation 2012

Figure 10-12. Basic grep

LX027.2

Notes:

grep searches for the string given. If not specified otherwise, **grep** does not see the difference between a whole word matching the pattern or just a portion of a word matching the pattern.

The **-v** option reverses the working of **grep**; only lines that do not match are displayed.

grep with regular expressions

- Patterns with metacharacters should be in single quotation marks ('') so that the shell will leave it alone.
- Valid metacharacters with grep include \$. * ^ [-].
 - . Any single character
 - * Zero or more occurrences of the preceding character
 - [a-f] Any one of the characters in the range a through f
 - ^a Any line that starts with a
 - z\$ Any line that ends with z

© Copyright IBM Corporation 2012

Figure 10-13. grep with regular expressions

LX027.2

Notes:

The purpose of regular expressions on lines is the same as wildcards for file names.

When an asterisk (*) is used with the **grep** command to specify a regular expression, it matches zero or more occurrences of the previous character. If you want to use it to match zero or more arbitrary characters, it should be preceded by a dot, which means any single character.

Note that many metacharacters can have different meanings in regular expressions than they do in the shell. The following is a chart that compares **grep** metacharacters to the shells.

grep	grep Interpretation	Shell	Shell Interpretation
^	beginning of line	^	"old" Bourne pipe symbol
\$	end of line	\$	variable
.	any single character	?	any single character
.*	any number of any characters	*	any number of any characters
[-]	character range	[-]	character range

Regular expression

1. Display all processes that belong to user tux1:

\$ _____

2. Display all lines of the phone1 file (blank and non-blank):

\$ grep _____ phone1

3. Display only the lines of phone1 that contain an e and end in a 0:

\$ grep _____ phone1

© Copyright IBM Corporation 2012

Figure 10-14. Regular expression

LX027.2

Notes:

Answers:

1. To display all processes running on the system that belong to tux1, enter:

\$ ps aux | grep '^tux1 '

(Note the space after tux1.)

2. Display all lines of the phone1 file:

\$ grep '.*' phone1

3. Display only the lines of phone1 that contain an e and end in a 0:

\$ grep 'e.*0\$' phone1

grep options

- **-v** Print lines that do not match
- **-c** Print only a count of matching lines
- **-l** Print only the names of files with matching lines
- **-n** Number the matching lines
- **-i** Ignore case of letters when making comparisons
- **-w** Do a whole word search
- **-f <file>** Read expressions from file instead command line

© Copyright IBM Corporation 2012

Figure 10-15. grep options

LX027.2

Notes:

grep supports various useful options. Some of the most important ones are listed on the chart. These options can be combined. One of the most useful options is the **-v** option in combination with the **-f** option. This is typically used in logfile analysis: Logfiles are usually full of routine messages, and by putting these routine messages in an ignore file, you can filter your actual log file, discarding all routine messages, and displaying only the interesting messages, with this single command:

```
grep -v -f ignorefile logfile
```

Another useful trick is to do a full-text search over a large number of files, displaying the name of the file that matches automatically:

```
grep -l searchstring 'find $HOME -name "*.txt"'
```

Other greps

- **fgrep** allows only fixed strings (no regular expressions).
- **egrep** allows for multiple (alternate) patterns.

```
$ egrep '20500|40599|50599' phone1
Jan      20500    internal
Pat      40599    external
Robin    50599    external
```

- What does the following command do?

```
$ grep 30 phone1 | grep intern
????????
```

© Copyright IBM Corporation 2012

Figure 10-16. Other greps

LX027.2

Notes:

egrep is slightly slower than normal **grep** because it allows you to or patterns together using the vertical bar (|). To match all the patterns against a line takes a little bit more time than just matching one pattern.

fgrep is slightly faster because there is no interpretation that must take place first. Every regular expression must be evaluated first and this takes a little bit of time.

The answer to the question on this chart is: Display all lines in phone1 that contain both the string *30* and the string *intern*.

With a pipe from one **grep** to another, you can define an and-construct.

The cut command

- Pull selected columns or fields from one or more files
- Syntax:

```
cut -f(fields) -d(separator) file(s)  
cut -c(characters) file(s)
```

© Copyright IBM Corporation 2012

Figure 10-17. The cut command

LX027.2

Notes:

The **cut** command is used to pull columns out of text files.

The content of a file determines what syntax you should use.

The first syntax can be used when there is a special character used to mark the columns.

The second syntax is used when the output is arranged in visual columns.

cut example (1 of 2)

```
$ cat /etc/passwd
root:x:0:0:Big Brother:/root:/bin/bash
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
tux1:x:500:500:Tux the Penguin(1):/home/tux1:/bin/bash
tux2:x:501:501:Tux the Penguin(2):/home/tux2:/bin/bash

$ cut -f1,3,6,7 -d: /etc/passwd
root:0:/root:/bin/bash
shutdown:6:/sbin:/sbin/shutdown
tux1:500:/home/tux1:/bin/bash
tux2:501:/home/tux2:/bin/bash
```

© Copyright IBM Corporation 2012

Figure 10-18. cut example (1 of 2)

LX027.2

Notes:

/etc/passwd is divided into seven columns separated by a colon (:).

If you want to display only the first, sixth and seventh column, you could use the example on the chart. With the -f option, you specify which columns you want to see, and the -d option specifies the delimiter between columns.

What would be the output of the following command?

```
$ cut -f1 -d: /etc/passwd
```

The output would be:

```
r
shutd
tux1:!:500:500::/h
tux2:!:501:501::/h
```

This syntax can only be used correctly if the file is divided into columns that are separated with a special delimiter character.

cut example (2 of 2)

```
$ ps
  PID TTY STAT TIME COMMAND
  9374 p0 S    0:00 -bash
  14460 p0 R    0:00 ps

$ ps | cut -c-5,20-
  PID COMMAND
  9374 -bash
  14471 ps
```

© Copyright IBM Corporation 2012

Figure 10-19. cut example (2 of 2)

LX027.2

Notes:

In some files, the columns are not evenly divided by delimiters. The output of **ps** is an example of this. We see five columns, but what is the delimiter character between these columns? We cannot say that a space is the delimiter because the first line has two leading spaces, then PID. PID would thus be in the third column according to **cut -d' '**. However, on the second line, the actual PID has one leading space, putting it in the 2nd column, and the third line has no leading spaces, so PID would be in the first column.

If we run **ps | cut -f5 -d" "**, we would get this as output:

STAT

To overcome this problem we cannot use the syntax that defines columns with delimiters. Instead we have to tell **cut** what characters we want to see.

ps | cut -c-5,20- tells the **cut** command only to display the characters specified; characters 1 to 5 and characters 20 and further.

The sort command

- The **sort** command sorts the lines in the file specified and writes the result to standard output.

```
sort -tdelimiter -kfield -options file
```

```
$ cat animals
dog.2
cat.4
penguin.10

$ sort animals
cat.4
dog.2
penguin.10
```

© Copyright IBM Corporation 2012

Figure 10-20. The sort command

LX027.2

Notes:

To specify a delimiter with **sort**, use the **-t** option. This option has the same function as the **-d** option for **cut**. The **-t** option tells **sort** what character separates fields. This is often a : (colon), \t (tab) or \n (new line) character.

sort examples

```
$ sort -k1.2 animals
cat.4
penguin.10
dog.2
$ sort -t. -k2 animals
penguin.10
dog.2
cat.4
$ sort -t. -n -k2 animals
dog.2
cat.4
penguin.10
```

- Options
 - **-d** Sorts in dictionary order (only letters, digits, spaces considered in comparisons)
 - **-r** Reverses the order of the specified sort
 - **-n** Sorts numeric fields in arithmetic value

© Copyright IBM Corporation 2012

Figure 10-21. sort examples

LX027.2

Notes:

`$ sort animals` sorts the file `animals` starting with the first character of each line as the primary key.

`$ sort -k1.2 animals` forces a sort on the second character of each line.

`$ sort -t. -k2 animals` forces sort to sort on the second (-k2) column. Columns are separated here with a dot. Be aware that sort always tries to perform an ASCII (not necessarily numeric) sort, so 10 comes before 2, for example.

`$ sort -t. -n -k2 animals` performs a numeric sort on the second (+1) column.

The head and tail commands

- The **head** command can be used to view the first few lines of a file or files. The command syntax is:

```
$ head [-lines] file(s)
```

```
$ head -5 myfile
$ ls -l | head -12
```

- The **tail** command displays the last few lines of a file or files. The command syntax is:

```
$ tail [{-lines|-n lines|-n +lines|-f}] file(s)
```

```
$ tail -20 file
$ tail -n +20 file
$ tail -f file
```

© Copyright IBM Corporation 2012

Figure 10-22. The head and tail commands

LX027.2

Notes:

The **head** command shows you the first ten lines of a file by default. You can change the default by specifying a number to **head**.

The **tail** command can be used with either a positive or a negative number.

- n indicates the number of lines to read beginning from the end of the file. This displays the last *n* lines of the file.
- +n indicates the number of the line where you want to start displaying the lines.

The **tail -f** command can be used to monitor the growth of a file being written by another process. The -f option causes the **tail** command to continue to read additional lines from the input file as they become available.

For example: **tail -f logfile**

displays the last ten lines of the **logfile** file. The **tail** command continues to display lines as they are added to the **logfile**. The display continues until the interrupt key **<Ctrl-c>** is pressed. The -f option of **tail** can only be used when you specify a file. It cannot be used when **tail** has to read its input from STDIN.

The type, which, and whereis commands

- To find out what the path to a command is, use **type**.

```
$ type find echo
find is /usr/bin/find
echo is a shell builtin
```

- To find out where the binary is located, use **which**.

```
$ which find echo
/usr/bin/find
/bin/echo
```

- To locate the binary, source, and manual page files of a command, use **whereis**.

```
$ whereis find echo
find: /usr/bin/find /usr/share/man/man1/find.1.gz
echo: /bin/echo /usr/share/man/man1/echo.1.gz
```

© Copyright IBM Corporation 2012

Figure 10-23. The type, which, and whereis commands

LX027.2

Notes:

What if you are writing a program that uses **grep** and you must include the full path name, but you do not know where the command resides? The **type** command can tell you. When you type the name of a command, the shell searches for the command in your search path and runs the first one it finds. You can find out which copy of the program the shell runs by using the **type** utility.

An alternative to **type** is **which**. This command looks only in your search path. Note the different answers for **echo**. **which** does not know that **echo** is also a shell built-in. The reason for this difference is that **type** is a shell built-in itself, and **which** is not.

To locate a command, try using the **whereis** command, which looks in a few standard locations instead of using your search path. The **whereis** command also displays any manual page and source code files found.

The **type** command also reports on shell built-ins whereas the **whereis** command doesn't. A problem that could show up is shown on the chart. The **type** command tells you that the **echo** command is a shell built-in, but the **whereis** command tells you that there is an executable in /bin.

The file command

- With the **file** command, you can find out what the type of data is in the file.

```
$ file /etc/passwd /bin/ls /home/tux1 /tmp/fake.jpg
/etc/passwd:    ASCII text
/bin/ls:         ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), dynamically linked (uses shared libs
), for GNU/Linux 2.6.15, stripped
/home/tux1:      directory
/tmp/fake.jpg:   PDF document, version 1.5
```

© Copyright IBM Corporation 2012

Figure 10-24. The file command

LX027.2

Notes:

The **file** command can be used to determine the type of a file. This can be useful for a couple of reasons. First, it can tell you what files are readable before you potentially hang your screen by trying to display an executable file. Second, it can help you determine what kind of binary file it is and what operating system version it was compiled under.

The file command uses the `/usr/share/misc/magic` file to identify files that have some sort of magic number; that is, any file containing a numeric or string constant that indicates the type.

Using file on a non-existing file results in an error message stating that it could not get a file status.

You could use the file command to find out if a command is a shell script or an executable. To find out enter this:

```
$ file 'which command'
```

The join and paste commands

- The **join** and **paste** commands combine files.

```
$ cat one
a apple another
b bee beast
c cat
$ cat two
a ape
b broken
d dog
$ join one two
a apple another ape
b bee beast broken
$ paste one two
a apple another a ape
b bee beast      b broken
c cat      d dog
```

© Copyright IBM Corporation 2012

Figure 10-25. The join and paste commands

LX027.2

Notes:

The **join** and **paste** commands allow you to merge files together. They are rarely used, except for the most complicated shell scripts.

Compressing and uncompressing Files

- **gzip, gunzip, zcat; bzip2, bunzip2, bzcat**

```
$ ls -l file1
-rw-r--r-- [...] 34833 2009-05-13 12:33 file1
$ gzip -v file1
file1:      69.6% -- replaced with file1.gz
$ ls -l file1.gz
-rw-r--r-- [...] 10615 2009-05-13 12:33 file1.gz
$ zcat file1
(original contents of file1 displayed)
$ gunzip file1.gz
$ ls -l file1
-rw-r--r-- [...] 34833 2009-05-13 12:33 file1
$ bzip2 file1
$ ls -l file1.bz2
-rw-r--r-- [...] 10335 2009-05-13 12:33 file1.bz2
$ bzcat file1.bz2 | wc -c
34833
```

© Copyright IBM Corporation 2012

Figure 10-26. Compressing and uncompressing Files

LX027.2

Notes:

The **gzip** command compresses data, using the Lempel-Ziv coding (LZ77), to reduce the size of files. A compressed file replaces each file with a new compressed file with **.gz** appended to the original name. The compressed file retains the same ownership, modes and modification time of the original file. If compression does not reduce the size of the file, a message is written to stderr and the original file is not replaced.

The **-v** option writes the percentage of compression that occurred.

The **zcat** command allows the user to expand and view a compressed file without creating an uncompressed version of that file first. It does not rename the expanded file or remove the **.gz** extension. It simply writes the expanded output to stdout.

The **gunzip** command restores the original file that was compressed by the **gzip** command. Each compressed file is removed and replaced by the expanded copy. The expanded file has the same name as the compressed version without the **.gz** extension.

Files compressed with the **compress** command can also be uncompressed with **gunzip**.

The **bz** family of commands operates very similarly. The **bzip2** command compresses files using the Burrows-Wheeler block sorting text compression algorithm, and Huffman coding. **bunzip2** uncompresses **bzip2**-compressed files, and **bzcat** expands **bzip2**-compressed files to standard output.

Unit review

- The following commands were considered:
 - The **find** command is used to recursively search directories for files with particular characteristics.
 - The **grep** command is used to select entire lines containing a particular pattern.
 - The **head** and **tail** commands are used to view specific lines in a file.
 - The **sort** command sorts the contents of a file by the options specified.
 - Find out where you can find commands with **type**, **where**, and **whereis**.
 - The **gzip**, **gunzip**, **zcat**, **bzip2**, **bunzip2**, and **bzcat** commands can be used to create and work with compressed files.

© Copyright IBM Corporation 2012

Figure 10-27. Unit review

LX027.2

Notes:

Checkpoint

1. True or False: The command `ps -aux | grep tux | grep firefox` lists all Firefox processes of a user named tux.

2. Which command would best be used to locate all files in your system that begin with the string *team*?
 - a. `find / -name "^team"`
 - b. `find / -name "team*"`
 - c. `find / -name "*team*"`
 - d. `find / -type f -name "team"`

3. Translate the following command into your native language:

```
ls -lR|egrep "txt$|tab$" | sort -rn -k5|tail -n +4|head -5
```

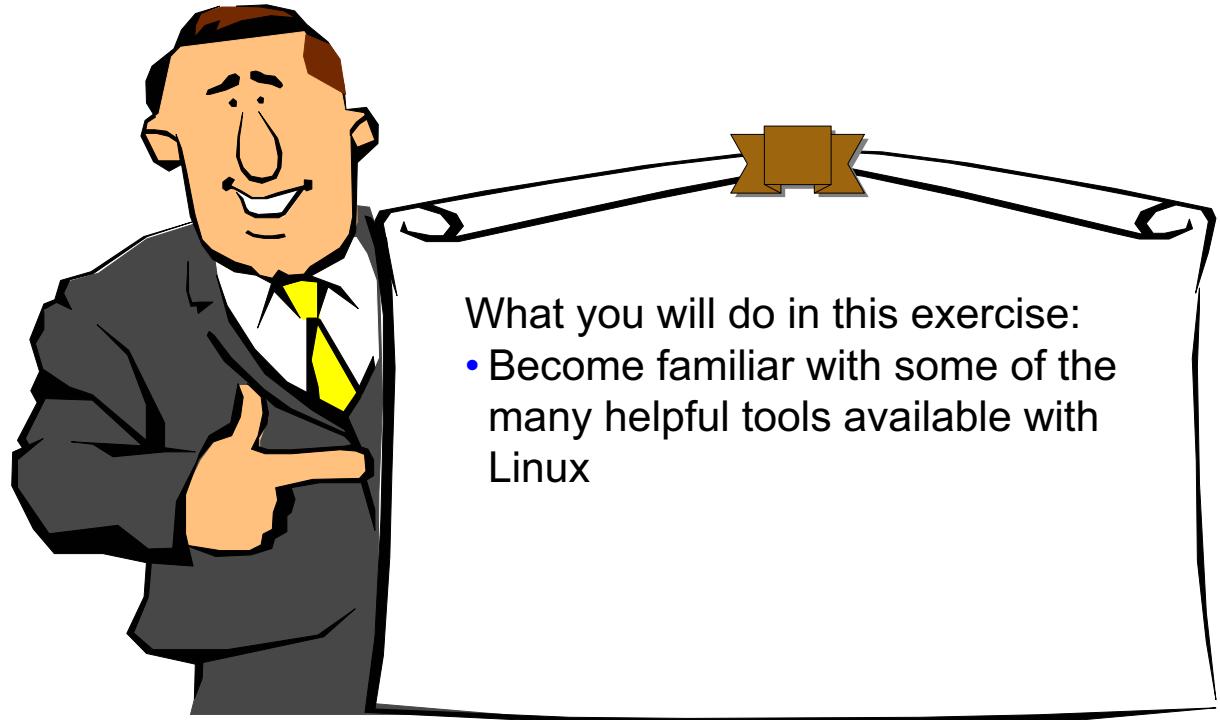
© Copyright IBM Corporation 2012

Figure 10-28. Checkpoint

LX027.2

Notes:

Exercise: Linux utilities



© Copyright IBM Corporation 2012

Figure 10-29. Exercise: Linux utilities

LX027.2

Notes:

Unit summary

Having completed this unit, you should be able to:

- Use the **find** and **locate** commands to search for files
- Use the **grep** command to search text files for patterns
- Use the **cut** command to list specific columns of a file
- Use the **sort** command to sort the contents of a file
- Use the **head** and **tail** commands to view specific lines in a file
- Use the **type**, **which**, and **whereis** commands to find commands
- Use the **file** command to find out the content of a file
- Use the **join** and **paste** commands to combine files
- Compress and uncompress files with **gzip**, **gunzip**, **zcat**, **bzip2**, **bunzip2**, and **bzcat**

© Copyright IBM Corporation 2012

Figure 10-30. Unit summary

LX027.2

Notes:

Unit 11. Shell scripting

What this unit is about

This unit shows how to invoke shell scripts, how to pass positional parameters to shell scripts, how to implement interactive shell scripts, and how to use conditional execution and loops.

What you should be able to do

After completing this unit, you should be able to:

- Invoke shell scripts in three separate ways and explain the difference
- Pass positional parameters to shell scripts and use them within scripts
- Implement interactive shell scripts
- Use conditional execution and loops
- Perform simple arithmetic

How you will check your progress

- Checkpoint questions
- Machine exercises

Unit objectives

After completing this unit, you should be able to:

- Invoke shell scripts in three separate ways and explain the difference
- Pass positional parameters to shell scripts and use them within scripts
- Implement interactive shell scripts
- Use conditional execution and loops
- Perform simple arithmetic

© Copyright IBM Corporation 2012

Figure 11-1. Unit objectives

LX027.2

Notes:

What is a shell script?

- A *shell script* is a collection of commands stored in a text file.

```
$ pwd  
$ date  
$ ls -l  
  
$ cat script1  
pwd  
date  
ls -l  
$
```

© Copyright IBM Corporation 2012

Figure 11-2. What is a shell script?

LX027.2

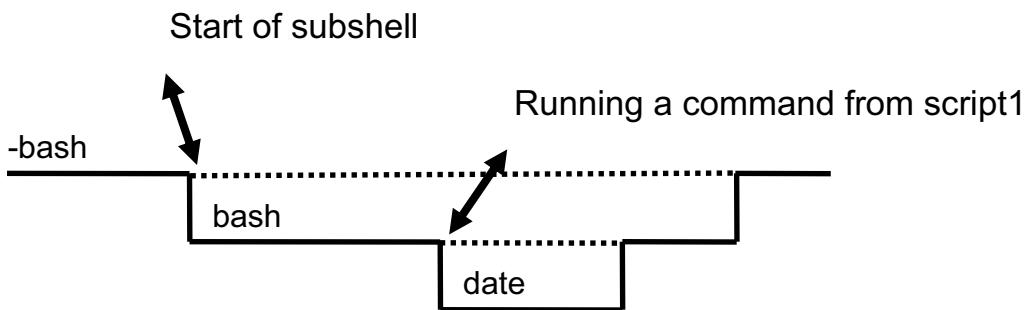
Notes:

A shell script basically is a collection of shell commands stored in a text file. This makes it easier to repeat a sequence of commands and is especially handy for automating your work.

Invoking shell scripts (1 of 3)

- The script does not have to be marked executable, but it must be readable.
- bash invokes a script in a child shell.

```
$ cat script1
date
$ bash script1
```



© Copyright IBM Corporation 2012

Figure 11-3. Invoking shell scripts (1 of 3)

LX027.2

Notes:

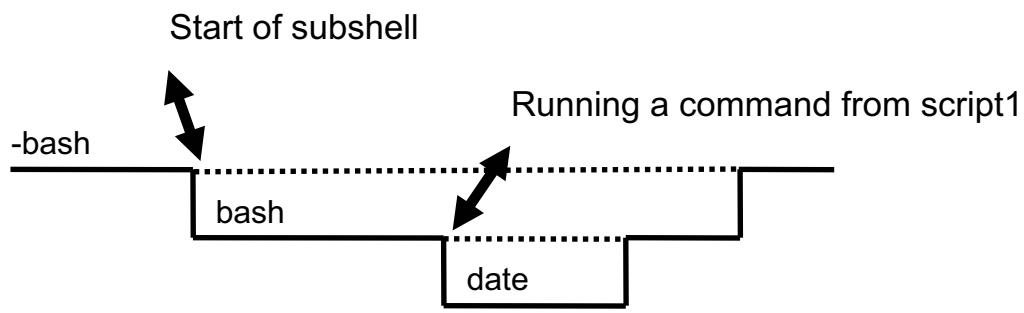
There are three ways of invoking a shell script. The chart shows the first method. With this method, a bash subshell is started with the script name as argument. Obviously, for this to work, the script needs to be readable. It does not have to be executable however, nor does it have to be in the PATH.

With this method, the script is executed within the shell that was started. This means that any environment variable changes are not propagated to the initial shell.

Invoking shell scripts (2 of 3)

- Use the **chmod** command to make the script executable.
- Then run the script as if it were a command.
- The script is run in a child shell.

```
$ chmod 755 ./script1
$ ./script1
```



© Copyright IBM Corporation 2012

Figure 11-4. Invoking shell scripts (2 of 3)

LX027.2

Notes:

The second method of invoking a shell script is by making it executable with **chmod**. This allows you to call the script directly. As with the first method, the script is executed in a subshell and thus any changes to environment variables are not propagated to the initial shell.

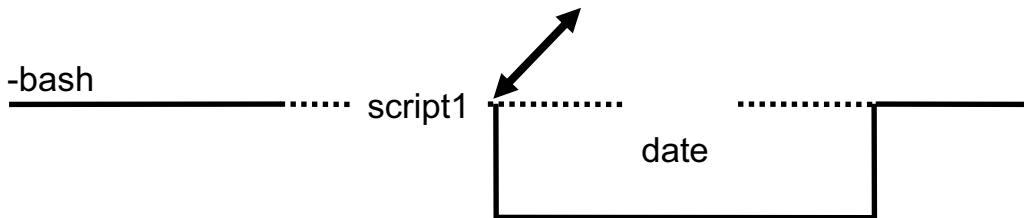
If you make sure that the script is located somewhere in your PATH, then you can invoke the script with just its script name. If the script is not in your PATH, you must invoke it with its relative or absolute path name.

Invoking shell scripts (3 of 3)

- Use the . (dot) or **source** command to execute the script in your current shell environment.
- Scripts executed with the dot command can change your current environment.

```
$ . script1  
$ source script1
```

The date command is invoked by your current shell.



© Copyright IBM Corporation 2012

Figure 11-5. Invoking shell scripts (3 of 3)

LX027.2

Notes:

The third method is by invoking the script using the . (dot) or **source** command. In this case, the script is executed in the current shell. This means that the script can make changes to environment variables in the current shell.

Invoking shell scripts in another shell

- To make sure the shell script always runs in the shell it was intended for (sh, bash, csh), use the following on the first line of your script:

```
#!/bin/bash
```

- The script will now always run in bash even if the user's default shell is something else.

© Copyright IBM Corporation 2012

Figure 11-6. Invoking shell scripts in another shell

LX027.2

Notes:

A shell is a very personal choice. Under Linux, most people prefer the bash shell, but people from an AIX background might prefer pdksh, and C programmers might prefer csh. If your script was written for the bash shell and contains bash-specific commands or constructs, you might want to make sure that the script is always invoked in a bash shell. This is done by adding the following line on top:

```
#!/bin/bash
```

When a shell (any shell) encounters a shell script that starts with the magic marker `#!`, it knows that this script is to be executed using the command that follows the magic marker, in our case `/bin/bash`.

The same magic marker can also be used to identify perl scripts (`#!/usr/bin/perl`), **awk** scripts (`#!/usr/bin/awk`), and so forth.

Typical shell script contents

- Handling of shell script arguments
- Complex redirection
- Conditional execution
- Repeated execution
- User interfacing
- Arithmetic

© Copyright IBM Corporation 2012

Figure 11-7. Typical shell script contents

LX027.2

Notes:

As said, a shell script is nothing more than a series of shell commands. Any shell command can be used in a shell script and vice versa. Having said that, there are a few things that are typically only found in shell scripts because using them on the command line would be silly or complicated. The rest of the unit covers these things.

Shell script arguments

- Parameters can be passed to shell scripts as arguments on the command line.
- These arguments are stored in special shell variables.
 - `$1, $2, $3 ...` refers to each of the arguments
 - `$@` is `"$1" "$2" "$3"`
 - `$*` is `"$1 $2 $3"`
 - `$#` is the number of parameters

```
$ cat ascript
#!/bin/bash
echo First parameter: $1
echo Second parameter: $2
echo Number of parameters: $#
$ ascript ant bee
First parameter: ant
Second parameter: bee
Number of parameters: 2
```

© Copyright IBM Corporation 2012

Figure 11-8. Shell script arguments

LX027.2

Notes:

Arguments, also called positional parameters, can be passed to shell scripts when the shell script is invoked. Within the shell script, they are available as special shell variables:

- The positional parameters themselves are available as `$1, $2, $3` and so forth. Positional parameters after number 9 must be referenced using the curly braces `{ }` and a number, like `${10}`.
- The curly brace notation for numbers above 9 is rarely used, because large numbers of parameters are typically handled using the shift command, which we discuss later.
- The amount of positional parameters is stored in `$#`.
- All positional parameters are stored in `$@` and `$*`. The only difference is the way they are stored:

`$@` is equal to `"$1" "$2" "$3" ...`

`$*` is equal to `"$1 $2 $3 ..."`

The difference is important when using `$@` and `$*` as parameters for another command.

Complex redirection

- To redirect fixed text into a command, use << END.

```
$ cat << END > cities
Atlanta
Chicago
END
```

- To avoid large argument lists, use xargs.

```
$ rm *.txt
-bash: /bin/rm: Argument list too long
$ find . -name "*.txt" | xargs rm
$
```

- Avoids large argument lists by running the command multiple times, if necessary.
- -0 (-zero) option to find and xargs needed when file names contain blanks or new lines.

© Copyright IBM Corporation 2012

Figure 11-9. Complex redirection

LX027.2

Notes:

In the previous units, we have already seen redirection using >, >> and <. This allows us to redirect input and output to a file. But what if we want to input some static content to a command? In that case, we can use the << operator. This allows us to specify the input to a command on the lines that follow that command in our script, until we reach the specified delimiter. In the visual, the content of the file cities would be:

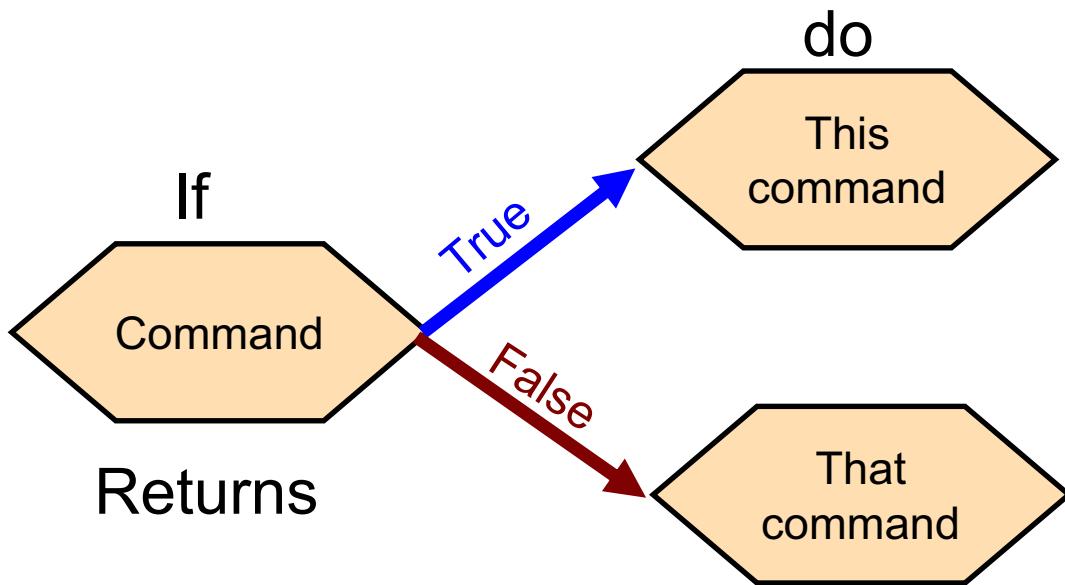
Atlanta
Chicago

END is the delimiter string. It is not included in the file. Note that this delimiter should start at the start of a new line. It is not permitted to put any characters (including spaces) in front of the delimiter.

Another form of complex redirection is the **xargs** command. It is used to avoid the situation where a command such as **cat 'ls *.txt'** would possibly dump so many filenames onto the argument list of **cat** that certain limits would be exceeded. With **xargs**, the arguments to use are fed from stdin, which can handle unlimited input, and the command to execute is executed as many times as necessary, without ever reaching shell limits.

Conditional execution

- The return code from a command or group of commands can be used to determine whether to start the next command.



© Copyright IBM Corporation 2012

Figure 11-10. Conditional execution

LX027.2

Notes:

Conditional execution means that the execution of a command or block of commands is dependent on the return code of another command.

There are two ways of making commands conditional:

- If you need to make one command conditional, then you can use the `&&` or `||` notation.
- If you need to make a block of commands conditional, then you can use the `if then else fi` notation.

In most cases, the return code to test is actually generated by the `test` command, which is a really versatile command for testing for files, strings, variables, and so forth.

The test command (1 of 2)

- The **test** command allows you to test for a given condition.
- Syntax:

```
test expression ...or: [ expression ]
```

```
$ test -f myfile.txt  
$ echo $?  
0
```

- Expressions to test file status:

— -f <file>	file is an ordinary file
— -d <file>	file is a directory
— -r <file>	file is readable
— -w <file>	file is writable
— -x <file>	file is executable
— -s <file>	file has non-zero length

© Copyright IBM Corporation 2012

Figure 11-11. The test command (1 of 2)

LX027.2

Notes:

The **test** command can be invoked in two ways:

- **test expression**
- **[expression]**

It does not matter which syntax you use. It is a question of taste and personal preference. Various expressions are possible. The charts in this visual and the next list a few of them.

The test command (2 of 2)

- String tests

-n <string>	string is not empty
-z <string>	string is empty
<string> == <string>	strings are equal
<string> != <string>	strings are not equal

- Arithmetic tests

<value> -eq <value>	equals
<value> -ne <value>	not equal
<value> -lt <value>	less than
<value> -le <value>	less than or equal
<value> -gt <value>	greater than
<value> -ge <value>	greater than or equal

© Copyright IBM Corporation 2012

Figure 11-12. The test command (2 of 2)

LX027.2

Notes:

The chart lists some more examples of **test** expressions. Remember that in each and every case the return value of the **test** command (\$?) is set to 0 if the test is positive, and 1 if the test is negative. Another thing to remember is to surround all your variables you use in a **test** expression with double quotes. As an example, consider what would happen with the following command:

```
$ test $var == "test"
```

If \$var is not empty, then the test is carried out and the return code is set to 0 or 1, depending on whether \$var actually contained the word *test*. But if \$var happens to be empty or not defined, then the shell changes this statement into:

```
$ test == "test"
```

This statement is obviously incorrect and gives you a syntax error, and a return code of 2. This could easily be prevented by using the following syntax:

```
$ test "$var" == "test"
```

Now, if \$var is empty or not defined, the statement is changed into:

```
$ test "" == "test"
```

This is a legal syntax. Obviously, the return code is 1.

The && and || commands

- **&&** and **||** (two vertical bars) can be used to conditionally execute a single command.

```
command1 && command2
```

```
if (command1 successful) then do (command2)
```

```
command1 || command2
```

```
if (command1 not successful) then do (command2)
```

```
$ [ -f testfile ] && rm testfile
$ [ -f lockfile ] || touch lockfile
$ [ "$TERM" = "xterm" ] && echo This is no tty
$ cat doesnotexist 2>/dev/null || echo \
> "Oh boy, this file does not exist."
```

© Copyright IBM Corporation 2012

Figure 11-13. The && and || commands

LX027.2

Notes:

The first way of conditionally executing commands is by using the **&&** and **||** operators. These operators allow a single command to be executed, depending on the return code of another single command.

The if command

- The structure of the basic if statement is:

```
if command-sequence returns true (0)
then
    carry out this set of actions
else
    carry out this set of actions
fi
```

```
$ cat myscript
if [ "$MY_VALUE" -eq 10 ]
then
    echo MY_VALUE contains the value 10
else
    echo MY_VALUE is not 10
fi
$
```

© Copyright IBM Corporation 2012

Figure 11-14. The if command

LX027.2

Notes:

The **if then else fi** construct allows you to execute multiple commands, based on the return code of a command.

You do not always need an **else** statement, but you can use only one within an **if** statement.

`command1 && command2`

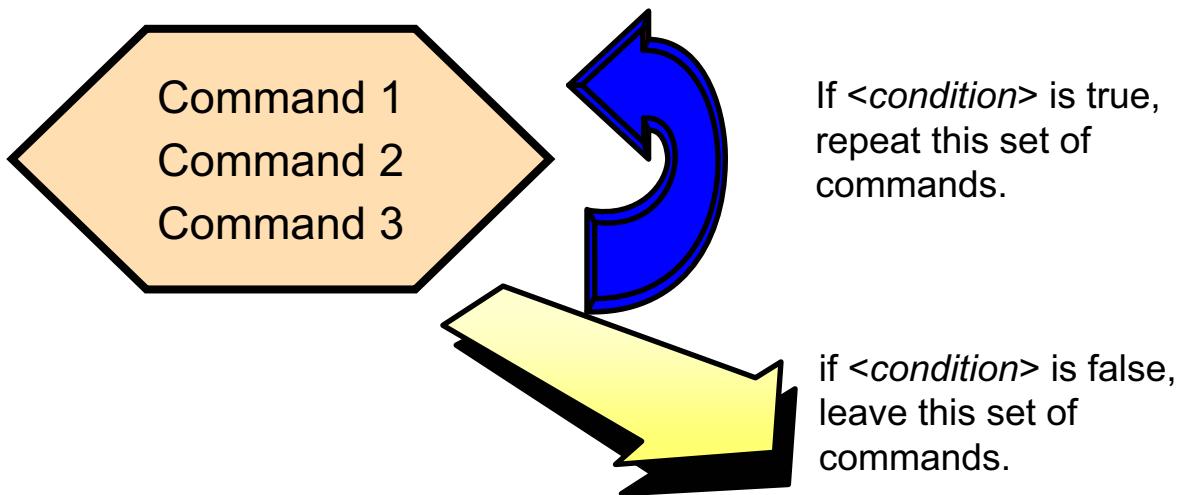
is the same as

```
if command1'
    command2'
```

However, the **if** statement is usually more readable, especially if there are a lot of commands to be executed.

Command repetition

- A *loop* is a set of commands that is executed over and over.
 - Until or while a certain condition is true
 - Or for each item from a list



© Copyright IBM Corporation 2012

Figure 11-15. Command repetition

LX027.2

Notes:

A *loop* is the programmers term for a set of commands that is executed over and over again. Loops in the bash shell can be of these two forms:

- Loops that run until or while a certain condition is true
- Loops that are executed for each item of a list

The while command

- The syntax of the **while** command:

```
while command-sequence-returns-true (0)
do
  commands
done
```

```
$ cat myloop
while true
do
echo "It is now $(date)"
echo "There are `ps aux | wc -l` processes"
sleep 600
done
$
```

Note that the command **true** always returns true (0)!

© Copyright IBM Corporation 2012

Figure 11-16. The while command

LX027.2

Notes:

The **while** loop is executed only while the expression evaluates true. By using the **true** argument with the **while** command, it forces the set of commands to be executed until the script is interrupted, for instance, with <Ctrl-c>.

The **sleep** command suspends execution of a process for the specified number of seconds, or until externally interrupted.

The expression used in the **while** loop can be any command, but in practice it is most often a **test** command, just like with **if**.

The for command

- The structure of the **for** loop is:

```
for identifier in list
do
commands to be executed on $identifier
done
```

```
$ cat my_forloop
for file in /tmp/mine_*
do
cp $file /other_dir/$file
done
$
```

© Copyright IBM Corporation 2012

Figure 11-17. The for command

LX027.2

Notes:

The **for** command sets the *identifier* variable to each of the values from the list in turn and executes the command block. Execution ends when the list is finished.

In the given example, the list for the **for** command has been formed by metacharacter expansion into certain file names in the /tmp directory.

Other examples are:

```
for fruit in Apple Banana Carrot
do
echo I would like a $fruit
done
```

And

```
for file in `find /home -perm 777'
do
echo Dangerous File Permissions on $file
done
```

Shifting shell script arguments

- If you expect a large number of shell arguments (for example, file names), use the **shift** command in a **while** loop to handle them all.

Variables:	\$1	\$2	\$3	\$4	\$5	\$# (count)
At start:	arg1	arg2	arg3	arg4	arg5	5
After first loop:	arg2	arg3	arg4	arg5	<i>unset</i>	4
After second loop:	arg3	arg4	arg5	<i>unset</i>	<i>unset</i>	3

```
$ cat make_backup
while [ $# -gt 0 ]
do
    cp $1 $1.bak
    shift
done
$
```

© Copyright IBM Corporation 2012

Figure 11-18. Shifting shell script arguments

LX027.2

Notes:

while and **for** loops are typically used to evaluate a large number of command line arguments. Shell scripts with a large number of arguments are typically shell scripts that are called with wildcards. In that case, the number of arguments is unpredictable and can, of course, be quite large.

There are basically two methods of evaluating such a large number of arguments: by using a **for** loop and by using a while loop. The for loop is the easiest and therefore not shown in the visual. It would look like this:

```
for file in @@
do
    cp $file $file.bak
done
```

Evaluating a large number of command line arguments using the **while** loop generally involves the **shift** command. This command shifts the arguments down by one (or perhaps to the left by one):

- \$2 is copied to \$1, \$3 is copied to \$2, and so forth.
- The last set variable is unset.
- \$# is decreased by one.

So, the basic construct works like this: As long as \$1 is not empty, perform the operations on \$1 and then execute the **shift** command. The chart shows how this is implemented in bash.

Although the **while** construct looks a little more complicated than the **for** construct, it is used more often for handling large numbers of arguments. The reason for this is that it gives more flexibility in that its loop test is re-evaluated on each loop iteration, not set at the beginning. As an example, your script might accept a number of options in addition to arguments. These should first be evaluated, possibly within a **while** construct, and then the arguments should be evaluated, again in their own **while** construct.

User interaction: The read command

- The **read** command reads one line from STDIN and assigns the values read to a variable.

```
$ cat delfile
#!/bin/bash
#Usage delfile
echo Please enter the file name:
read name
if [ -f $name ]
then
rm $name
else
echo $name is not an ordinary file -
echo so it is not removed
fi
```

© Copyright IBM Corporation 2012

Figure 11-19. User interaction: The read command

LX027.2

Notes:

The **read** command can be used to set more than one variable with a value. If more than one argument is given, the first argument would be assigned to the first variable name specified. The second argument would be assigned to the second variable and so on until the last argument is reached.

If there are more arguments supplied than variable names defined, the last variable name is given the value of all remaining arguments.

Usage is by programming convention. In this example, the usage statement is preceded by a **#**, which indicates that it is a comment.

The example does not test for the file permissions. This could be a worthwhile addition.

Arithmetic using let and \$()

- The bash shell can perform simple arithmetic on integers using the built-in **let** command or the `$((expr))` notation.
 - Operators: *, /, +, -, %

```
$ let x=2+3
$ echo $x
5
$ echo $(( 2+3 ))
5
$ let x=3*(3+5)
$ echo $x
24
$ let x=3*3+5
echo $x
14
$ x=$(( 3 * ( 3 + 5 ) ))
```

© Copyright IBM Corporation 2012

Figure 11-20. Arithmetic using let and \$()

LX027.2

Notes:

The bash shell allows you to do simple arithmetic using the **let** command. An alternative notation for this command is the `$(())` notation.

The **let** command and the `$(())` notation only works on integers and is limited by certain bounds, because it uses a “signed long int” variable type for internal representation. This means that, depending on the hardware architecture, the lowest number that can be stored is -2^{31} or -2^{63} , and the highest number is $2^{31} - 1$ or $2^{63} - 1$.

The operators that can be used are:

- An asterisk (*) for multiplications
- A slash (/) for division (results are rounded down to the nearest integer)
- A plus sign (+) for addition
- A minus sign (-) for subtraction
- A percent symbol (%) for the remainder of a division
- Left and right parenthesis () for argument grouping

According to the POSIX standard, arithmetic is done before wildcard expansion and command grouping. Because of this, the (,), and * characters do not need to be escaped, independent of whether you are using **let** or \$(()). However, experience has shown that not all versions of bash correctly implement this. If you want to be safe, make sure you surround your expression with double quotes.

Also, make sure that any shell variables you use indeed contain integers: they should only contain the digits 0-9.

Arithmetic using expr

- If your shell does not support \$(()) or let, use the **expr** command for integer arithmetic.
 - Not a shell built-in, therefore about 10 times slower (it rarely matters)
- Same operators as let:

```
$ echo `expr 3 + 5`
8
```

- Beware of the shell metacharacters *, (and)!

```
$ expr 3 * ( 3 + 5 )
bash: syntax error near unexpected token `('
$ expr 3 \* \( 3 + 5 \)
24
```

© Copyright IBM Corporation 2012

Figure 11-21. Arithmetic using expr

LX027.2

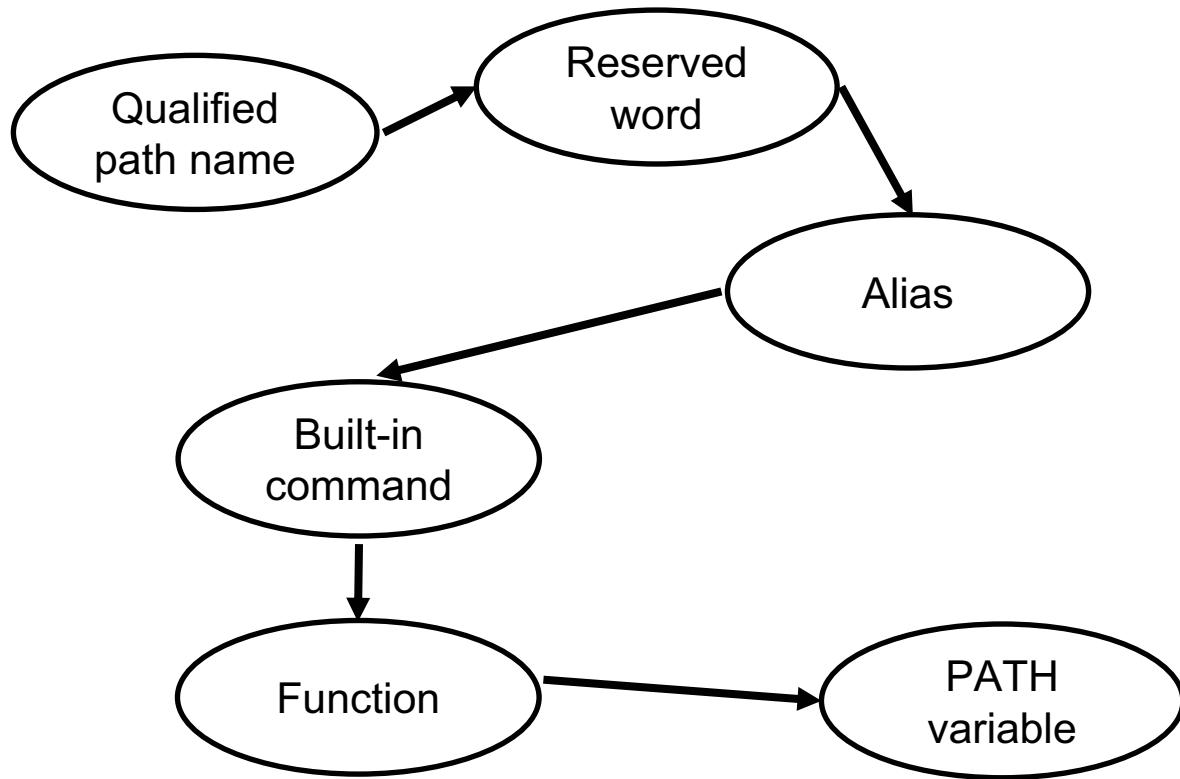
Notes:

Not all shells support the built-in **let** command or the \$(()) notation. In that case, you have to use the **expr** command. This is not a shell built-in and thus takes some extra time to execute. In practice, a script which does a lot of arithmetic is about 10 times slower when it has to use **expr**.

The usage of **expr** differs slightly from **let**, because **expr** cannot do assignments to environment variables directly. Instead, it prints the output of an arithmetic expression on stdout. This output can then be assigned to a variable by command evaluation.

Just as with **let**, **expr** also suffers from the drawback that it can only do calculations on integers, and has to stay within certain bounds (depending on the architecture and the version of **expr**) to work properly.

Command search order



© Copyright IBM Corporation 2012

Figure 11-22. Command search order

LX027.2

Notes:

You can see where the shell looks for the commands to be executed when it is ready.

Reserved words are those words that have a special meaning to the shell, such as: **if**, **then**, **else**, **while**, and so forth.

Aliases are set and managed with the **alias** and **unalias** commands.

Built-in commands are those commands that are part of the shell. Examples include **cd**, **umask**, **read**, and **echo**. If you cannot find a command in the manual pages, try searching in the manual page of the bash shell.

Functions have not been covered in this unit. They can be thought of as shell scripts within shell scripts.

The PATH variable is the last thing searched.

Unit review

- Positional parameters are used to pass to scripts the values from the invoker; they are also in \$* or \$@.
- To test for a particular condition, the **test** command can be used. This feature is frequently coupled with the **if** statement to control the flow of a program and allow for conditional execution within scripts.
- The **read** command can be used to implement interactive scripts.
- The **while** and **for** commands are used to create loops in a script.
- Simple integer arithmetic can be performed by the **expr** or **let** commands or the \$(()) notation.

© Copyright IBM Corporation 2012

Figure 11-23. Unit review

LX027.2

Notes:

Checkpoint (1 of 2)

1. What will the following piece of code do?

```
TERMTYPE=$TERM
if [ -n "$TERMTYPE" ]
then
    if [ -f /home/tux1/custom_script ]
    then
        /home/tux1/custom_script
    else
        echo No custom script available!
    fi
else
    echo You don't have a TERM variable set!
fi
```

© Copyright IBM Corporation 2012

Figure 11-24. Checkpoint (1 of 2)

LX027.2

Notes:

Checkpoint (2 of 2)

- 2.** Write a script that will multiply any two numbers together.

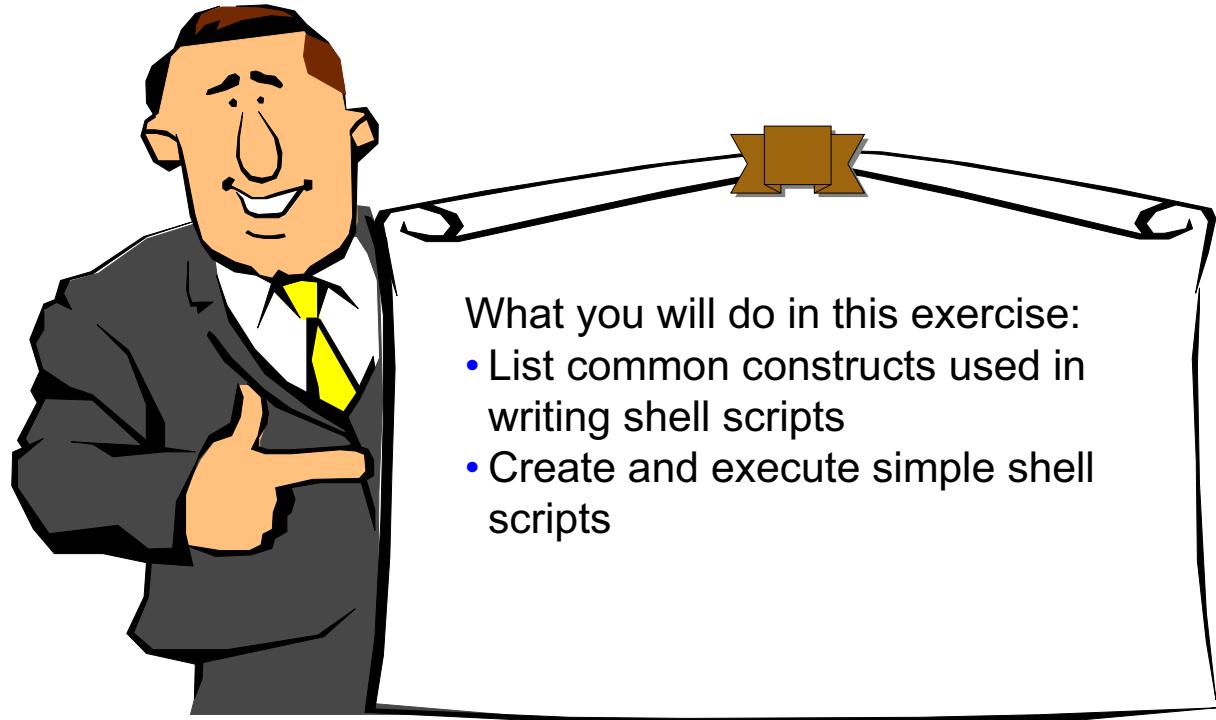
© Copyright IBM Corporation 2012

Figure 11-25. Checkpoint (2 of 2)

LX027.2

Notes:

Exercise: Shell scripting



© Copyright IBM Corporation 2012

Figure 11-26. Exercise: Shell scripting

LX027.2

Notes:

Unit summary

Having completed this unit, you should be able to:

- Invoke shell scripts in three separate ways and explain the difference
- Pass positional parameters to shell scripts and use them within scripts
- Implement interactive shell scripts
- Use conditional execution and loops
- Perform simple arithmetic

© Copyright IBM Corporation 2012

Figure 11-27. Unit summary

LX027.2

Notes:

Unit 12. The Linux GUI

What this unit is about

This unit describes the main components of the X Window System, the function of the X Server, the function of window managers, and the main characteristics of desktop environments. This unit also explains how to switch between GNOME and KDE.

What you should be able to do

After completing this unit, you should be able to:

- List the main components of the X Window System
- List the function of the X Server
- List the function of a window manager
- List the main characteristics of desktop environments
- Switch between GNOME and KDE

How you will check your progress

- Checkpoint questions
- Machine exercises

Unit objectives

After completing this unit, you should be able to:

- List the main components of the X Window System
- List the function of the X Server
- List the function of a window manager
- List the main characteristics of desktop environments
- Switch between GNOME and KDE

© Copyright IBM Corporation 2012

Figure 12-1. Unit objectives

LX027.2

Notes:

The Linux graphical user interface

- The X Window System is the GUI of Linux.
 - Developed at MIT in 1984
 - Current standards body: X Consortium
 - Short name: X
- X uses client-server model with network connections.
 - Highly flexible
 - Easy exchange of components
 - Supports networked applications and sessions, independent of the OS

© Copyright IBM Corporation 2012

Figure 12-2. The Linux graphical user interface

LX027.2

Notes:

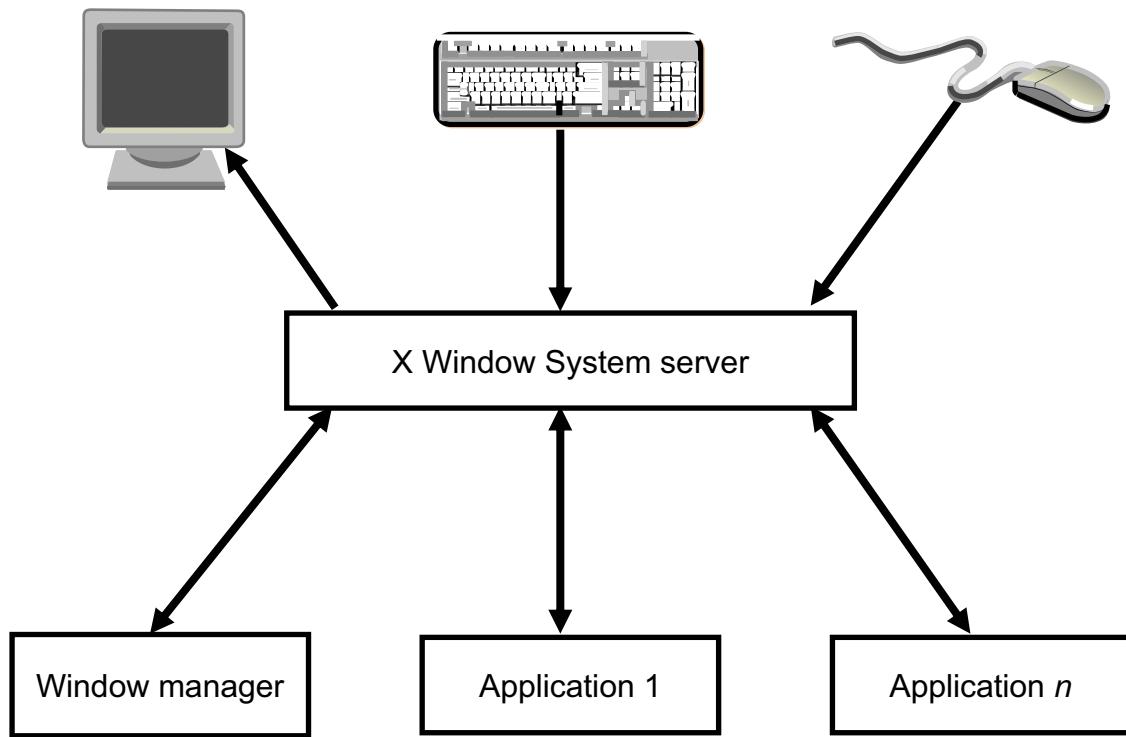
The X Window System, called X for short, is a network-based graphics system that was developed at the Massachusetts Institute of Technology (MIT) in 1984. In 1985, MIT released X (version 9) to the public, license free. It was designed as a generic, UNIX-oriented basis for graphical user interfaces (GUIs). Prior to X, the only way to communicate with a UNIX system was using commands in an ASCII environment.

In 1987, a group of vendors and researchers formed the X-Consortium to continue work on this windowing system. X version 11 (X11) was released in 1987, and continues to be the version of X that is used. There have been several releases of X, the most current being release 6 (1994), better known as X11R6 or Xorg.

Linux doesn't use the original X Window System but an implementation called XFree86.

X is an open standard that heavily uses standard TCP/IP network connections. This makes it ideal in a mixed UNIX/Linux environment, because it allows applications written for and running on Linux to display their windows on any other X-capable system. In fact, applications exist that can capture Microsoft Windows application windows and display them on an X capable workstation over a network connection.

Client/server architecture



© Copyright IBM Corporation 2012

Figure 12-3. Client/server architecture

LX027.2

Notes:

The X Window System uses a client/server architecture, which makes it very flexible. The central piece of software is the X Server. The X Server traps all keyboard and mouse events and sends them to the appropriate application. If an application wants to put something on the screen, it sends that data to the X Server, which then performs the necessary hardware calls to the graphical adapter.

Any application can connect to the X Server, but there should always be one special application active: the window manager. The window manager basically puts a border around each application window and allows you to drag windows around. Another task of the window manager is to allow you to resize windows.

There are numerous window managers available, each with its own style. The most popular window managers are the window managers that come with the GNOME and KDE projects (discussed later), but other window managers may also be present or can be downloaded from the Internet: fvwm, fvwm95, twm, mwm, olvwm, afterstep, and so forth.

X components

- An X Server (usually Xorg):
 - Controls keyboard, mouse, and one or more screens
 - Controls resolution, refresh rate, and color depth
 - Allows simultaneous access by several clients
 - Performs basic graphic operations
 - Forwards keyboard and mouse events to the correct clients
- An X Client:
 - Is, for instance, an application
 - Receives keyboard and mouse inputs from server
 - Sends output to be displayed to server
- A window manager:
 - Is a special X Client
 - Performs window dressing on other clients
 - Allows other client windows to be moved, iconified, and so forth

© Copyright IBM Corporation 2012

Figure 12-4. X components

LX027.2

Notes:

The central component in any X configuration is the X server. This is a piece of software which handles the low-level complexities of controlling a keyboard, mouse, graphical adapter and monitor. One of the most important configuration choices that need to be made while configuring this server is the resolution, the monitor refresh rate and the color depth. When the X server is running, all keyboard and mouse events are received and forwarded to the appropriate X clients.

X clients are the applications that the user started, or that are started by the system by default. If an application is activated by the user (usually by clicking it), it is sent all relevant keyboard and mouse events, and can react to it. If the client wants to output something, then it sends these requests to the server, which displays them.

A special client is the window manager. This client usually does not have its own window, but displays the borders around other windows, and thus ensures that windows can be resized, moved and iconified. The window manager is usually the first client that is started, and it may start other clients in turn, if configured. Typical applications that are started by the window manager are task bars, launchers, pagers, and so forth.

Desktop environments

- A desktop environment is:
 - A set of tools, libraries and standards that allows rapid development of X clients
 - A set of X clients (including one or more window managers) that are developed with these tools, libraries, and standards
- Examples:
 - GNU Network Object Model Environment (GNOME)
 - K Desktop Environment (KDE)
- Advantages of desktop environments:
 - Integration (cut and paste through clipboard and drag and drop)
 - Common look (themes)

© Copyright IBM Corporation 2012

Figure 12-5. Desktop environments

LX027.2

Notes:

A *desktop environment* basically consists of two things:

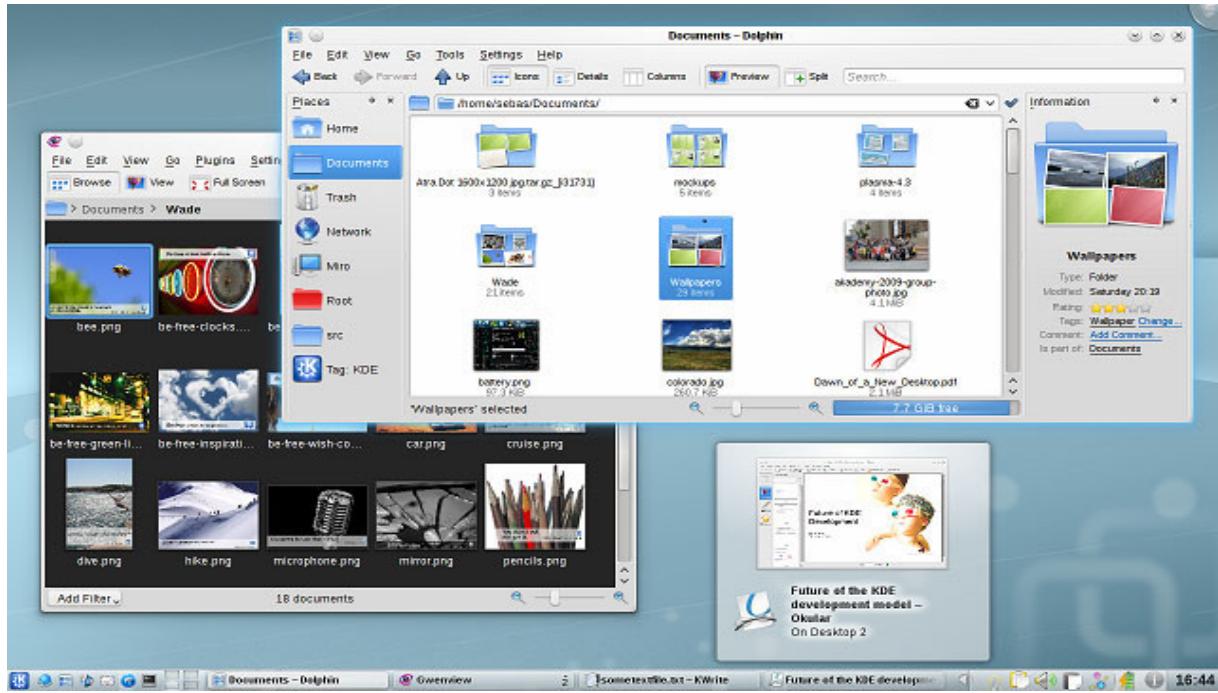
- A set of tools, libraries and standards that allow a programmer to develop X clients.
- A set of X clients (usually including one or more window managers) that were developed using these tools, libraries and standards.

The most popular examples of desktop environments today are GNOME and KDE.

Using a desktop environment instead of a collection of loose X clients has several advantages:

- X clients that are developed as part of a desktop environment tend to have better integration with other clients from that same environment. This makes things like cross-application cut & paste and drag & drop possible.
- These X clients typically have the same look and feel. In most cases, this culminates in the use of *themes*: a combination of colors and textures that look good with each other (in the eyes of the person that developed a theme), and that, when selected once, is used by any client from that desktop environment.

The KDE



© Copyright IBM Corporation 2012

Figure 12-6. The KDE

LX027.2

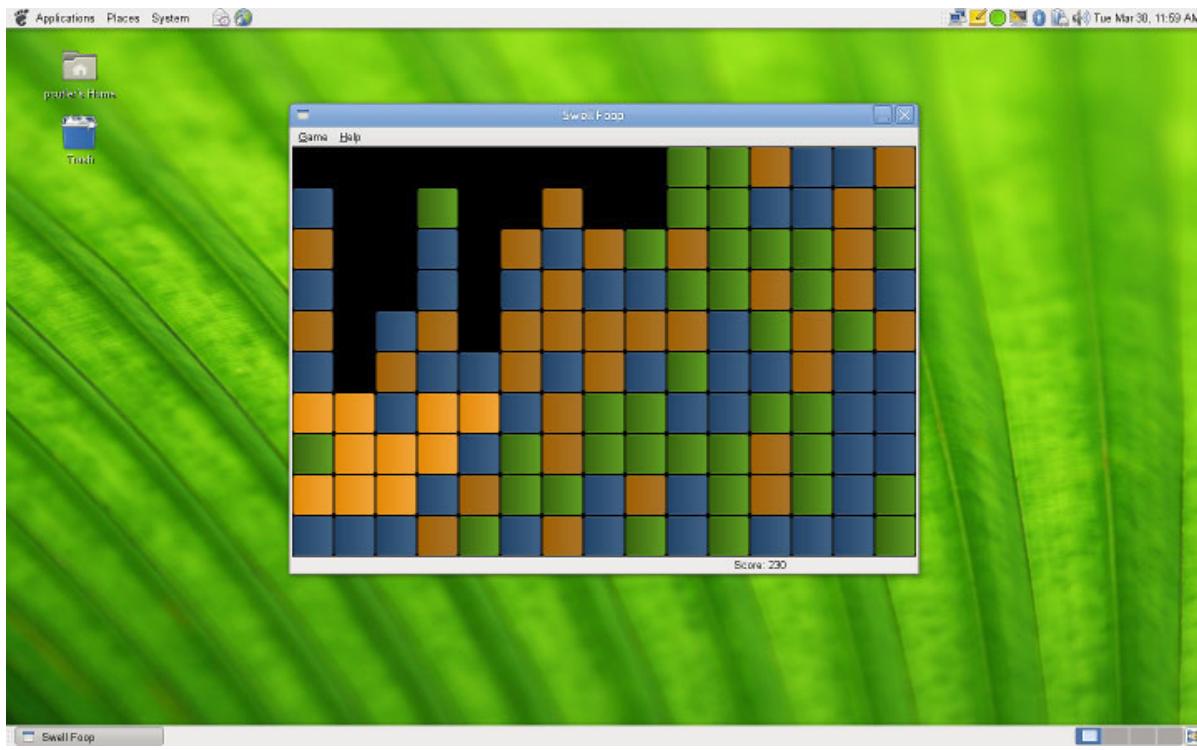
Notes:

The visual shows a screenshot of the K Desktop Environment. Several things can be identified. First, the desktop itself. On this are a number of icons which were created automatically when the system was installed. Clicking these icons launches various programs, varying from the mount command (to mount the CD-ROM) to help browsers. Second, the panel, which stretches all the way along the bottom of the screen. The panel is divided into a number of areas:

- The first area starts with the KDE button, which starts a Start menu just like Windows.
- The second area shows the virtual desktops that are currently configured.
- The third area shows the applications that are currently running in this desktop.
- The fourth area holds a lock and an Exit button.
- The fifth area holds icons for certain KDE applications, such as Klipper, the KDE clipboard.
- The sixth area holds a clock.

The third thing that can be seen is the set of applications that are running. In the visual, only a terminal window, a browser (Konqueror) and the GIMP have been started.

The GNOME desktop environment



© Copyright IBM Corporation 2012

Figure 12-7. The GNOME desktop environment

LX027.2

Notes:

The visual shows a screenshot of a GNOME (GNU Object Modeling Environment) desktop, again taken from a Red Hat system. As you can see, the basic functionality is not all that different.

Starting X

- If logged in on a text terminal, run `startx`.
 - This only starts a single session.
 - When the session ends, you are back in your text terminal.
- If you want to enable the graphical log in screen, bring the system into runlevel 5.
 - To switch manually, use `init 5` command.
 - To make change permanent, edit `/etc/inittab`:
 - `id:5:initdefault:`

© Copyright IBM Corporation 2012

Figure 12-8. Starting X

LX027.2

Notes:

X can be started in two ways. The first way is by running the `startx` command. This command searches for the first free virtual terminal and starts XFree86 on that terminal. It then starts your favorite window manager of your favorite desktop environment. The window manager finally starts all other applications that make up your desktop.

The second way is by switching to runlevel 5. Switching to runlevel 5 can be achieved with the command `init 5`, and can be configured as default runlevel by editing the file `/etc/inittab`. As root, find the line that currently says:

`id:3:initdefault:`

and change it to

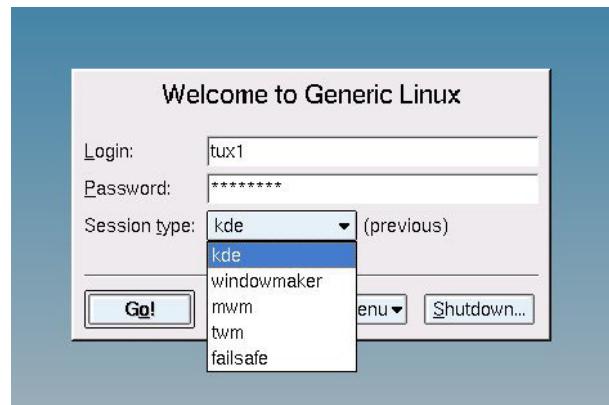
`id:5:initdefault:`

When your system reboots, it starts in runlevel 5 automatically.

In runlevel 5, a so-called *display manager* is started. This program (depending on your settings this is xdm, kdm, or gdm) displays a graphical login prompt. When a user logs on, it subsequently starts the favorite window manager of that user, just as with `startx`.

Choosing your desktop environment

- Most distributions provide multiple desktop environments.
- To choose between them, select from the Login prompt.
- Every user can have his or her own preference.



© Copyright IBM Corporation 2012

Figure 12-9. Choosing your desktop environment

LX027.2

Notes:

Between distributions, there is no default way of selecting your favorite desktop environment. Most distributions store the desktop environment in some hidden file in the home directory of the user. This ensures that each user can have his own favorite environment. But the name of the file is not really standardized, as are the tools that allow you to change the file.

Fortunately, a generic way has been added to the graphical login prompt (gdm or kdm). A pull-down menu allows you to choose between a number of available desktop environments. Furthermore, your choice is stored and is listed as your preferred choice next time you log in.

Unit review

- The GUI of Linux is based on the X Window System (X for short).
- X uses a client-server model.
- The most common X Server under Linux is XFree86 or X.org.
- A desktop environment is a set of tools, libraries, and standards that allow development of X clients and a set of X clients developed with this.
- The most common desktop environments on Linux are KDE and GNOME.
- To switch between desktop environments, use the session type list from the graphical log in prompt.

© Copyright IBM Corporation 2012

Figure 12-10. Unit review

LX027.2

Notes:

Checkpoint

1. True or False: The main configuration file of KDE is /etc/X11/XF86Config.

2. What statement describes the function of the X Server best?
 - a. It receives input from the keyboard and mouse and forwards this to the appropriate client, and it receives output from the clients and displays this on the screen.
 - b. It performs the window dressing; it makes sure that every application has a border around its windows so that the window can be resized, moved, and iconified.
 - c. It allows the user to type commands while in a graphical environment.
 - d. It shows a set of eyes looking at the cursor.

3. How do you start X?

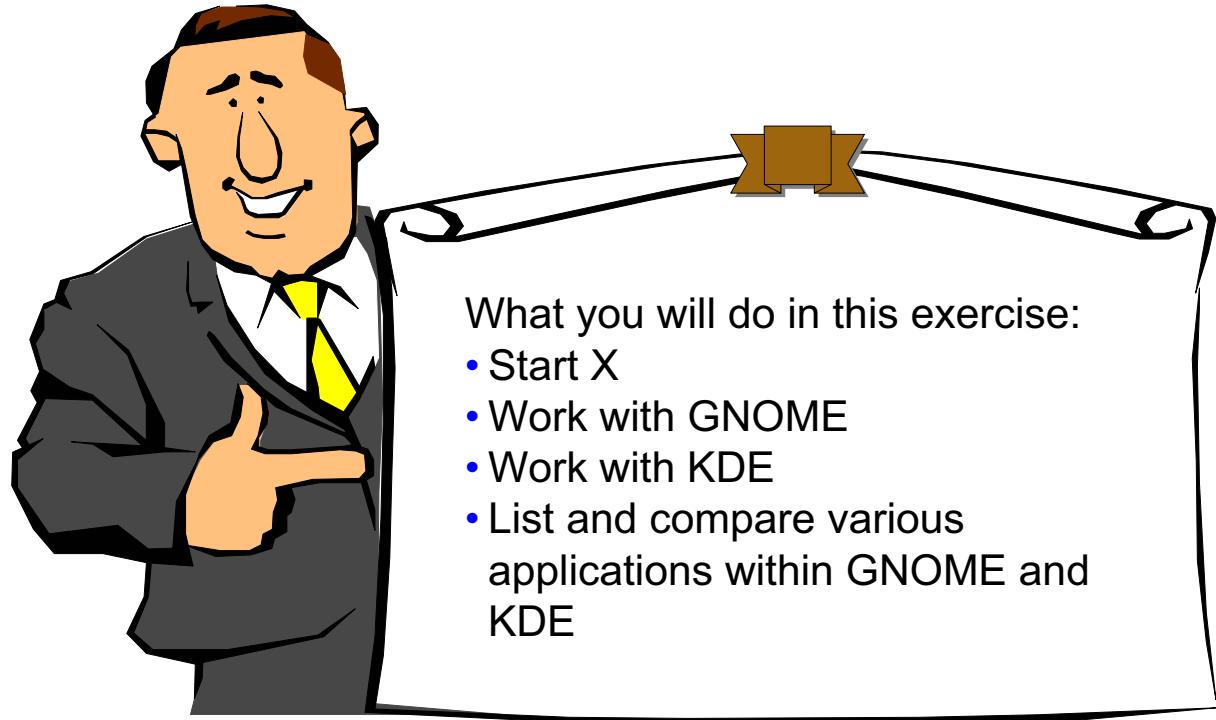
© Copyright IBM Corporation 2012

Figure 12-11. Checkpoint

LX027.2

Notes:

Exercise: The Linux GUI



What you will do in this exercise:

- Start X
- Work with GNOME
- Work with KDE
- List and compare various applications within GNOME and KDE

© Copyright IBM Corporation 2012

Figure 12-12. Exercise: The Linux GUI

LX027.2

Notes:

Unit summary

Having completed this unit, you should be able to:

- List the main components of the X Window System
- List the function of the X Server
- List the function of a window manager
- List the main characteristics of desktop environments
- Switch between GNOME and KDE

© Copyright IBM Corporation 2012

Figure 12-13. Unit summary

LX027.2

Notes:

Unit 13. Customizing the user environment

What this unit is about

This unit describes the order of login scripts, explains how to modify login scripts to customize the bash environment, and covers the tools available for customized the GUI.

What you should be able to do

After completing this unit, you should be able to:

- List the order of login scripts
- Modify login scripts to customize the bash environment
- List the tools available for customizing the GUI

How you will check your progress

- Checkpoint questions
- Machine exercises

Unit objectives

After completing this unit, you should be able to:

- List the order of login scripts
- Modify login scripts to customize the bash environment
- List the tools available for customizing the GUI

© Copyright IBM Corporation 2012

Figure 13-1. Unit objectives

LX027.2

Notes:

Bash initialization: Login shell

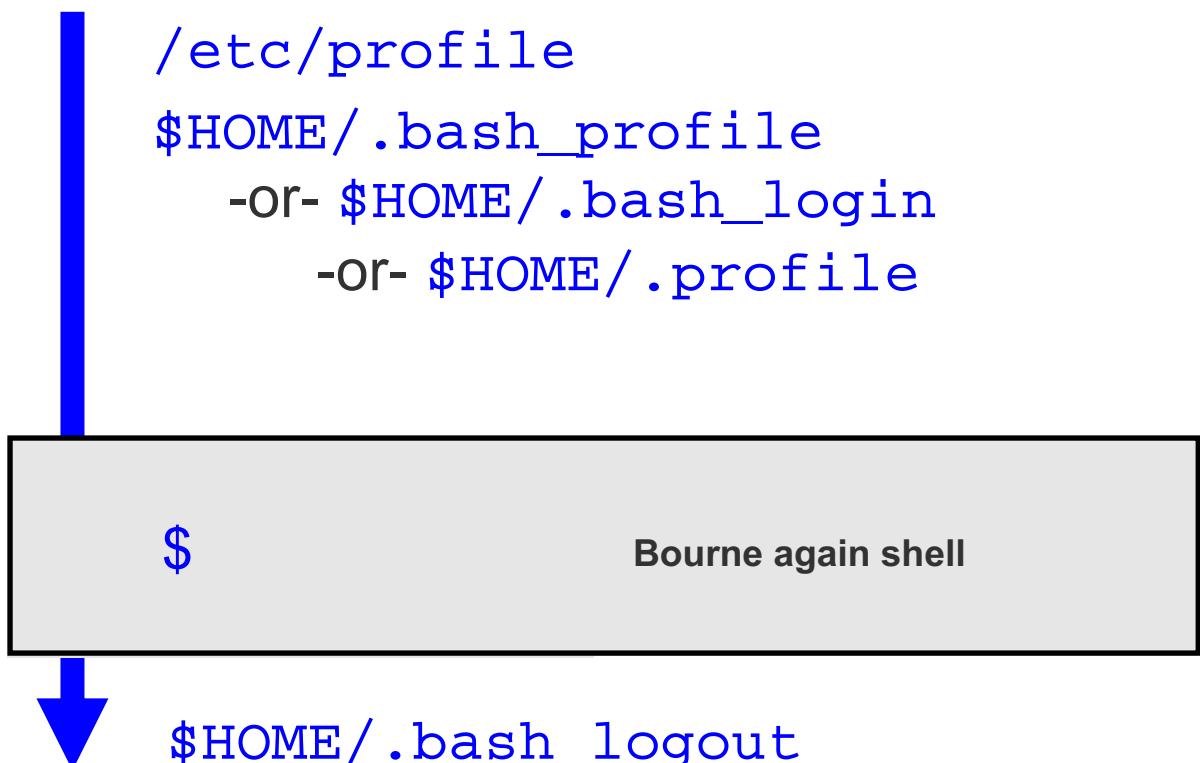


Figure 13-2. Bash initialization: Login shell

LX027.2

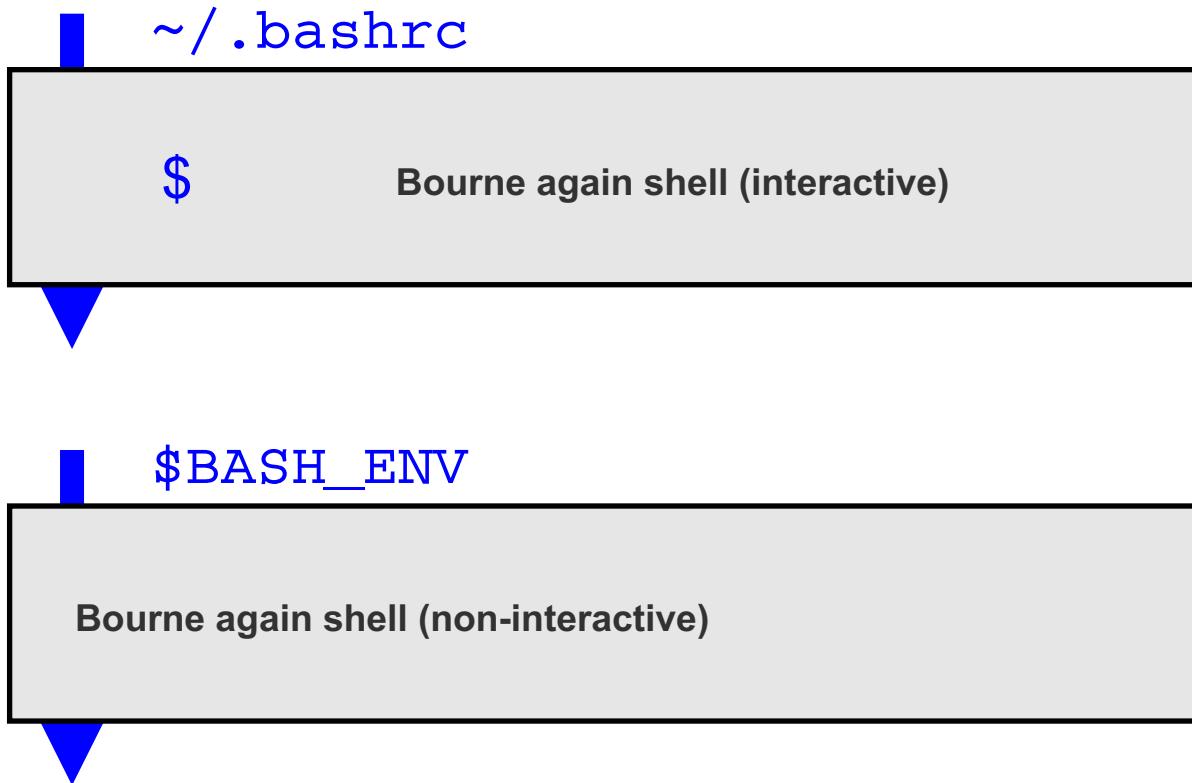
Notes:

The first file that shell uses at login is `/etc/profile`. This file contains variables specifying the basic environment for all processes and can be changed only by the system administrator. Furthermore, this file runs commands in your environment when you log in.

Next, the shell executes `$HOME/.bash_profile`. This file serves the same purpose as `/etc/profile` but this file can be changed by the user. If it is not found, `$HOME/.bash_login` is used, and if that file is not found, `$HOME/.profile` is used.

Ensure that newly created variables do not inadvertently conflict with standard variables such as `MAIL`, `PS1`, `PS2`, and so forth.

Bash initialization: Non-login shell



© Copyright IBM Corporation 2012

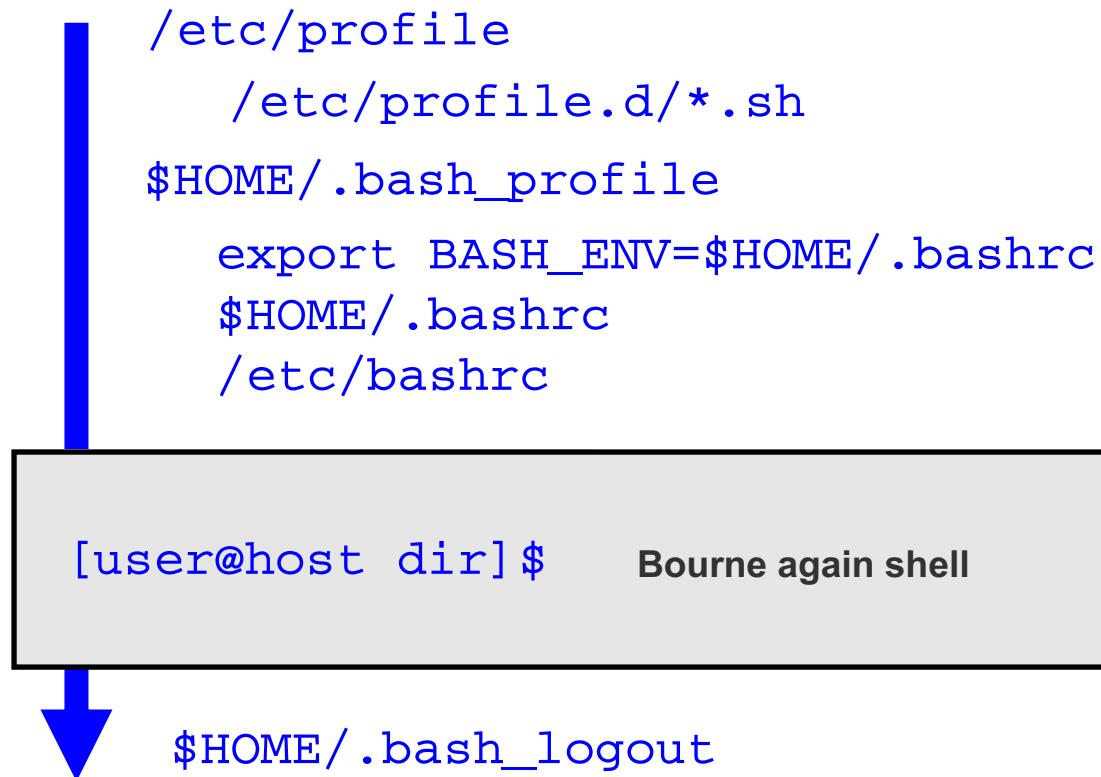
Figure 13-3. Bash initialization: Non-login shell

LX027.2

Notes:

When a shell is started, but not as a login shell, `/etc/profile` and `~/.bash_profile` are not read. Instead, the bash shell either uses `$HOME/.bashrc` (for a non-login, interactive shell, such as the shell that runs in a terminal window) or the `$BASH_ENV` script (for a non-interactive shell, such as a shell executing a shell script).

Bash initialization with Red Hat extensions



© Copyright IBM Corporation 2012

Figure 13-4. Bash initialization with Red Hat extensions

LX027.2

Notes:

On a Red Hat system, a number of extra files are called when a user starts bash.

- First, `/etc/profile` also calls every shell script in `/etc/profile.d`. To avoid confusing this with the csh shell, only scripts with the extension `.sh` are called.
- In `$HOME/.bash_profile`, `$BASH_ENV` is set to `$HOME/.bashrc`. This ensures that a non-interactive, non-login shell initializes itself with the `$HOME/.bashrc` file too.
- Then, from `$HOME/.bash_profile`, `$HOME/.bashrc` is called. This ensures that even a login shell initializes itself with the `$HOME/.bashrc` file.
- `$HOME/.bashrc` in turn calls `/etc/bashrc`. This ensures that global options can be defined by the system administrator, even for non-login and non-interactive shells.

All files mentioned are customizable. As an example, the system administrator could add scripts that send an alert (log file entry or mail) when a user logs in or out. If you modify the shell scripts mentioned, there are two things to be aware of:

- Make sure you do not inadvertently change standard shell variables, such as `$HOME`, `$MAIL`, and so forth.
- An upgrade of a system might overwrite these standard shell scripts.

Bash initialization with SLES extensions

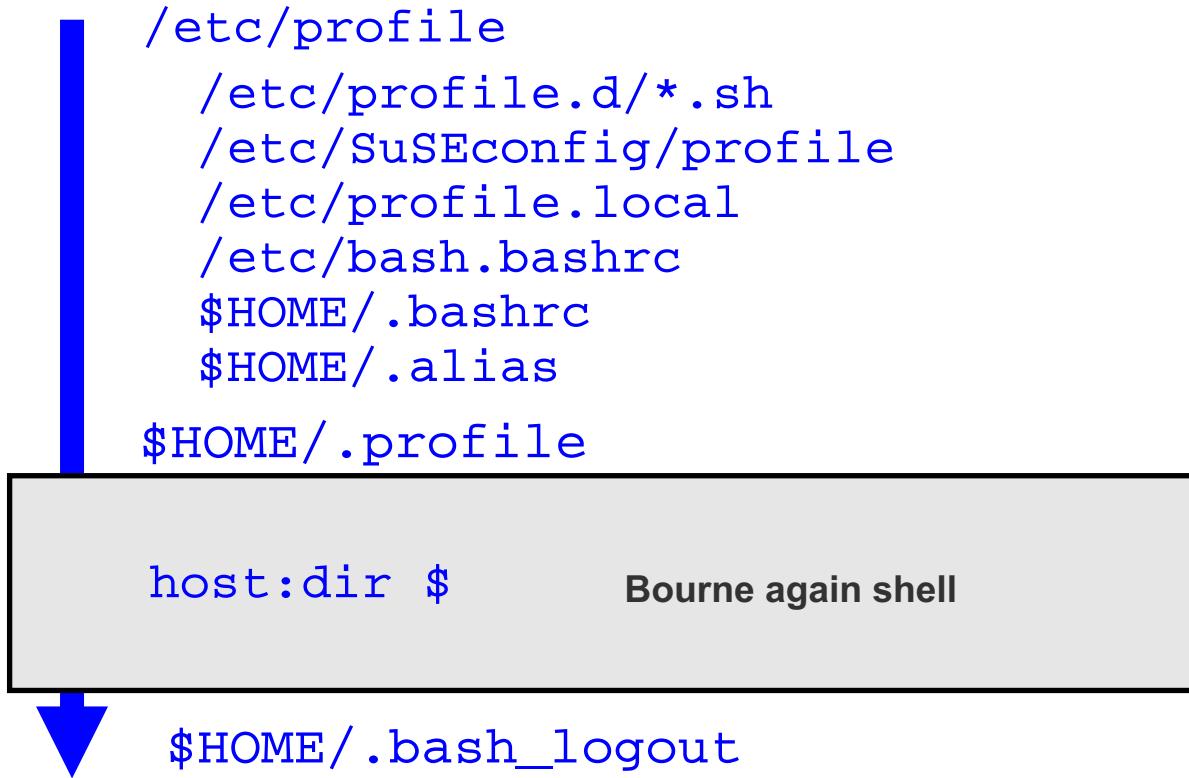


Figure 13-5. Bash initialization with SLES extensions

LX027.2

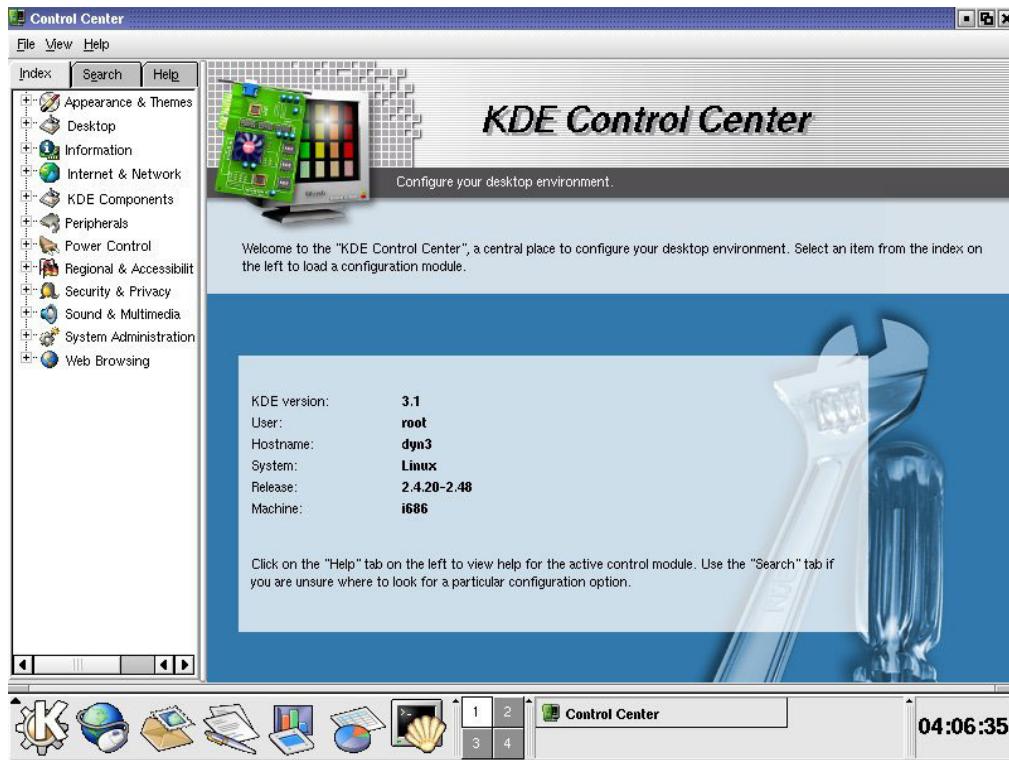
Notes:

SLES has also extended the bash startup flow, but differently. Here, the `/etc/profile` script has a far more important function. It calls, in turn:

- All shell scripts in `/etc/profile.d`. Only scripts with a `.sh` extension are called.
- `/etc/SuSEconfig/profile`. This script is modified by YaST when the system configuration changes, and should not be changed by hand.
- `/etc/profile.local`. This script is for local customizations.
- `/etc/bash.bashrc`. This script contains system-wide bash customizations that are also applied to non-login shells.
- `/etc/bash.bashrc` also calls `/etc/bash.bashrc.local`, which can be used for local customizations. This is not shown in the visual.
- `$HOME/.bashrc`. This script can obviously be modified by the user.
- `$HOME/.alias`. This script can also be modified by the user, and is used to define user-specific aliases.

Obviously, after having called `/etc/profile`, bash also calls `$HOME/.profile`. This file is virtually empty on a default SLES system, but can be modified by the user.

KDE customization



© Copyright IBM Corporation 2012

Figure 13-6. KDE customization

LX027.2

Notes:

Although the KDE configuration consists of a large number of text files as well, you do not have to know or edit them by hand: KDE comes with a control center, which allows you to modify your environment easily.

GNOME customization

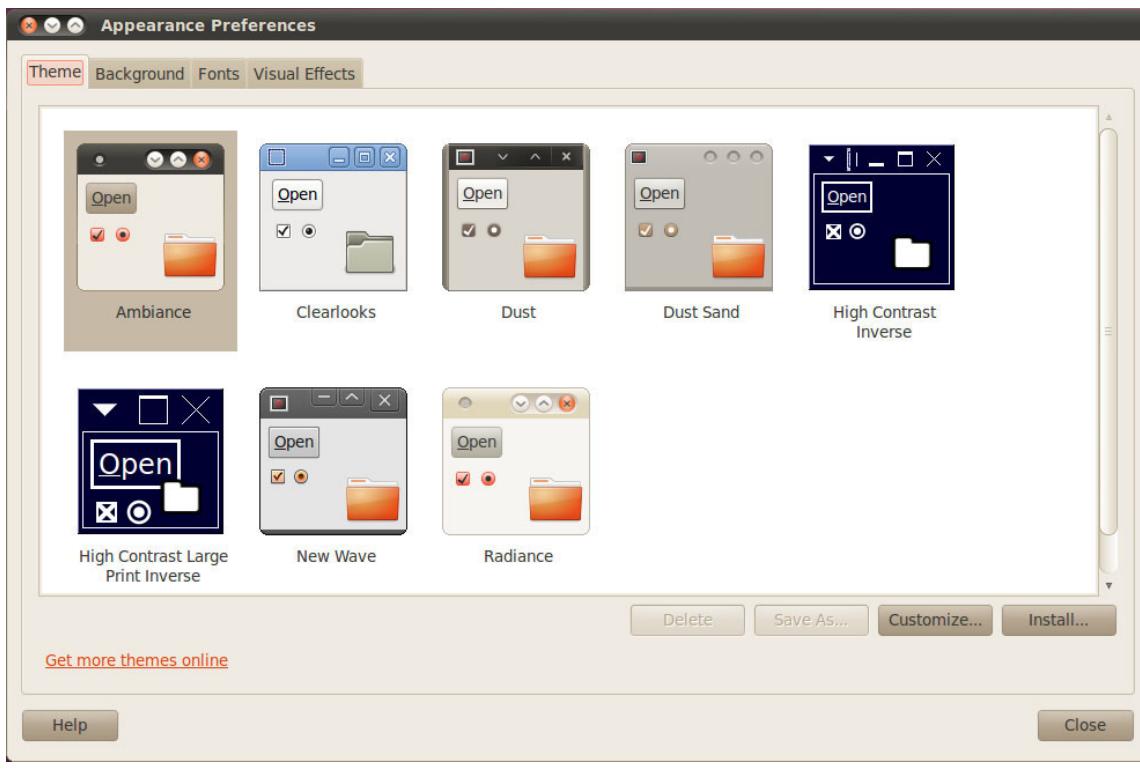


Figure 13-7. GNOME customization

LX027.2

Notes:

GNOME also comes with its own graphical configuration screens. As with KDE, your changes are actually stored in text files which can also be modified by hand, using a text editor, such as Vi, but this is hardly ever necessary.

Unit review

- A number of shell scripts allow you to customize your shell environment.
- Both KDE and GNOME have an integrated control center for customization.

© Copyright IBM Corporation 2012

Figure 13-8. Unit review

LX027.2

Notes:

Checkpoint

1. Which file would you use to customize your environment variables, and why this file?

2. What do the following variables define on your system:
PATH, PS1

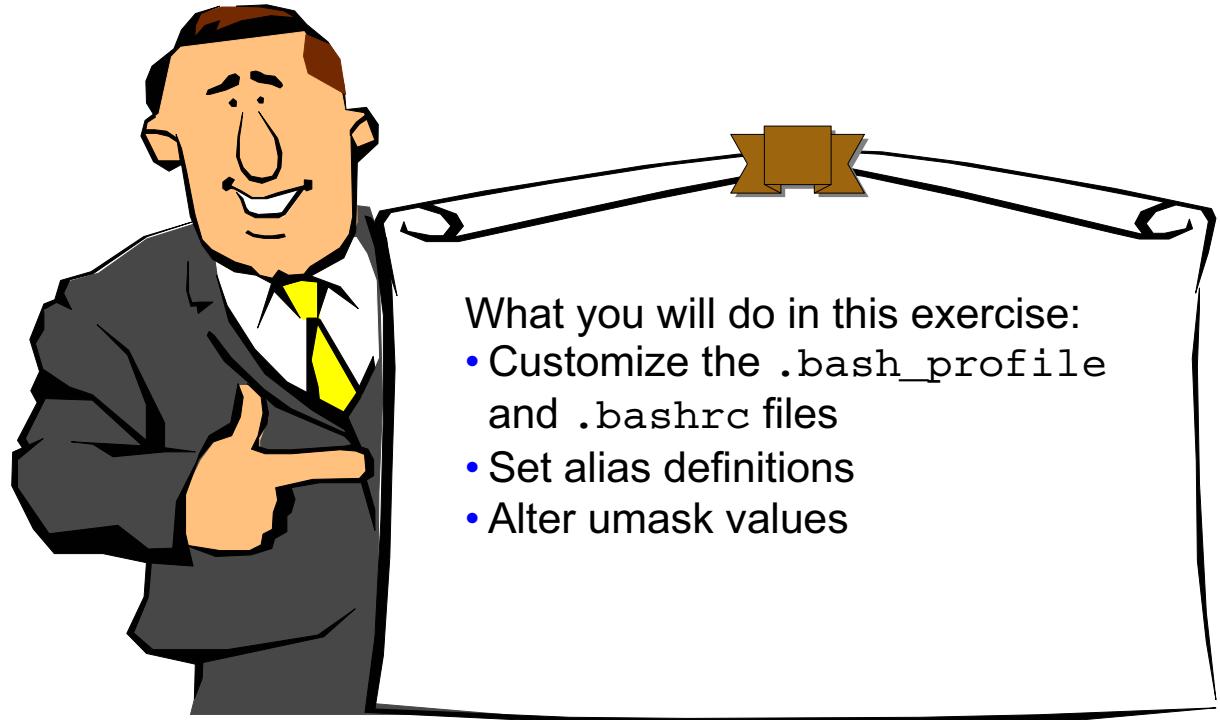
© Copyright IBM Corporation 2012

Figure 13-9. Checkpoint

LX027.2

Notes:

Exercise: Customizing the user environment



What you will do in this exercise:

- Customize the .bash_profile and .bashrc files
- Set alias definitions
- Alter umask values

© Copyright IBM Corporation 2012

Figure 13-10. Exercise: Customizing the user environment

LX027.2

Notes:

Unit summary

Having completed this unit, you should be able to:

- List the order of login scripts
- Modify login scripts to customize the bash environment
- List the tools available for customizing the GUI

© Copyright IBM Corporation 2012

Figure 13-11. Unit summary

LX027.2

Notes:

Unit 14. Basic system configuration

What this unit is about

This unit discusses system management tools, installing and deinstalling additional software, and configuring a printer, sound card, and network adapter.

What you should be able to do

After completing this unit, you should be able to:

- Discuss system management tools
- Install and deinstall additional software
- Configure a printer
- Configure a sound card
- Configure a network adapter

How you will check your progress

- Checkpoint questions
- Machine exercises

Unit objectives

After completing this unit, you should be able to:

- Discuss system management tools
- Install and deinstall additional software
- Configure a printer
- Configure a sound card
- Configure a network adapter

© Copyright IBM Corporation 2012

Figure 14-1. Unit objectives

LX027.2

Notes:

Why system configuration?

- Most system configuration is done during installation.
- You might need to change system configuration afterwards.
 - Things not configured during installation
 - Configuration failed during installation
 - Environment changed after installation
- There are three ways to change system configuration.
 - Temporary: Until next system reboot
 - Manually: Changing config files by hand
 - Automated: Using system administration tools
- The following are typical items that need to be configured on a workstation:
 - Add or remove software
 - Printers
 - Sound cards
 - Network

© Copyright IBM Corporation 2012

Figure 14-2. Why system configuration?

LX027.2

Notes:

In most distributions, a workstation install takes care of configuring most of your system. In fact, in all but a few cases you can be productive right away when your system has been installed.

There are, however, a few cases in which you need to do additional configuration to your Linux workstation:

- Certain things were not be configured during the installation. This might be because the distribution manufacturer left that component out altogether, or that you decided to skip that part of the installation process.
- The attempted configuration of a certain thing failed. This typically happens to sound cards: Trying to detect certain older types of sound cards may cause your system to hang.
- After installation, the environment in which your workstation has to operate changes. You might be getting a new printer, or be relocated to another network, for instance.

All these factors might require you to do system configuration by hand. There is a number of ways this can be done on a Linux system, and the following three terms are widely used in this respect.

- Temporary system administration means changing a parameter without making the change permanent. So the next time your system reboots, the old configuration which was stored on disk is used again. This sort of administration is for instance used when you need to connect your laptop to another network than the network you usually connect to.

Temporary system administration usually comes down to entering a single command which overrides the parameters that are stored on disk.

- Manual system administration generally refers to making the configuration change on disk yourself. As we've already seen, virtually all configuration options of a Linux system are stored in text files somehow. Editing these files by hand, using a text editor, is called manual administration.

Manual administration is typically done only by experienced system administrators who know the internal layout of all these configuration files, and who understand the interactions between the various components of a Linux system.

Manual configuration typically requires you to restart the appropriate service afterwards manually too.

- Automated configuration means that you use some sort of system administration tool with a menu-driven interface that makes the desired change for you. The advantage of this method is that you don't need to know the internal layout of the configuration files, that the chance of making errors is smaller, and that the restart of the appropriate service is taken care of too. This makes automated configuration the ideal method for a beginning user.

There are disadvantages too to automated configuration. One is that an experienced system administrator can usually make the desired change faster by doing a manual configuration change than by using a system administration tool. And the other, bigger disadvantage is that the changes you can make are limited by the capabilities of the tool. Typical system administration tools are not written by the programmers of the service to be configured, but by someone else. And that someone else typically does not support all configuration options that the programmer has built into the service.

A Linux system is highly configurable and supports a lot of services and hardware. In this unit, we're only going to look at four of the most common system administration tasks that you encounter on a typical workstation:

- Adding and removing software
- Configuring printers
- Configuring sound cards
- Configuring network adapters

System configuration tools

- Various tools have been developed to ease system administration.
 - Application specific (Samba SWAT and so on)
 - Distribution specific (RH system-config-*, SLES YaST, and so on)
 - Desktop environment specific (gmenu, kcontrol, and so on)
 - Generic Linux/UNIX (webmin and so on)
- The perfect tool does not exist (yet).



© Copyright IBM Corporation 2012

Figure 14-3. System configuration tools

LX027.2

Notes:

One of the main disadvantages of Linux as compared to commercial UNIXes as AIX, for instance, is that there is no single, large manufacturer behind it who can force all developers to work according to a single standard. This is particularly visible when it comes to system configuration tools. Where for instance AIX comes with one tool, SMIT, with which you can manage the entire system, there is no Linux distribution who can do that.

Nevertheless, there is a need for system administration tools. Various people have worked on developing these, but all these tools had a specific, limited purpose.

There are, for instance, tools that are developed by the authors of an application, to allow management of that particular application. Other tools are developed by distribution manufacturers to allow basic configuration of that particular distribution. There are tools that attempt to be generic Linux configuration tools, and there are even some tools that attempt to be generic UNIX configuration tools.

The perfect tool, however, does not exist. Now what *perfect* is, is in the eyes of the beholder, but all of the tools that present themselves as generic still suffer one major deficiency (as of now): They currently do not motivate the author of a program to also write

the configuration menus for that program, for that particular system configuration tool. It is virtually always the author (or a team of people working for the author) of the system configuration tool who writes the configuration menus for a particular application. This means that the system configuration tool always lags behind, increases the chance of errors, and limits the features that are supported by the configuration tool.

A “perfect” administration tool should preferably work like the **man** command: It should have a published interface (file format or whatever) that everybody can produce, and be available on every Linux (or UNIX) distribution. Only then would this motivate and allow the authors of an application to also write the configuration menus for that application, just like they are currently already doing for manual pages.

That perfect tool is not yet available. Instead, every distribution comes with its own distribution specific tools, and some applications have their own tools as well. This means that you have to figure out from the information about your distribution which tools are available for you, and you also need to figure out which tool you prefer, if multiple tools perform the same function. This unit attempts to be a guide in that, but it can never be complete, unfortunately.

Adding or removing software using RPM

- Use RPM to install or upgrade software packages.
- Common options include:
 - **-i**: Installing new packages
 - **-U**: Upgrading existing packages
 - **-e**: Removing packages

```
$ rpm -ihv myprog-1.2-34.i386.rpm
myprog #####.....
$ rpm -Uhv myprog-1.2-78.i386.rpm
myprog #####.....
$ rpm -e myprog
```

- The **-h** option shows a progress bar. The **-v** option is verbose output.

© Copyright IBM Corporation 2012

Figure 14-4. Adding or removing software using RPM

LX027.2

Notes:

Most distributions in the market today distribute their software in RPM format¹. RPM, which stands for RPM Package Manager², is a file format that is extremely suitable for software distribution, because it combines the following things in one file:

- Name of the software
- Version numbers
- Copyright License
- Authors
- Build information
- Dependency information
- The files that make up the software package, grouped into programs, configuration files and other files.

¹ In fact, the Linux Standards Base (LSB), which aims at developing a set of standards that every distribution has to adhere to, has specified RPM to be the package distribution format.

² RPM used to stand for Red Hat Package Manager. But to encourage the use of RPM by other distributions, Red Hat has released it under the GPL and has renamed it to RPM Package Manager. And yes, that's a self-referencing acronym, just like GNU.

- Pre- and post-installation scripts
- Pre- and post-deinstallation scripts
- Cryptographic checksums (so the integrity of the individual files and the whole package can be verified.)

Furthermore, all these components are automatically compressed.

When an RPM package (a file which generally ends with `.rpm`) is installed, the information about it goes into a database on disk (usually `/var/lib/rpm`). This makes it possible to retrieve all information about all installed packages without having to read the original RPM files.

Virtually everything you want to do with an RPM package or with the RPM database is done with the **rpm** command. It is really powerful, but we only cover the most common things:

Installing additional software is done with the `rpm -i` command, followed by the file names of the packages to install.

Updating software is done with the `rpm -U` command, followed by the package file name.

Removing software is done with the `rpm -e` command, followed by the package name.

Querying the RPM database

- Options
 - **-i:** List information
 - **-l:** List all files
 - **-p:** Queries new packages before installing

```
# rpm -qi myprog
Name           : myprog          Relocations: (not relocatable)
Version        : 1.0.1           Vendor: IBM Inc.
...
# rpm -ql myprog
/usr/bin/myprog
/etc/myprogrc
/usr/share/man/man1/myprog.1.gz

# rpm -qlp yourprog-1.2-34.i386.rpm
/usr/bin/foo
/etc/foorc
/usr/sha    re/man/man1/foo.1.gz
```

© Copyright IBM Corporation 2012

Figure 14-5. Querying the RPM database

LX027.2

Notes:

Querying already installed packages is done with the `rpm -q` command, followed by the package name. If no further options are given, only the package name shows up. Adding the `-i` option gives you all information about a package, while adding the `-l` option gives you all the files that are part of this package. To query all packages, use the `-a` option and do not give a package name.

Querying not yet installed packages is done the same as querying already installed packages, with one exception: you need to specify the `-p` option and the package file name instead of the package name.

Note that there is a difference between the package file name and the package name. A package file name typically consists of the package name, the version number and the `.rpm` extension like this: `<packagename>-<version>.rpm`. Also note that some distributions (for example, SUSE) do not include the version number in the package file name.

Adding or removing software from a .tar.gz file

- .tar.gz (.tgz) is default distribution format for source code.
 - tar = tape archiver: Stores a directory tree in a single file
 - gz = GNU Zip: Compression program
- To unpack a .tar.gz or .tgz archive:

```
cd /usr/local/src
tar -zxvf archive-version.tar.gz
cd <archivename>
```
- Read INSTALL or README file for installation instructions.
- It should be installed under /usr/local.

© Copyright IBM Corporation 2012

Figure 14-6. Adding or removing software from a .tar.gz file

LX027.2

Notes:

The default distribution format for source code in the Linux community (and to a large extent, the UNIX community as well) is not .rpm, but .tar.gz. This extension means that:

- All files have been combined into one large archive file using the **tar** command.
- The resulting single file has been compressed using the **gzip** command.

If you want to make use of these files, you need to uncompress and unpack them first. Fortunately, GNU tar (the **tar** command that is available on most Linux distributions) can do all of this in one run. The default location to perform this operation is /usr/local/src. The archive itself usually creates its own subdirectory (/usr/local/src/<archive-version>) and stores all its files in there.

The next step is then to **cd** into the directory and view the documentation that the author wrote to figure out the next steps. This is usually stored in a file called README or INSTALL. If you only received the source code, you need to compile the software too, which usually involves running a configuration program (**./configure** or **make config**), a compilation program (typically **make**) and an installation program (**make install** or **./install**).

Other Linux software installers

- Other methods of installing or updating software on Linux distributions include the following:
 - **yum** (Yellowdog Updater, Modified) is a Red Hat update tool. A RH support license is required. **yum** will also download any other necessary packages from the software repository if possible.
 - **you** (YaST Online Update) is a SUSE update tool with similar functionality as **yum**.

© Copyright IBM Corporation 2012

Figure 14-7. Other Linux software installers

LX027.2

Notes:

On top of **rpm**, both Red Hat and SUSE have created additional programs. The main functionality of these is that they are able to communicate with internet-based repositories. This allows you to update to the latest level automatically, and allows you to install any prerequisite software automatically.

yum (Yellowdog Updater, Modified) is the Red Hat tool of choice. It communicates with the Red Hat Network (RHN), which is (amongst other things) the repository for Red Hat software. To make use of this repository you need an RHN license.

you (YaST Online Update) is the YaST module that takes care of keeping the system up to date, and installing new software from the SUSE repository.

Printer configuration

- On RHEL and SLES, the printer subsystem is Common UNIX Printing System (CUPS).
 - Configuration is done through **Ipadmin** or with a browser (<http://hostname:631/>) (*recommended*)

© Copyright IBM Corporation 2012

Figure 14-8. Printer configuration

LX027.2

Notes:

Printer configuration is, technically speaking, one of the most complex tasks of system administration. The reason lies not in the configuration files itself, but in the fact that there are so many file formats that your printer subsystem may need to handle, while your printer typically only supports one or two. This requires the setup of elaborate filters, which detect the file format of the file to be printed, and convert it into a format suitable for your printer.

Furthermore, printers themselves have evolved a lot in recent history: from daisy wheel printers that supported only a single character set in a single font, to high-volume, double sided color laser printers that support various color models.

These developments have led to a number of printer subsystems, each progressively more complex and powerful. The printer subsystem currently in use on Red Hat and SUSE is CUPS (Common UNIX Printing System). Other distributions might use other printing subsystems, such as BSD, LPRng, or PPS.

CUPS is highly configurable. As with most subsystems, it can be configured by editing a series of configuration files, but this is not recommended. Instead, you should use **Ipadmin** to configure CUPS, or point your browser to <http://<hostname>:631>. This gives you an

HTTP-interface to the cupsd daemon and is by far the easiest method of configuring CUPS.

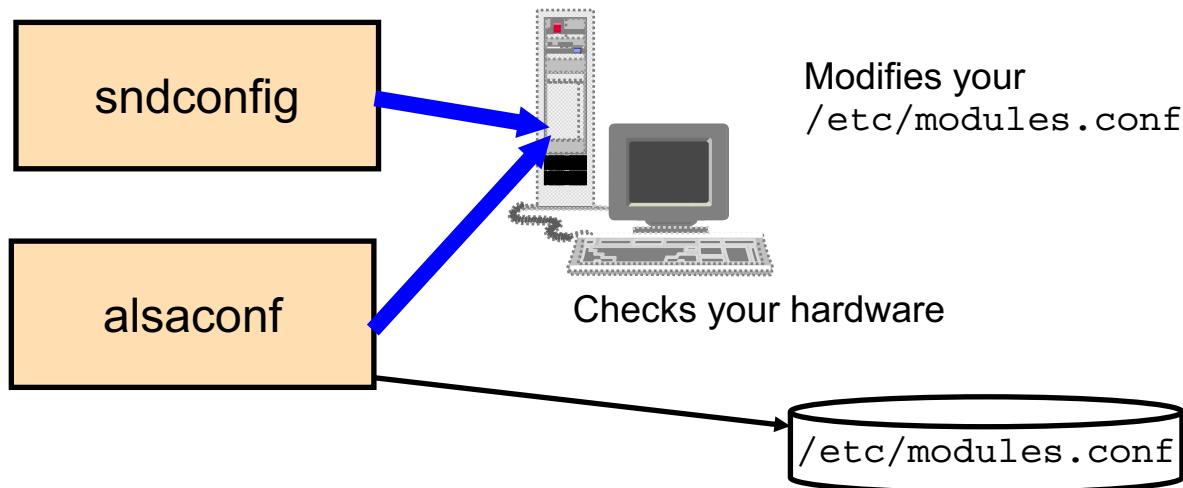
When configured, you can submit print jobs to CUPS by using the **lp** command (derived from and compatible with AT&T's System V printing subsystem) or the **lpr** command (derived from and compatible with BSD's printing subsystem). Graphical applications are typically compatible with **kdeprint**, which means that they call **kdeprint** and interface with it to get the job printed. This gives you a seamless print interface from that application.

Other useful commands are:

- **lpstat** and **lpq**: These commands show you the jobs queued to be printed.
- **lpcancel** and **lprm**: These commands allow you to cancel a print job.

Sound card configuration

- Configuration is usually done with a dedicated tool.
 - RHEL: system-config-soundcard
 - SLES: YaST2 or Alsacnf



Sound card support requires correct loading of kernel modules!

© Copyright IBM Corporation 2012

Figure 14-9. Sound card configuration

LX027.2

Notes:

Sound card support is performed by the Linux kernel itself, in the form of kernel modules³. The file that holds the information about the modules to be loaded to support various hardware components is `/etc/modules.conf`⁴. This file itself is not that hard to configure (it is usually only a few lines after all), but it is hard to obtain the correct parameters (a typical sound card can be configured in a dozen ways, depending on whether MIDE needs to be supported, or wavetables, and so forth). Because of this, a dedicated tool is usually used.

On a Red Hat system, use **system-config-soundcard**. On a SUSE system, use **yast2**.

When your sound card has been configured, many multimedia programs can use your sound card. The visual lists a few of these.

³ Pieces of code that can be loaded into the kernel while the system is running to provide support for one specific hardware component.

⁴ This file used to be called `/etc/conf.modules`.

Network configuration

- Need correct network module to be loaded into kernel
 - /etc/modules.conf or /etc/modprobe.conf
- Need to set correct IP addresses and so forth.
 - Generally done with **ifconfig** command
 - For DHCP, **dhpcd**, **pump**, or **dhclient**
- Configuration done through scripts, which are different in each distribution
 - RHEL: /etc/sysconfig/network-scripts/ifcfg-eth0
 - SLES: /etc/sysconfig/network/ifcfg-eth0
- Use distribution-specific tool to configure
 - RHEL: system-config-network
 - SLES: YaST
- For desktop use, **NetworkManager** is an alternative

© Copyright IBM Corporation 2012

Figure 14-10. Network configuration

LX027.2

Notes:

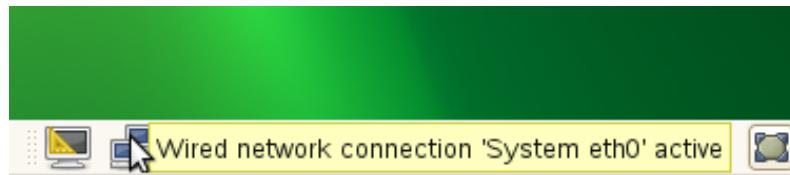
To configure a network, two things need to be done:

- The correct module needs to be loaded into the kernel. This is done by configuring the /etc/modules.conf file correctly.
- The IP address needs to be configured. This is done with the **ifconfig** command. This command does not read a configuration file, but rather expects the IP addresses to be listed as parameters on the command line.

Because **ifconfig** does not read a standard configuration file, every distribution has to come up with its own way of storing the IP addresses and other parameters that need to be configured. Red Hat for instance stores them in /etc/sysconfig/network-scripts/ifcfg-eth0, while SUSE stores them in /etc/sysconfig/network/ifcfg-eth0. In both cases, these files are read by the startup scripts, who in turn execute the corresponding **ifconfig** command. If you are using DHCP, then you don't need to configure all these parameters locally. Instead, you need to start a DHCP client which requests all parameters from a DHCP server on the network. There are several DHCP clients for Linux. The one most often used today is **dhclient**.

NetworkManager

- NetworkManager: Framework for automatic configuration of network parameters
- DHCP client
- Wireless networks: SSID detection, security parameters
- Support for various VPNs (Cisco, OpenVPN, PPTP)
- Components:
 - System Daemon
 - User Interface Applet



© Copyright IBM Corporation 2012

Figure 14-11. NetworkManager

LX027.2

Notes:

NetworkManager is a relatively new tool that has not made it into every distribution yet. It is developed as a replacement for the previously mentioned shell scripts and configuration files.

It consists of two components: a system daemon (NetworkManager) and a user interface applet (nm-applet).

At startup, NetworkManager automatically detects all available network adapters, and will try to configure any network adapter for which it has sufficient information. Usually, this means that it will only configure wired ethernet adapters (through the use of DHCP).

Once a user logs in to the graphical interface, the NetworkManager applet (nm-applet) will be started. This enables an icon in the system tray, through which the user can interact with the NetworkManager daemon. The nm-applet also contacts the local keychain program for any stored WEP/WPA keys for any detected SSIDs. With these security parameters, NetworkManager is able to connect to a secured Wireless Access Point, and will again use DHCP to configure the IP parameters of the adapter.

Lastly, NetworkManager has support for three different kinds of Virtual Private Networks:

- Cisco
- PPTP
- OpenVPN (IPSec)

The configuration for these VPNs is again stored in the local keychain so that the information is secure from other users. Through the NetworkManager applet you can configure, start and stop these VPNs.

Unit review

- System configuration is necessary if the installation program did or could not configure your system or if your environment changed after installation.
- System administration can be temporary, manual, or automatic.
- System administration is made easy by system administration tools.
- The perfect system administration tool does not yet exist.
- You must find out which tools are available on your distribution and which tool works for you.
- Common things to do on a workstation are adding and removing software, configuring printers, configuring sound cards, and configuring network interfaces.

© Copyright IBM Corporation 2012

Figure 14-12. Unit review

LX027.2

Notes:

Checkpoint

1. True or False: When you configure your system as a DHCP client, you do not need to configure IP addresses, and so forth, yourself.

2. The correct command to install an additional RPM would be which of the following:
 - a. rpm -U xpuzzles.rpm
 - b. rpm -e xpuzzles-5.5.2-4.i386.rpm
 - c. rpm -qip xpuzzles.rpm
 - d. rpm -i xpuzzles-5.5.2-4.i386.rpm

3. What is the proper series of commands to install a .tar.gz file?

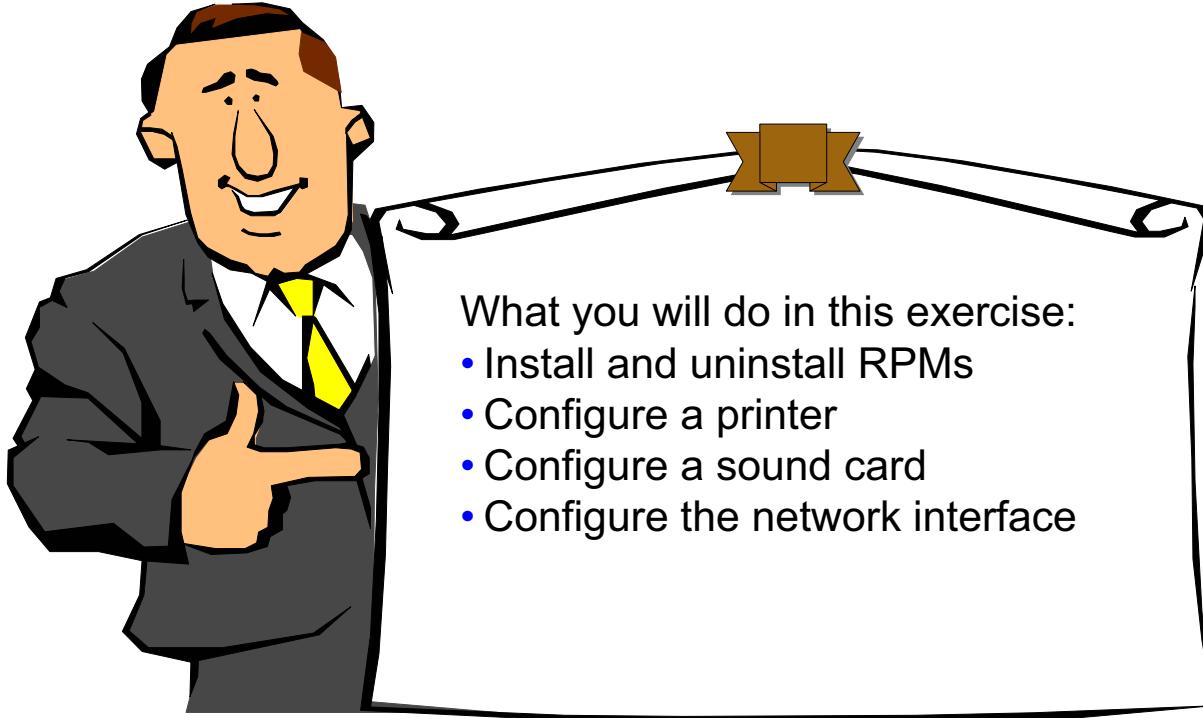
© Copyright IBM Corporation 2012

Figure 14-13. Checkpoint

LX027.2

Notes:

Exercise: Basic system configuration



© Copyright IBM Corporation 2012

Figure 14-14. Exercise: Basic system configuration

LX027.2

Notes:

Unit summary

Having completed this unit, you should be able to:

- Discuss system management tools
- Install and deinstall additional software
- Configure a printer
- Configure a sound card
- Configure a network adapter

© Copyright IBM Corporation 2012

Figure 14-15. Unit summary

LX027.2

Notes:

Unit 15. Integrating Linux in a Windows environment

What this unit is about

This unit shows you how to access Windows-based file systems, run Windows programs, access Windows servers, and read Windows document formats.

What you should be able to do

After completing this unit, you should be able to:

- Access Windows-based file systems
- Run Windows programs
- Access Windows servers
- Read Windows document formats

How you will check your progress

- Checkpoint questions
- Machine exercises

Unit objectives

After completing this unit, you should be able to:

- Access Windows-based file systems
- Run Windows programs
- Access Windows servers
- Read Windows document formats

© Copyright IBM Corporation 2012

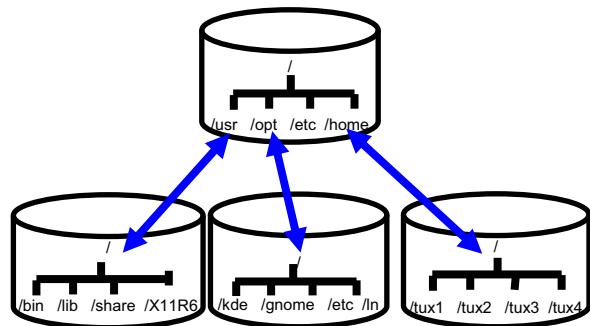
Figure 15-1. Unit objectives

LX027.2

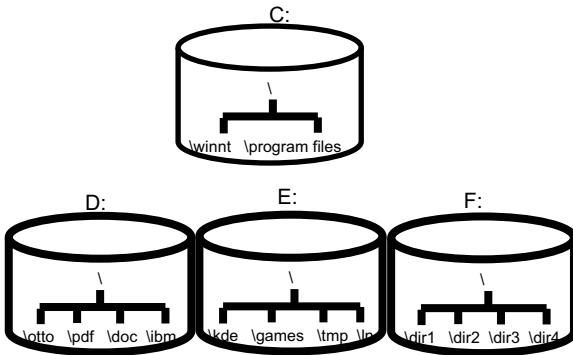
Notes:

Differences in file systems

- Linux: Unified file system
 - Virtual directory tree
 - All physical file systems are mounted



- Windows: Drive letters for each separate file system



© Copyright IBM Corporation 2012

Figure 15-2. Differences in file systems

LX027.2

Notes:

In most cases where people integrate Linux in a Windows environment, dual-boot PC's are used. In this case, data residing on a certain partition has to be read both by Windows and by Linux. Because Windows can't handle Linux partitions, shared file systems are typically a Windows filesystem of some sort. Depending on the Windows version used, and the size of the filesystem, various file systems can be used:

- **fat-12, fat-16:** These are older file systems used with MS-DOS and floppy disks.
- **fat-32:** These are used on Windows-95 formatted hard disks.
- **vfat:** This is virtually identical to fat-32 but supports long file names.
- **NTFS:** This is used by Windows NT and later Windows versions. It is more efficient than all FAT-based file systems, and supports Access Control Lists (ACLs). Note: NTFS support is provided by the NTFS-3G package, which is included in most Linux distributions.

Windows partitions are, under Windows, referred to with a drive letter, followed by a colon. Drives A: and B: are floppy disks, and drive letters C: and up are hard disk partitions. Linux

partitions are not accessed by drive letters, but are mounted into one big, hierarchical file system. The content of the floppy disk can therefore be found under /mnt/floppy, for instance.

To access Windows file systems under Linux, there are basically two options:

- Mounting the partition into the Linux file system
- Using **mtools** to access the file systems by drive letters

Mounting Windows file systems

- To mount the first primary partition on your first hard disk on the mount point /mnt/winC, do the following.

```
# mount /dev/hda1 /mnt/winC
...or
# mount -t ntfs /dev/hda1 /mnt/winC
```

- All files on your C: disk are now accessible in /mnt/winC.
- Make this permanent by adding this to /etc/fstab:

/dev/hda5	/	ext2	defaults	1	1
/dev/hda2	/boot	ext2	defaults	1	2
/dev/hda6	swap	swap	defaults	0	0
/dev/hda1	/mnt/winC	ntfs	defaults	0	0

© Copyright IBM Corporation 2012

Figure 15-3. Mounting Windows file systems

LX027.2

Notes:

Because the Linux kernel supports virtually all Windows file systems, it can simply mount the file system into the Linux virtual filesystem structure. This is done with the **mount** command.

If you want the mount to be permanent, you should add an entry for it to the /etc/fstab file. The **-t** option specifies specifically which filesystem type to expect when mounting. If you are not sure which filesystem to expect, you can always use the **fdisk -l** command to list all of the known volumes and information on each, including the detected filesystem.

Accessing Windows file systems directly

- mtools is a collection of tools that can read/write Windows file systems directly using drive letters.

```
# mcopy c:autoexec.bat /root/autoexec.bat  
# mformat a:  
# mdir a:
```

- Devices must not be mounted!
- Drive letters are mapped to physical devices in /etc/mtools.conf.

```
drive a: file="/dev/fd0" exclusive 1.44m mformat_only  
drive c: file="/dev/hda1"
```

© Copyright IBM Corporation 2012

Figure 15-4. Accessing Windows file systems directly

LX027.2

Notes:

The **mtools** is a collection of programs that can read/write Windows file systems directly, without mounting them, using drive letters. All commands have the same name (with an *m* prepended) and the same syntax as the corresponding MS-DOS/Windows command.

The commands included in the mtools are:

- mattrib:** Change MS-DOS file attribute flags.
- mbadblocks:** Test a floppy disk and marks the bad blocks in the FAT.
- mcat:** Dump raw disk image.
- mcd:** Change MS-DOS directory.
- mcopy:** Copy MS-DOS files to/from UNIX.
- mdel:** Delete an MS-DOS file.
- mdeltree:** Delete an MS-DOS directory tree.
- mdir:** Display an MS-DOS directory.

- **mdu:** Display the amount of space occupied by an MS-DOS directory.
- **mformat:** Add an MS-DOS filesystem to a low-level formatted floppy disk.
- **minfo:** Print the parameters of an MS-DOS filesystem.
- **mlabel:** Make an MS-DOS volume label.
- **mmd:** Make an MS-DOS subdirectory.
- **mmount:** Mount an MS-DOS disk.
- **mmove:** Move or rename an MS-DOS file or subdirectory.
- **mpartition:** Partition an MS-DOS hard disk.
- **mrd:** Remove an MS-DOS subdirectory.
- **mren:** Rename an existing MS-DOS file.
- **mtoolstest:** Tests and displays the mtools configuration.
- **mtype:** Display contents of an MS-DOS file.
- **mzip:** Change protection mode and eject disk on Zip/Jazz drive.

All commands use the /etc/mtools.conf file to determine the drive letter assignation.

Note that to avoid inconsistencies, drives accessed by the mtools may not be mounted!

Running Windows programs

- To run a Windows program, the underlying Windows OS needs to be emulated.
- This can be done at two levels.
 - Emulate a PC and install Windows on it.
 - Emulate Windows itself.
- Note that you need a license for Windows if you use any Windows software (including single DLLs).

© Copyright IBM Corporation 2012

Figure 15-5. Running Windows programs

LX027.2

Notes:

Every Windows program that you want to run under Linux, expects the underlying operating system to be Windows. So you have to somehow emulate this operating system. This can be done in roughly two ways:

- By emulating a PC and installing Windows on this emulated PC
- By emulating Windows itself.

Depending on your needs, one or both solutions can be used.

Licensing note: If you want to run Windows programs under Linux, you might need various components (most often DLLs) from the Windows environment. If you use these components, make sure that you have a license to use these!

Emulators and virtual machines

- Emulate a PC on which you install Windows
- Do need a Windows license
- Open source:
 - Bochs (<http://bochs.soundforge.net>)
 - Xen (<http://www.xen.org>)
 - KVM (www.linux-kvm.org)
 - Qemu (www.qemu.org)
- Commercial:
 - VMWare (<http://www.vmware.com>)
 - VirtualBox (<http://www.virtualbox.org>)

© Copyright IBM Corporation 2012

Figure 15-6. Emulators and virtual machines

LX027.2

Notes:

A PC emulator is a piece of software that emulates a complete PC, including a CPU, memory, BIOS, hard disks and various other devices in software. On this emulated PC you can install basically any PC operating system you want. The disadvantage of such a solution is that there is a performance loss when comparing performance to running the same OS on a native PC. How large this loss is, is dependent on a large number of factors.

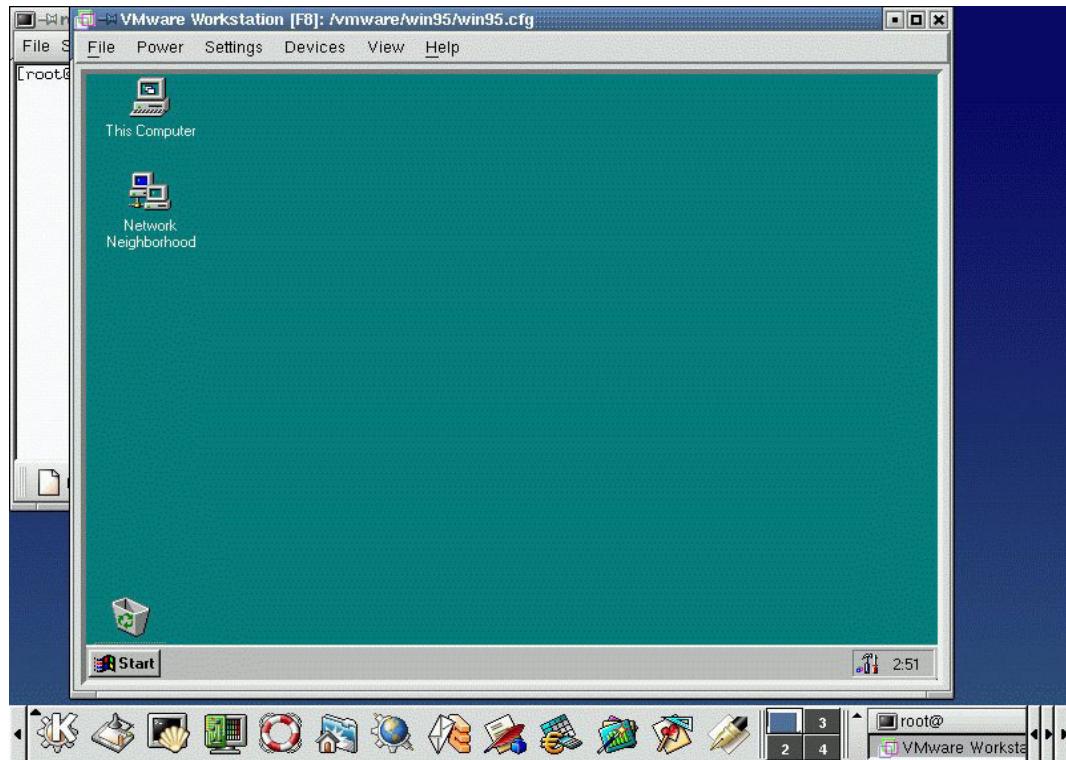
PC emulator example

Bochs is an open source PC emulator that works completely in software: Every CPU instruction is completely handled by the Bochs software. This makes it possible to run Bochs under any POSIX compatible operating system. There is an extreme performance loss using Bochs in a production situation is therefore only recommended for really fast workstations with really low performance requirements regarding the Windows application.

Virtual machines takes the concept of a system emulator and expands the functionality, efficiency, and usability. Acting in concert with a system software virtualization layer called a hypervisor, and some special hardware features, a virtual machine can effectively partition a machine to allow for multiple OS environments to co-exist.

VMWare is a commercial virtualization software that makes use of special features in modern CPUs. These features allow VMWare to let the CPU itself execute most CPU instructions, instead of emulating these CPU instructions in software.

VMWare screenshot



© Copyright IBM Corporation 2012

Figure 15-7. VMWare screenshot

LX027.2

Notes:

The visual shows VMWare in action. When started, it opens a window, and this window is the console to your virtual machine. Once started, the BIOS takes control and proceeds to boot the operating system from the virtual hard disk.

Windows emulators

- WINE (a compatibility layer) (<http://www.winehq.com>):
 - Open source product
 - Does not need a Windows license if only WINE or third party DLLs are used
 - Can use Windows DLLs (beware of license!)
 - To see if your application is supported, go to Web site
- CrossOver Office (<http://www.codeweavers.com>):
 - Commercial extension to WINE

© Copyright IBM Corporation 2012

Figure 15-8. Windows emulators

LX027.2

Notes:

On the other side of the spectrum is the software that can emulate an operating system environment. This solution to running Windows programs under Linux is more lightweight, therefore impacting performance less. Most Windows emulators will require a Windows license since they access certain proprietary DLLs. OS-level emulators have mostly fallen out of favor because of their laborious update schedules and their loss of performance edge versus virtual machines.

Although not strictly an emulator, WINE can be grouped in this category. WINE is an open source Windows compatibility layer, which allows you to run any Windows application without a Windows license; however, some Windows DLLs are still needed for some functions. With native WINE, you need to make all configuration settings to run a particular Windows application yourself.

CrossOver Office is a commercial extension to WINE, developed by CodeWeavers (<http://www.codeweavers.com>). It contains all the necessary settings for a lot of Windows applications, and that makes installation of these applications much smoother.

WINE screenshot

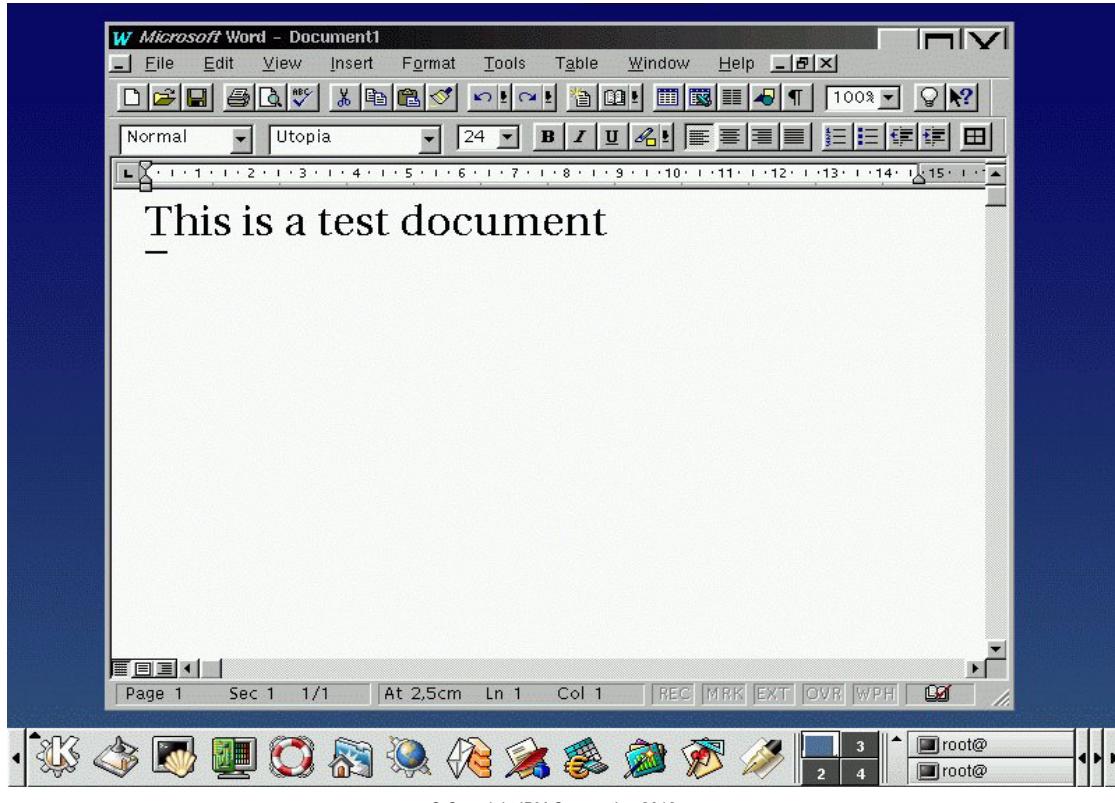


Figure 15-9. WINE screenshot

LX027.2

Notes:

The visual shows Wine in action. An application running under Wine works just like any other application under Linux. Note that in some cases there are minor graphical glitches. Windows application performance under Wine is generally close to native speeds.

Accessing Windows servers

- Samba (<http://www.samba.org>):
 - Open source product
 - Runs on any UNIX
 - Used to replace a Windows server (not covered here)
 - Also includes a number of client tools (smbclient and smbmount)
- smbclient allows you to retrieve information about a Windows server and access files ftp-style.
- mount.cifs allows you to mount Windows shares over the network.

© Copyright IBM Corporation 2012

Figure 15-10. Accessing Windows servers

LX027.2

Notes:

In a Windows environment, your data frequently resides on a Windows server, or you need to access printers that are connected to Windows servers. To be able to do this, you need to act like a Windows client to the server. This can be done using the client tools from the Samba product.

Samba is an open source product which implements the SMB/CIFS networking protocol. It can be run on any UNIX machine and used to replace Windows servers, which use SMB/CIFS for common networking. The server side of Samba is not covered in this course.

Samba includes a number of client tools, which were originally used to test the server side of Samba, but have grown to be an excellent solution when you want to integrate Linux in a Windows environment.

These tools are important in this respect:

- smbclient allows you to retrieve information about a Windows server, and to access files on that server using an FTP-like interface.
- mount.cifs allows you to mount a Windows share and access it as if it were local files.

smbclient examples

```
# smbclient -L winserver -N
# smbclient -L winserver -U user
# smbclient -L winserver -U user%password
# smbclient //winserver/sharename -U user%pw
smb> get file1
smb> put file2
smb> quit
```

- Options
 - **-L**: Lists the shares on the server
 - **-N**: Guest access
 - **-U**: Qualified access

© Copyright IBM Corporation 2012

Figure 15-11. smbclient examples

LX027.2

Notes:

The smbclient tool allows you to do interact with a Windows server. The first thing is to retrieve information about a Window server. This is done with the **-L <servername>** option. The **-N** option allows you to set up anonymous connections, and the **-U <username> or -U <username>%<password>** allows you to retrieve information as a specific user.

Note: If you use the **<username>%<password>** notation, you need to realize that everybody on the system can read your password with a simple **ps** command! You can also access your files using smbclient. This is done by specifying the share as **//winserver/share**. (Note that we are using forward slashes here, because the backslash has a special meaning for the shell. We can use backslashes too, but we need to escape them from the shell. The full command is then **smbclient \\\winserver\\share -U user%pw**.) When the connection has been made, we can then access the files on the share with the ftp-style commands **get**, **put** and so forth.

Mount.cifs examples

- To mount a share as a file system:

```
# mount.cifs //winserver/sharename /mnt/winC
...or
# mount -t cifs //winserver/sharename /mnt/winC
# mount -t cifs -o username=tux1 \
> password=secret //winserver/sharename /mnt/winC
```

- Make permanent by adding this to /etc/fstab:

/dev/hda5	/	ext2 defaults	1 1
/dev/hda2	/boot	ext2 defaults	1 2
/dev/hda6	swap	swap defaults	0 0
//winserver/sharename	/mnt/winC	cifs defaults,username=user%password	0 0

© Copyright IBM Corporation 2012

Figure 15-12. Mount.cifs examples

LX027.2

Notes:

The Linux kernel has support for the cifs filesystem built-in. We can therefore mount a Windows share as any other file system. This is done with the **mount.cifs** command. The -o option allows you to specify the username and password to be used.

If you want to make this permanent, you can add the share to your /etc/fstab file. It is then automatically be mounted when your system boots. Note again that if you put the password on the command line or in the /etc/fstab file, everybody on the system is able to see the password through a **ps** command or by viewing the contents of /etc/fstab!

Note that **smbclient** is a command from the Samba suite, and will happily accept an IP address, DNS hostname or NetBIOS hostname. **mount.cifs** will only accept an IP address or DNS hostname, but no NetBIOS hostname.

Reading Windows document formats

- Most native office programs for Linux read and save Windows document formats.
 - OpenOffice/LibreOffice
 - koffice
 - AbiWord

```
$ file mytext.doc
mytext.doc: Microsoft Office Document
$ oowriter mytext.doc
```

- Note that all document features might not be supported.

© Copyright IBM Corporation 2012

Figure 15-13. Reading Windows document formats

LX027.2

Notes:

Most native office applications for Linux, such as OpenOffice/LibreOffice, koffice and AbiWord can read, and in most cases also write documents in Microsoft Windows native formats. Note, however, that not all document features might be supported. As an example, if you try to open a PowerPoint presentation that has a video clip in it in kpresenter, the presentation is read correctly, but the video clip does not play. This is simply because kpresenter as of yet does not have the ability to play video clips.

At the moment, the most advanced office suite is LibreOffice. LibreOffice is a fork of OpenOffice which happened when Oracle purchased Sun (the official owner of OpenOffice). Most OpenOffice developers migrated to the LibreOffice development team, with the result that OpenOffice development virtually halted.

The latest developments are that Oracle has donated the OpenOffice code to the Apache foundation. In the future, OpenOffice might or might not merge back with the LibreOffice project.¹

¹ For current developments, see the Wikipedia page on LibreOffice.

Other useful programs

- **rdesktop:** Connect to a Windows terminal server
- **VNC:** Allows you to take over a Windows systems screen remotely
- **dos2unix** and **unix2dos:** Convert Windows text files (CR/LF) to UNIX text files (LF)
- **cygwin** (<http://www.cygwin.com>): Series of Linux tools running under Windows

© Copyright IBM Corporation 2012

Figure 15-14. Other useful programs

LX027.2

Notes:

There are a few other programs that might be useful if you try to integrate your Linux workstation in a Windows environment:

- rdesktop is a program that runs under Linux and allows you to connect to a Windows terminal server. This essentially allows you to run applications on the Windows server, with the output displayed in the rdesktop window on your Linux workstation.
- VNC (Virtual Network Computing) allows you to *take over* the screen of another computer. This screen is then displayed locally on your own GUI. VNC works with virtually all operating systems, including Linux and Windows.
- dos2unix and unix2dos are little programs that convert text files in Windows format (using the CR/LF end-of-line standard) to UNIX format (using the single LF end-of-line standard) and vice versa.
- cygwin is a library that implements the UNIX API, combined with a set of Linux tools, that run under Windows. This means that all the tools that you learned how to use in this course, including Vi, are now available under Windows as well.

Unit review

- To access files on Windows file systems, either mount these file systems or use the mtools.
- To run Windows programs, use a PC emulator, such as Bochs or VMWare, or use a Windows emulator, such as WINE.
- CrossOver Office allows you to install Windows programs directly under Linux.
- To access Windows servers, you can use the client programs from the Samba product: smbclient and smbmount.
- To read Windows document formats, you can use almost any native Linux office program: OpenOffice, koffice, AbiWord, and others.
- Several other useful programs exist, including rdesktop.

© Copyright IBM Corporation 2012

Figure 15-15. Unit review

LX027.2

Notes:

Checkpoint

1. True or False: To make a mount of a Windows file system permanent, you need to add it to /etc/fstab.

2. You want to run a third-party program under Linux that was written to run under Windows. Which solution generally does not require you to have a Windows license?
 - a. Bochs
 - b. WINE
 - c. VMWare
 - d. VirtualBox

3. What command allows you to mount a share from a Windows server?

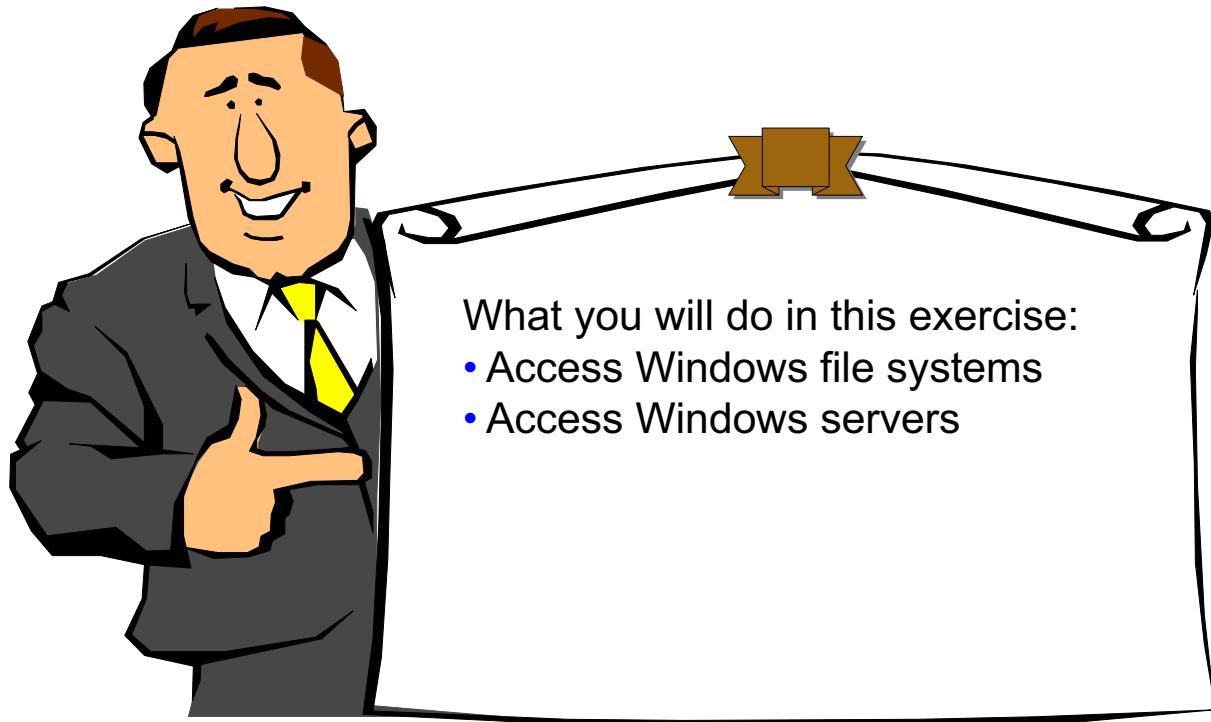
© Copyright IBM Corporation 2012

Figure 15-16. Checkpoint

LX027.2

Notes:

Exercise: Integrating Linux in a Windows environment



© Copyright IBM Corporation 2012

Figure 15-17. Exercise: Integrating Linux in a Windows environment

LX027.2

Notes:

Unit summary

Having completed this unit, you should be able to:

- Access Windows-based file systems
- Run Windows programs
- Access Windows servers
- Read Windows document formats

© Copyright IBM Corporation 2012

Figure 15-18. Unit summary

LX027.2

Notes:

Appendix A. Checkpoint solutions

Unit 1 - Introduction to Linux

Solutions for Figure 1-11, Checkpoint, on page 1-20

Checkpoint solutions

1. True or False: Linus Torvalds wrote the Linux operating system all by himself.

The answer is false.

2. Which of the following statements is *not* true about software licensed under the GNU GPL?

- a. You have the right to obtain and review the source code.
- b. You cannot charge any money for the software.
- c. You cannot change the license statement.
- d. You can modify the source code and subsequently recompile it.

The answer is you cannot charge any money for the software.

3. Who is the mascot of Linux?

The answer is Tux.

© Copyright IBM Corporation 2012

Unit 2 - Installing Linux

Solutions for Figure 2-17, Checkpoint (1 of 2), on page 2-21

Checkpoint solutions (1 of 2)

1. True or False: Linux can coexist with Windows on the same hard disk.
The answer is true.

2. Which of the following steps is not essential in the installation process:
 - a. Create partitions for Linux.
 - b. Configure your printer.
 - c. Select your keyboard type.
 - d. Identify the packages to install.

The answer is configure your printer.

3. What is the best source of information about your hardware?

The answer is a currently installed and configured operating system, such as Windows.

© Copyright IBM Corporation 2012

Solutions for Figure 2-18, Checkpoint (2 of 2), on page 2-22**Checkpoint solutions (2 of 2)**

4. True or False: A network install server needs to be a Linux system.

The answer is false. It can be any operating system that provides NFS, FTP, or HTTP services.

5. Which of the following install methods does not require a network server?

- a. NFS
- b. SMB
- c. FTP
- d. CD-ROM

The answer is CD-ROM.

© Copyright IBM Corporation 2012

Unit 3 - Using the system

Solutions for Figure 3-23, Checkpoint, on page 3-26

Checkpoint solutions

1. True or False: A Linux system always needs a graphical user interface.
The answer is false.
2. Which of the following commands is not a legal command in Linux?
 - a. ls/dev/bin
 - b. ls -al/dev/bin
 - c. ls -a -l .
 - d. ls -a-l/dev

The answer is ls/dev/bin, ls -al/dev/bin, and ls -a-l/dev are all illegal commands.

3. How do you switch between virtual terminals?

The answer is by entering the key combination Alt-Fn or, when in X, Ctrl-Alt-Fn.

© Copyright IBM Corporation 2012

Unit 4 - Working with files and directories

Solutions for Figure 4-39, Checkpoint, on page 4-42

Checkpoint solutions

1. True or False: Linux imposes an internal structure on a regular file (not a directory or special file).
The answer is false.
2. Which of the following is not a legal file name?
 - a. `~tux1/mydocs.tar.gz`
 - b. `/home/tux1/mydoc(1)`
 - c. `/var/tmp/.secret.doc`
 - d. `/home/.../home/tux1/one+one`
The answer is `/home/tux1/mydoc(1)`.
3. What command would you use to copy the file `/home/tux1/mydoc` to `/tmp` and rename it at the same time to `tempdoc`?
The answer is `cp /home/tux1/mydoc /tmp/tempdoc`.

© Copyright IBM Corporation 2012

Unit 5 - File and directory permissions

Solutions for Figure 5-14, Checkpoint, on page 5-17

Checkpoint solutions

```
$ pwd  
/groups/  
$ ls -l  
drwxrwxr-x  2 root penguins 1024 Jan 1 10:03 penguins  
$ ls -l penguins  
-rw-r--r--  1 tux1 penguins 544 Jan 1 10:15 hello.c  
-rw-r--r--  1 tux1 penguins 544 Jan 1 10:15 task.c  
-rw-r--r--  1 tux1 penguins 544 Jan 1 10:15 zip.c
```

- Can tux2 (who is also a member of the penguins group) successfully execute the following commands?
 1. cd /groups/penguins
[The answer is yes.](#)
 2. mkdir /groups/penguins/mydir
[The answer is yes.](#)
 3. cp /groups/penguins/task.c ~/task.c
[The answer is yes.](#)
 4. vi /groups/penguins/zip.c
[The answer is no.](#)
 5. vi /groups/penguins/newfile.c
[The answer is yes.](#)
 6. rm /groups/penguins/hello.c
[The answer is yes.](#)

© Copyright IBM Corporation 2012

Unit 6 - Linux documentation

Solutions for Figure 6-14, Checkpoint, on page 6-16

Checkpoint solutions

1. True or False: A HOWTO document is the best source of documentation if you want up-to-date information about a specific command.

The answer is false.

2. The main Linux documentation Web site is:

- a. <http://www.tldp.org>
- b. <http://www.linux.org>
- c. <http://www.lwn.net>
- d. <http://www.kernel.org>

The answer is <http://www.tldp.org>.

3. In which sections are manual pages divided?

The answers are user commands, system calls, library calls, devices, file formats and protocols, games, conventions, macro packages, and so forth, system administration, and Linux kernel.

© Copyright IBM Corporation 2012

Unit 7 - Editing files

Solutions for Figure 7-21, Checkpoint, on page 7-26

Checkpoint solutions

1. True or False: You need to learn Vi because Vi is the best editor for any job.
The answer is false. Vi is just the most common editor available on Unix-style systems and is thus more likely to be there when you need to edit a file. Indeed, it might be the only editor available. Having Vi skills can be quite handy on unfamiliar Unix-style systems.
2. What does the **file** command do?
 - a. It looks at the extension to determine the type of file.
 - b. It looks at the first few characters of the file and compares this to a database of known file types.
 - c. It asks the kernel for information about the file.
 - d. It makes a wild guess.The answer is it looks at the first few characters of the file and compares this to a database of known file types.
3. What is a hex editor?
The answer is a hex editor is an editor that treats the contents of a file as a series of bytes, displays those bytes in hexadecimal format, and allows the hexadecimal representation of the contents of the file to be edited.

© Copyright IBM Corporation 2012

Unit 8 - Shell basics

Solutions for Figure 8-27, Checkpoint, on page 8-31

Checkpoint solutions

1. True or False: A filter is a command that reads a file, performs operations on this file and then writes the result back to this file.
The answer is false.

2. The output of the **ls** command is:

```
one    two    three   four   five  
six    seven   eight   nine   ten
```

What will the output be if you run the **ls ?e*** command?

- a. three seven ten
- b. seven ten
- c. one three five seven eight nine ten
- d. ?e*

The answer is seven ten.

© Copyright IBM Corporation 2012

Unit 9 - Working with processes

Solutions for Figure 9-20, Checkpoint, on page 9-23

Checkpoint solutions

1. True or False: Any user can send a signal to a process of another user and cause that process to halt.
The answer is false. A normal user will not have the appropriate permission to send signals to another user's processes. The root user does have permission to do this.
2. If a process is hanging, the proper order of trying to terminate it with the lowest chance of data corruption is:
 - a. kill -1, <Ctrl-C>, kill, <Ctrl-\>
 - b. <Ctrl-Z>, kill, kill -9, kill -15, <Ctrl-C>
 - c. kill -9, kill -15, <Ctrl-C>, <Ctrl-Z>
 - d. <Ctrl-C>, <Ctrl-\>, kill, kill -9The answer is <Ctrl-C>, <Ctrl-\>, kill, kill -9. kill -9 is always the last resort.
3. What is a daemon?
The answer is a daemon is a never-ending process which provides a system service.

© Copyright IBM Corporation 2012

Unit 10 - Linux utilities

Solutions for Figure 10-28, Checkpoint, on page 10-30

Checkpoint solutions

1. True or False: The command `ps -aux | grep tux | grep firefox` lists all Firefox processes of a user named tux.
The answer is true. Technically, the command lists all processes for which the `ps` command output includes the *tux* and *firefox* strings. Note that this command will also list all processes of a user named *tux01*, for example, and all processes named *tux* or *tuxedo*, for example. Note also that *all Firefox processes* is nebulous in that a Firefox application might include a process called plug-in container that would not appear in the command output.
2. Which command would best be used to locate all files in your system that begin with the string *team*?
 - a. `find / -name "^team"`
 - b. `find / -name "team*"`
 - c. `find / -name "*team*"`
 - d. `find / -type f -name "team"`The answer is `find / -name "team*"`. Note that the `find` command uses the same metacharacters as the shell.
3. Translate the following command into your native language:
`ls -lR|egrep "txt$|tab$"|sort -rn -k5|tail -n +4|head -5`
The answer is find the fourth through eighth largest files in the current directory tree whose file names end with either *txt* or *tab*.

© Copyright IBM Corporation 2012

Unit 11 - Shell scripting

Solutions for Figure 11-24, Checkpoint (1 of 2), on page 11-27

Checkpoint solutions (1 of 2)

1. What will the following piece of code do?

```
TERMTYPE=$TERM
if [ -n "$TERMTYPE" ]
then
    if [ -f /home/tux1/custom_script ]
    then
        /home/tux1/custom_script
    else
        echo No custom script available!
    fi
else
    echo You don't have a TERM variable set!
fi
```

The answer is if TERMTYPE is set to a non-null value and if custom_script exists as a normal file, the script attempts to execute it. If the script does not exist, this is reported.
If the TERMTYPE is null or not set, that is reported.

© Copyright IBM Corporation 2012

Solutions for Figure 11-25, Checkpoint (2 of 2), on page 11-28

Checkpoint solutions (2 of 2)

2. Write a script that will multiply any two numbers together.

The answer is:

```
#!/bin/bash
echo $(( $1 * $2 ))
```

© Copyright IBM Corporation 2012

Unit 12 - The Linux GUI

Solutions for Figure 12-11, Checkpoint, on page 12-12

Checkpoint solutions

1. True or False: The main configuration file of KDE is /etc/X11/XF86Config.
The answer is false.
2. What statement describes the function of the X Server best?
 - a. It receives input from the keyboard and mouse and forwards this to the appropriate client, and it receives output from the clients and displays this on the screen.
 - b. It performs the window dressing; it makes sure that every application has a border around its windows so that the window can be resized, moved, and iconified.
 - c. It allows the user to type commands while in a graphical environment.
 - d. It shows a set of eyes looking at the cursor.

The answer is it receives input from the keyboard and mouse and forwards this to the appropriate client, and it receives output from the clients and displays this on the screen.
3. How do you start X?
The answer is with the **startx** command or by switching the system to runlevel 5.

© Copyright IBM Corporation 2012

Unit 13 - Customizing the user environment

Solutions for Figure 13-9, Checkpoint, on page 13-10

Checkpoint solutions

1. Which file would you use to customize your environment variables, and why this file?

The answer is \$HOME/.bash_profile because this overwrites the defaults set in /etc/profile. If the settings cannot be exported to subshells, the \$HOME/.bashrc file can be used.

2. What do the following variables define on your system: PATH, PS1

The answer is PATH defines the directories to look for commands and PS1 is your primary shell prompt.

© Copyright IBM Corporation 2012

Unit 14 - Basic system configuration

Solutions for Figure 14-13, Checkpoint, on page 14-19

Checkpoint solutions

1. True or False: When you configure your system as a DHCP client, you do not need to configure IP addresses, and so forth, yourself.
The answer is true.
2. The correct command to install an additional RPM would be which of the following:
 - a. rpm -U xpuzzles.rpm
 - b. rpm -e xpuzzles-5.5.2-4.i386.rpm
 - c. rpm -qip xpuzzles.rpm
 - d. rpm -i xpuzzles-5.5.2-4.i386.rpmThe answer is rpm -i xpuzzles-5.5.2-4.i386.rpm.
3. What is the proper series of commands to install a .tar.gz file?
The answer is:
cd /usr/src
tar -zxvf archive-version.tar.gz
cd <archivename>

Read INSTALL or README file for further installation instructions.

© Copyright IBM Corporation 2012

Unit 15 - Integrating Linux in a Windows environment

Solutions for Figure 15-16, Checkpoint, on page 15-20

Checkpoint solutions

1. True or False: To make a mount of a Windows file system permanent, you need to add it to /etc/fstab.
The answer is true.
2. You want to run a third-party program under Linux that was written to run under Windows. Which solution generally does not require you to have a Windows license?
 - a. Bochs
 - b. WINE
 - c. VMWare
 - d. VirtualBox
The answer is WINE.
3. What command allows you to mount a share from a Windows server?
The answer is mount.cifs.

© Copyright IBM Corporation 2012

Appendix B. List of abbreviations

ACL	access control list
Bash	Bourne again shell
CUPS	Common UNIX Printing System
DHCP	Dynamic Host Configuration Protocol
FTP	File Transfer Protocol
GCC	GNU C Compiler
GID	group ID
GLIBC	GNU C Library
GNOME	GNU Network Object Model Environment
GPL	GNU General Public License
GRUB	Grand Unified Bootloader
GUI	graphical user interface
HTTP	Hypertext Transfer Protocol
ISDN	Integrated Services Digital Network
KDE	K Desktop Environment
MMU	memory management unit
MSN	Multiple Subscriber Number
NFS	Network File System
PID	process ID
PPID	parent process ID
PXE	Preboot Execution Environment
RPM	Red Hat Package Manager
SLES	SUSE Linux Enterprise Server
SMB	Server Message Block
STDERR	standard error
STDIN	standard input
STDOUT	standard output
UID	user ID
VFS	virtual file system
VNC	Virtual Network Computing

VT virtual terminal

YaBOOT Yet Another Bootloader

IBM
®