student guide

Fundamentals of the UNIX System

51434s h.02

training

Contents

Overview

Course Description 1
Student Performance Objectives 1
Student Profile and Prerequisites 5
Reference Documentation 6

Module 1.— Introduction to UNIX

1–1. SLIDE: What Is an Operating System? 1-2
1–2. SLIDE: History of the UNIX Operating System 1-4
1–3. TEXT PAGE: History of the UNIX Operating System 1-6
1–4. SLIDE: Features of UNIX 1-9
1–5. SLIDE: More Features of UNIX 1-13
1–6. SLIDE: The UNIX System and Standards 1-15
1–7. SLIDE: What Is HP-UX? 1-20

Module 2 — Logging In and General Orientation

2–1. SLIDE: A Typical Terminal Session 2-2
2–2. SLIDE: Logging In and Out 2-4
2–3. SLIDE: The Shell —Command Interpretation 2-6
2–4. SLIDE: Command Line Format 2-7
2–5. SLIDE: The Secondary Prompt 2-9
2–6. SLIDE: The Manual 2-10
2–7. SLIDE: Content of the Manual Pages 2-12
2–8. TEXT PAGE: The Reference Manual — An Example 2-14
2–9. SLIDE: The Online Manual 2-15
2–10. SLIDE: Some Beginning Commands 2-18
2–11. SLIDE: The `id` Command 2-19
2–12. SLIDE: The `who` Command 2-21
2–13. SLIDE: The `date` Command 2-22
2–14. SLIDE: The `passwd` Command 2-23
2–15. SLIDE: The `echo` Command 2-25
2–16. SLIDE: The `banner` Command 2-26
2–17. SLIDE: The `clear` Command 2-27
2–18. SLIDE: The `write` Command 2-28
2–19. SLIDE: The `mesg` Command 2-29
2–20. SLIDE: The `news` Command 2-30
2–21. LAB: General Orientation 2-32

Module 3 — Using CDE

3–1. SLIDE: Front Panel Elements 3-2
3–2. SLIDE: Front Panel Pop-Up Menus 3-5
3–3. SLIDE: Workspace Switch 3-6
3–4. SLIDE: Getting Help 3-7
3–5. SLIDE: File Manager 3-10
3–6. SLIDE: File Manager Menu Tasks 3-13
3–7. SLIDE: Using File Manager to Locate Files 3-16
3–8. SLIDE: Deleting Objects 3-18
3–9. SLIDE: Using the Text Editor 3-20
3–10. SLIDE: Running Applications Using the Application Manager 3-25

© 2003 Hewlett-Packard Developmrnt Company, L.P.

3–11. SLIDE: Using Mailer 3-27
3–12. SLIDE: Sending Mail 3-31
3–13. SLIDE: Customizing Mailer 3-34
3–14. SLIDE: Using Calendar 3-37
3–15. SLIDE: Scheduling Appointments 3-39
3–16. SLIDE: To Do Items 3-41
3–17. SLIDE: Browsing Calendars on a Network 3-43
3–18. SLIDE: Granting Access to your Calendar 3-45
3–19. LAB: Using CDE 3-47

Module 4 — Navigating the File System
4–1. SLIDE: What Is a File System? 4-2
4–2. SLIDE: The Tree Structure 4-3
4–3. SLIDE: The File System Hierarchy 4-4
4–4. SLIDE: Path Names 4-7
4–5. SLIDE: Some Special Directories 4-10
4–6. SLIDE: Basic File System Commands 4-13
4–7. SLIDE: `pwd` — Present Working Directory 4-14
4–8. SLIDE: `ls` — List Contents of a Directory 4-15
4–9. SLIDE: `cd` — Change Directory 4-18
4–10. SLIDE: The `find` Command 4-20
4–11. SLIDE: `mkdir` and `rmdir` — Create and Remove Directories 4-21
4–12. SLIDE: Review 4-23
4–13. SLIDE: The File System — Summary 4-24
4–14. LAB: The File System 4-25

Module 5 — Managing Files
5–1. SLIDE: What Is a File? 5-2
5–2. SLIDE: What Can We Do with Files? 5-4
5–3. SLIDE: File Characteristics 5-5
5–4. SLIDE: `cat` — Display the Contents of a File 5-7
5–5. SLIDE: `more` — Display the Contents of a File 5-9
5–6. SLIDE: `tail` — Display the End of a File 5-10
5–7. SLIDE: The Line Printer Spooler System 5-11
5–8. SLIDE: The `lp` Command 5-12
5–9. SLIDE: The `lpstat` Command 5-15
5–10. SLIDE: The `cancel` Command 5-18
5–11. SLIDE: `cp` — Copy Files 5-21
5–12. SLIDE: `mv` — Move or Rename Files 5-23
5–13. SLIDE: `ln` — Link Files 5-25
5–14. SLIDE: `rm` — Remove Files 5-27
5–15. SLIDE: File/Directory Manipulation Commands — Summary 5-29
5–16. LAB: File and Directory Manipulation 5-30

Module 6 — File Permissions and Access
6–1. SLIDE: File Permissions and Access 6-2
6–2. SLIDE: Who Has Access to a File? 6-3
6–3. SLIDE: Types of Access 6-5
6–4. SLIDE: Permissions 6-6
6–5. SLIDE: `chmod` — Change Permissions of a File 6-8

© 2003 Hewlett-Packard Developmrnt Company, L.P.

6–6. SLIDE: **umask** — Permission Mask 6-11
6–7. SLIDE: **touch** — Update Timestamp on File 6-12
6–8. SLIDE: **chown** — Change File Ownership 6-14
6–9. SLIDE: The **chgrp** Command 6-16
6–10. SLIDE: **su** — Switch User Id 6-18
6–11. SLIDE: The `newgrp` Command 6-20
6–12. SLIDE: Access Control Lists 6-22
6–13. SLIDE: File Permissions and Access — Summary 6-25
6–14. LAB: File Permissions and Access 6-26

Module 7 — Shell Basics
7–1. SLIDE: What Is the Shell? 7-2
7–2. SLIDE: Commonly Used Shells 7-4
7–3. SLIDE: POSIX Shell Features 7-7
7–4. SLIDE: Aliasing 7-8
7–5. SLIDE: File Name Completion 7-10
7–6. SLIDE: Command History 7-12
7–7. SLIDE: Re-entering Commands 7-14
7–8. SLIDE: Recalling Commands 7-15
7–9. SLIDE: Command Line Editing 7-17
7–10. SLIDE: Command Line Editing (Continued) 7-19
7–11. SLIDE: The User Environment 7-22
7–12 SLIDE: Setting Shell Variables 7-24
7–13. SLIDE: Two Important Variables 7-26
7–14. TEXT PAGE: Common Variable Assignments 7-28
7–15. SLIDE: What Happens at Login? 7-30
7–16. SLIDE: The Shell Startup Files 7-32
7–17. SLIDE: Shell Intrinsics versus UNIX Commands 7-34
7–18. SLIDE: Looking for Commands — **whereis** 7-35
7–19. TEXT PAGE: Sample **.profile** 7-37
7–20. TEXT PAGE: Sample **.kshrc** and **.logout** 7-38
7–21. LAB: Exercises 7-39

Module 8 — Shell Advanced Features
8–1. SLIDE: Shell Substitution Capabilities 8-2
8–2. SLIDE: Shell Variable Storage 8-3
8–3. SLIDE: Setting Shell Variables 8-5
8–4. SLIDE: Variable Substitution 8-6
8–5. SLIDE: Variable Substitution (Continued) 8-8
8–6. SLIDE: Command Substitution 8-10
8–7. SLIDE: Tilde Substitution 8-12
8–8. SLIDE: Displaying Variable Values 8-14
8–9. SLIDE: Transferring Local Variables to the Environment 8-15
8–10. SLIDE: Passing Variables to an Application 8-17
8–11. SLIDE: Monitoring Processes 8-19
8–12. SLIDE: Child Processes and the Environment 8-21
8–13. LAB: The Shell Environment 8-22

Module 9 — File Name Generation
9–1. SLIDE: Introduction to File Name Generation 9-2
9–2. SLIDE: File Name Generating Characters 9-3

© 2003 Hewlett-Packard Developmrnt Company, L.P.

9–3. SLIDE: File Name Generation and Dot Files 9-4
9–4. SLIDE: File Name Generation — ? 9-5
9–5. SLIDE: File Name Generation — [ ] 9-6
9–6. SLIDE: File Name Generation — * 9-7
9–7. SLIDE: File Name Generation — Review 9-8
9–8. LAB: File Name Generation 9-9

Module 10 — Quoting
10–1. SLIDE: Introduction to Quoting 10-2
10–2. SLIDE: Quoting Characters 10-3
10–3. SLIDE: Quoting — \ 10-4
10–4. SLIDE: Quoting — ' 10-5
10–5. SLIDE: Quoting — " 10-6
10–6. SLIDE: Quoting Summary 10-7
10–7. LAB: Quoting 10-8

Module 11 — Input and Output Redirection
11–1. SLIDE: Input and Output Redirection — Introduction 11-2
11–2. SLIDE: **stdin**, **stdout**, and **stderr** 11-3
11–3. SLIDE: Input Redirection — < 11-5
11–4. SLIDE: Output Redirection — > and >> 11-7
11–5. SLIDE: Error Redirection — 2> and 2>> 11-9
11–6. SLIDE: What Is a Filter? 11-10
11–7. SLIDE: **wc** — Word Count 11-11
11–8. SLIDE: **sort** — Alphabetical or Numerical Sort 11-13
11–9. SLIDE: **grep** — Pattern Matching 11-15
11–10. SLIDE: Input and Output Redirection — Summary 11-17
11–11. LAB: Input and Output Redirection 11-18

Module 12 — Pipes
12–1. SLIDE: Pipelines — Introduction 12-2
12–2. SLIDE: Why Use Pipelines? 12-3
12–3. SLIDE: The | Symbol 12-4
12–4. SLIDE: Pipelines versus Input and Output Redirection 12-6
12–5. SLIDE: Redirection in a Pipeline 12-7
12–6. SLIDE: Some Filters 12-9
12–7. SLIDE: The **cut** Command 12-10
12–8. SLIDE: The **tr** Command 12-12
12–9. SLIDE: The **tee** Command 12-13
12–10. SLIDE: The **pr** Command 12-14
12–11. SLIDE: Printing from a Pipeline 12-16
12–12. SLIDE: Pipelines — Summary 12-17
12–13. LAB: Pipelines 12-18

Module 13 — Using Network Services
13–1. SLIDE: What Is a Local Area Network? 13-2
13–2. SLIDE: LAN Services 13-3
13–3. SLIDE: The **hostname** Command 13-4
13–4. SLIDE: The **telnet** Command 13-5
13–5. SLIDE: The **ftp** Command 13-6

© 2003 Hewlett-Packard Developmrnt Company, L.P.

13–6. SLIDE: The rlogin Command 13-9
13–7. SLIDE: The rcp Command 13-10
13–8. SLIDE: The remsh Command 13-12
13–9. SLIDE: Berkeley — The rwho Command 13-14
13–10. SLIDE: Berkeley — The ruptime Command 13-15
13–11. LAB: Exercises 13-16

Module 14 — Introduction to the vi Editor
14–1. SLIDE: What Is vi? 14-2
14–2. SLIDE: Why vi? 14-4
14–3. SLIDE: Starting a vi Session 14-6
14–4. SLIDE: vi Modes 14-7
14–5. SLIDE: A vi Session 14-9
14–6. SLIDE: Ending a vi Session 14-11
14–7. SLIDE: Cursor Control Commands 14-12
14–8. SLIDE: Cursor Control Commands (Continued) 14-14
14–9. SLIDE: Input Mode: i, a, O, o 14-16
14–10. SLIDE: Deleting Text: x, dw, dd, dG 14-18
14–11. LAB: Adding and Deleting Text and Moving the Cursor 14-20
14–12. SLIDE: Moving Text: p, P 14-23
14–13. SLIDE: Moving Text: p, P (Continued) 14-24
14–14. SLIDE: Copying Text: yw, yy 14-25
14–15. SLIDE: Changing Text: r, R, cw, ... 14-27
14–16. SLIDE: Searching for Text: /, n, N 14-29
14–17. SLIDE: Searching for Text Patterns 14-31
14–18. SLIDE: Global Search and Replace — ex Commands 14-33
14–19. SLIDE: Some More ex Commands 14-35
14–20. TEXT PAGE: vi Commands — Summary 14-37
14–21. LAB: Modifying Text 14-38

Module 15 — Process Control
15–1. SLIDE: The ps Command 15-2
15–2. SLIDE: Background Processing 15-4
15–3. SLIDE: Putting Jobs in Background/Foreground 15-6
15–4. SLIDE: The nohup Command 15-7
15–5. SLIDE: The nice Command 15-9
15–6. SLIDE: The kill Command 15-11
15–7. LAB: Process Control 15-13

Module 16 — Introduction to Shell Programming
16–1. SLIDE: Shell Programming Overview 16-2
16–2. SLIDE: Example Shell Program 16-3
16–3. SLIDE: Passing Data to a Shell Program 16-5
16–4. SLIDE: Arguments to Shell Programs 16-7
16–5. SLIDE: Arguments to Shell Programs (Continued) 16-9
16–6. SLIDE: Some Special Shell Variables — # and * 16-10
16–7. SLIDE: Some Special Shell Variables — # and * (Continued) 16-11
16–8. SLIDE: The shift Command 16-12
16–9. SLIDE: The read Command 16-14

© 2003 Hewlett-Packard Developmrnt Company, L.P.

16–10. SLIDE: The **read** Command (Continued) 16-16
16–11 SLIDE: Additional Techniques 16-17
16–12. LAB: Introduction to Shell Programming 16-19

Module 17 — Shell Programming — Branches
17–1. SLIDE: Return Codes 17-2
17–2. SLIDE: The **test** Command 17-3
17–3. SLIDE: The **test** Command — Numeric Tests 17-4
17–4. SLIDE: The **test** Command — String Tests 17-6
17–5. SLIDE: SLIDE: The **test** Command — File Tests 17-8
17–6. SLIDE: The **test** Command — Other Operators 17-10
17–7. SLIDE: The **exit** Command 17-12
17–8. SLIDE: The **if** Construct 17-13
17–9. SLIDE: The **if-else** Construct 17-15
17–10. SLIDE: The **case** Construct 17-17
17–11. SLIDE: The **case** Construct — Pattern Examples 17-19
17–12. SLIDE: Shell Programming — Branches — Summary 17-20
17–13. LAB: Shell Programming Branches 17-21

Module 18 — Shell Programming — Loops
18–1. SLIDE: Loops — an Introduction 18-2
18–2. SLIDE: Arithmetic Evaluation Using `let` 18-3
18–3. SLIDE: The **while** Construct 18-5
18–4. SLIDE: The **while** Construct — Examples 18-8
18–5. SLIDE: The **until** Construct 18-9
18–6. SLIDE: The **until** Construct — Examples 18-11
18–7. SLIDE: The **for** Construct 18-12
18–8. SLIDE: The **for** Construct — Examples 18-14
18–9. SLIDE: The **break**, **continue** and **exit** Commands 18-16
18–10. SLIDE: **break** and **continue** — Example 18-18
18–11. SLIDE: Shell Programming — Loops — Summary 18-19
18–12. LAB: Shell Programming — Loops 18-20

Module 19 — Offline File Storage
19–1. SLIDE: Storing Files to Tape 19-2
19–2. SLIDE: The **tar** Command 19-4
19–3. SLIDE: The **cpio** Command 19-6
19–4. LAB: Offline File Storage 19-8

Appendix A — Commands Quick Reference Guide
A–1. Commands Quick Reference Guide A-2

Solutions

© 2003 Hewlett-Packard Developmrnt Company, L.P.

Overview

Course Description

This course is designed to be the first course in the UNIX® curriculum and the Linux curriculum presented by Hewlett-Packard. It is intended to introduce anyone (system administrators, programmers, and general users) to UNIX. It assumes that the student knows nothing about UNIX. (UNIX is a registered trademark of the Open Group in the U.S.A. and other countries).

Student Performance Objectives

Upon completion of this course, you will be able to do the following:

Module 1 — Introduction to UNIX

- Describe the basic structure and capabilities of the UNIX operating system.
- Describe HP-UX.

Module 2 — Logging In and General Orientation

- Log in to a UNIX system.
- Log out of a UNIX system.
- Look up commands in the *HP-UX Reference Manual.*
- Look up commands using the online manual.
- Describe the format of the shell's command line.
- Use some simple UNIX system commands for identifying system users.
- Use some simple UNIX system commands for communicating with system users.
- Use some simple UNIX system commands for miscellaneous utilities and output.

Module 3 — Using CDE

- Describe the Front Panel Elements.
- Understand how the Front Panel Pop-Up Menus work.
- Describe the Workspace Switch.
- Describe the Subpanel Controls.
- Understand how to use the Help System.

© 2003 Hewlett-Packard Development Company, L.P.

Overview

- Describe the File Manager.
- Understand how to use the File Manager Menu.
- Locate files using the File Manager.
- Delete files.
- Print files using the Front Panel, the File Manager, and the Print Manager.
- Display Print Spooler Information.
- Understand Printer Management.
- Use the Text Editor.
- Run Applications using the Application Manager.
- Use the Mailer and the Mailer Options, as well as how to create Mailboxes.
- Use the Calendar Manager to Schedule Appointments and To Do Items.
- Describe how to browse other calendars on the Network.
- Describe how to Grant or Prevent Access to Your Calendar.

Module 4 — Navigating the File System

- Describe the layout of a UNIX system's file system.
- Describe the difference between a file and a directory.
- Successfully navigate a UNIX system's file system.
- Create and remove directories.
- Describe the difference between absolute and relative path names.
- Use relative path names (when appropriate) to minimize typing.

Module 5 — Managing Files

- Use the common UNIX system file manipulation commands.
- Explain the purpose of the line printer spooler system.
- Identify and use the line printer spooler commands used to interact with the system.
- Monitor the status of the line printer spooler system.

© 2003 Hewlett-Packard Developmrnt Company, L.P.

Module 6 — File Permissions and Access

- Describe and change the ownership and group attributes of a file.
- Describe and change the permissions on a file.
- Describe and establish default permissions for new files.
- Describe how to change user and group identity.

Module 7 — Shell Basics

- Describe the job of the shell.
- Describe what happens when someone logs in.
- Describe user environment variables and their functions.
- Set and modify shell variables.
- Understand and change specific environment variables such as *PATH* and *TERM*.
- Customize the user environment to fit a particular application.

Module 8 — Shell Advanced Features

- Use shell substitution capabilities, including variable, command, and tilde substitution.
- Set and modify shell variables.
- Transfer local variables to the environment.
- Make variables available to subprocesses.
- Explain how a process is created.

Module 9 — File Name Generation

- Use file name generation characters to generate file names on the command line.
- Save typing by using file name generating characters.
- Name files so that file name generating characters will be more useful.

Module 10 — Quoting

- Use the quoting mechanisms to override the meaning of special characters on the command line.

© 2003 Hewlett-Packard Developmrnt Company, L.P.

Module 11 — Input and Output Redirection

- Change the destination for the output of UNIX system commands.
- Change the destination for the error messages generated by UNIX system commands.
- Change the source of the input to UNIX system commands.
- Define a filter.
- Use some elementary filters such as `sort`, `grep`, and `wc`.

Module 12 — Pipes

- Describe the use of pipes.
- Construct a pipeline to take the output from one command and make it the input for another.
- Use the `tee`, `cut`, `tr`, `more`, and `pr` filters.

Module 13 — Using Network Services

- Describe the different network services in HP-UX.
- Explain the function of a Local Area Network (LAN).
- Find the host name of the local system and other systems in the LAN.
- Use the ARPA/Berkeley Services to perform remote logins, remote file transfers, and remote command execution.

Module 14 — Introduction to the vi Editor

- Use `vi` to effectively edit text files.

Module 15 — Process Control

- Use the `ps` command.
- Start a process running in the background.
- Monitor the running processes with the `ps` command.
- Start a background process which is immune to the hangup (log off) signal.
- Bring a process to the foreground from the background.

© 2003 Hewlett-Packard Developmrnt Company, L.P.

- Suspend a process.
- Stop processes from running by sending them signals.

Module 16 — Introduction to Shell Programming

- Write basic shell programs.
- Pass arguments to shell programs through environment variables.
- Pass arguments to shell programs through the positional parameters.
- Use the special shell variables, `*`, and `#`.
- Use the `shift` and `read` commands.

Module 17 — Shell Programming Branches

- Describe the use of return codes for conditional branching.
- Use the `test` command to analyze the return code of a command.
- Use the `if` and `case` constructs for branching in a shell program.

Module 18 — Shell Programming Loops

- Use the `while` construct to repeat a section of code while some condition remains true.
- Use the `until` construct to repeat a section of code until some condition is true.
- Use the iterative `for` construct to walk through a string of white space delimited items.

Module 19 — Offline File Storage

- Use the `tar` command for storing files to tape.
- Use the `find` and `cpio` commands for storing files to tape.
- Retrieve files that were stored using `tar` or `cpio`.

Student Profile and Prerequisites

There are no prerequisites for this course. It is assumed, however, that students have been exposed to computers, and that they are familiar with the keyboard.

© 2003 Hewlett-Packard Developmrnt Company, L.P.

Reference Documentation

- *HP-UX Reference*, P/N B2355-90033.
- *Shells: User's Guide*, P/N B2355-90046.

The penguin icon is used throughout these course materials to help identify information that is specific to the Linux environment.

© 2003 Hewlett-Packard Developmrnt Company, L.P.

Module 1 — Introduction to UNIX

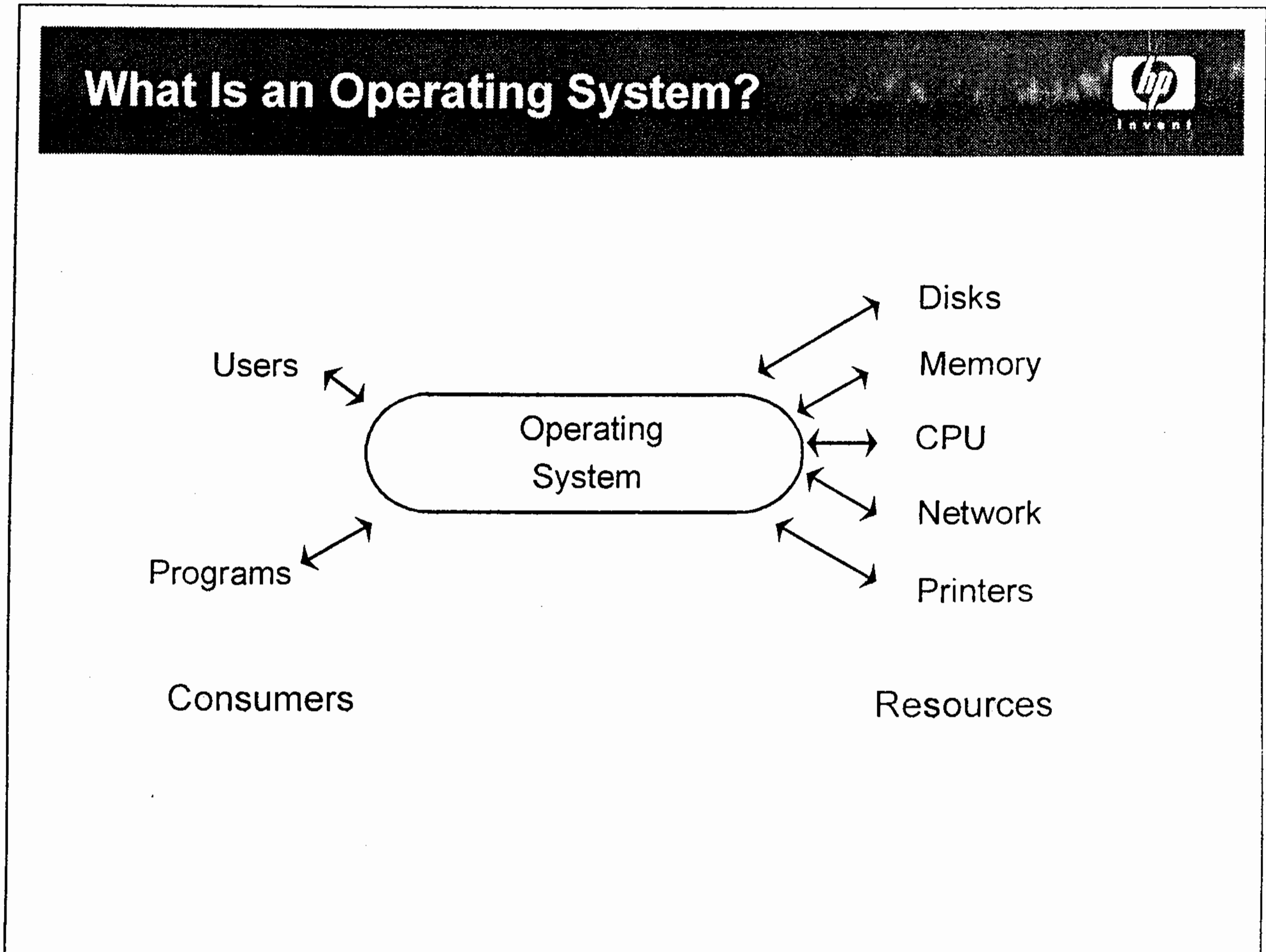
Objectives

Upon completion of this module, you will be able to do the following:

- Describe the basic structure and capabilities of the UNIX operating system.
- Describe HP-UX.

© 2003 Hewlett-Packard Development Company, L.P.

1–1. SLIDE: What Is an Operating System?



Student Notes

An **operating system** is a special computer program (software) that controls the computer (hardware). The operating system serves as a liaison between the consumers and the resources, often coordinating the allocation of limited resources among numerous consumers. The resources include, for example, the CPU, disks, memory, and printers and the consumers are running programs requiring access to the resources. As an example, a user (or a program) requests to store a file on the disk, the operating system intervenes to manage the allocation of space on the disk, and the transfer of the information from memory to the disk.

When a user requests program execution, the operating system must allocate space in memory to load and access the program. As the program executes, it is allowed access to the Central Processing Unit (CPU). In a time-sharing system, there are often several programs trying to access the CPU at the same time.

The operating system controls how and when a program will have its turn in the CPU, similar to a policeman directing traffic in a complex intersection. The intersection is analogous to the CPU; there is only one available.

© 2003 Hewlett-Packard Development Company, L.P.

Each road entering the intersection is like a program. Traffic from only one road can access the intersection at any one time, and the policeman specifies which road has access to the intersection, eventually giving all roads access through the intersection.

© 2003 Hewlett-Packard Development Company, L.P.

1–2. SLIDE: History of the UNIX Operating System

History of the UNIX Operating System

Late 1960s	AT&T development of MULTICS
1969	AT&T Bell Labs UNIX system starts
Early 1970s	AT&T development of UNIX system
Mid 1970s	University of California at Berkeley (BSD) and other universities also research and develop UNIX system
Early 1980s	Commercial interest in UNIX system DARPA interest in BSD Hewlett-Packard introduces HP-UX
Late 1980s	Development of standards Open Software Foundation (OSF) founded
Early 1990s	POSIX, standardization of the interactive user interface

Student Notes

The UNIX operating system started at Bell Laboratories in 1969. Ken Thompson, supported by Rudd Canaday, Doug McIlroy, Joe Ossana, and Dennis Ritchie, wrote a small general-purpose time-sharing system which started to attract attention. With a promise to provide good document preparation tools to the administrative staff at the Labs, the early developers obtained a larger computer and proceeded with the development.

In the mid 1970s the UNIX system was licensed to universities and gained a wide popularity in the academic community for the following reasons:

- It was small — early systems used a 512-kilobyte disk, using 16 kilobytes for the system, 8 KB for user programs, and 64 KB for files.

- It was flexible — the source was available, and it was written in a high-level language that promoted the portability of the operating system.

- It was cheap — universities were able to receive a UNIX system license basically for the price of a tape. Early versions of the UNIX system provided powerful capabilities that were available only in operating systems that were running on more expensive hardware.

These advantages offset the disadvantages of the system at the time:

© 2003 Hewlett-Packard Development Company, L.P.

- It had no support — AT&T had spent enough resources on MULTICS and was not interested in pursuing the UNIX operating system.
- It was buggy — and since there was no support, there was no guarantee of bug fixes.
- It had little or no documentation, but you could always go to the source code.

When the UNIX operating system reached the University of California at Berkeley, the Berkeley users created their own version of the system. Supported by the Department of Defense, they incorporated many new features. Berkeley, as a research institute, offered its licensees a support policy similar to AT&T's — none!

AT&T recognized the potential of the operating system and started licensing the system commercially. To enhance their product, they united internal UNIX system development that was being completed in different departments within AT&T, and also started to incorporate enhancements that Berkeley had developed.

Later success can be attributed to:

- A flexible user interface, and an operating environment that *includes* numerous utilities.
- The modularity of the system design that allows new utilities to be added.
- Capability to support multiple processes and multiple users concurrently.
- DARPA support at Berkeley.
- Availability of relatively powerful and cheap microcomputers.
- Availability of the UNIX system on a wide range of hardware platforms.
- Standardization of the interface definition to promote application portability.

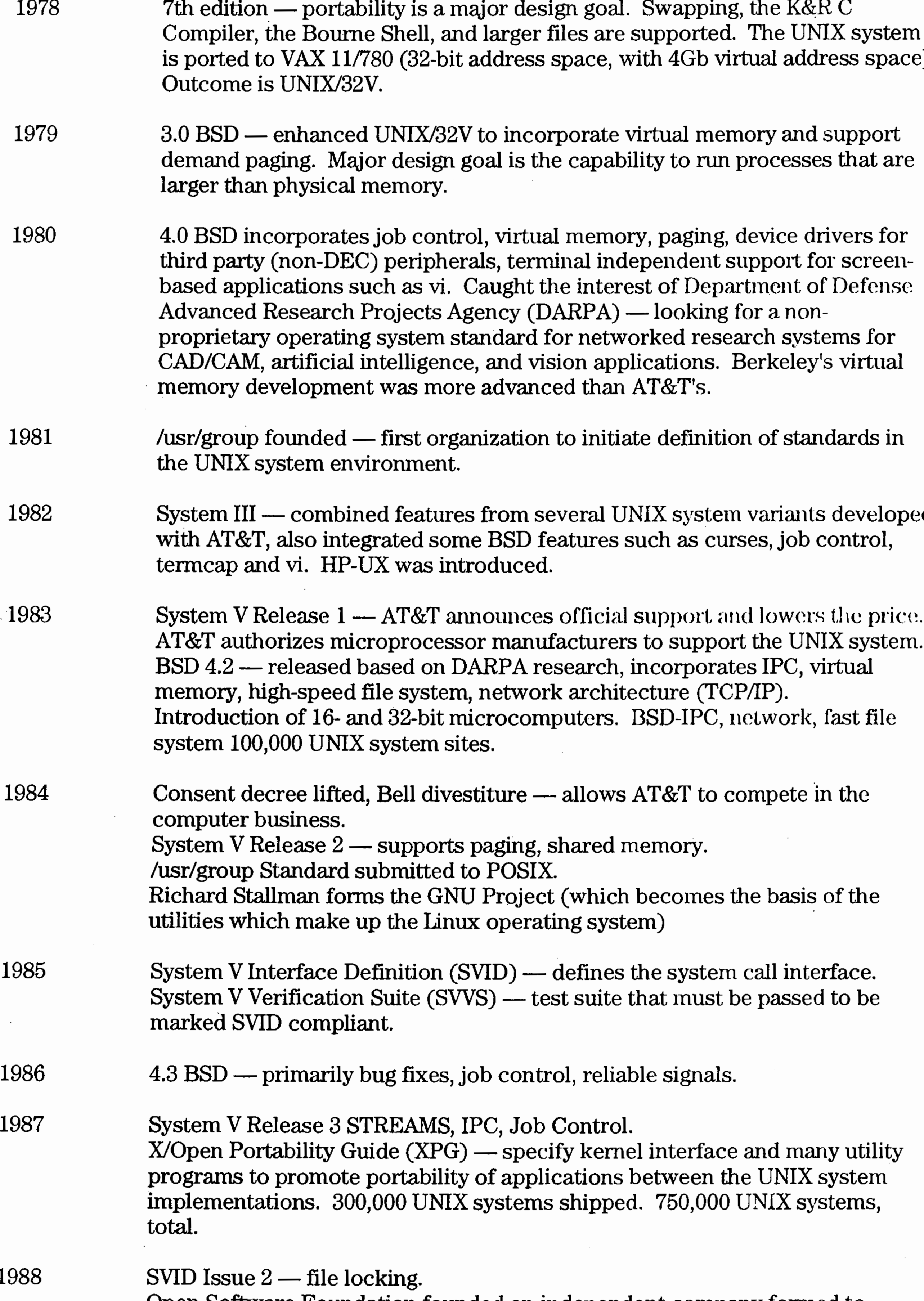
© 2003 Hewlett-Packard Development Company, L.P.

1–3. TEXT PAGE: History of the UNIX Operating System

The following provides some more detail on the history of the UNIX system:

1956	AT&T Consent Decree — AT&T antitrust lawsuit that prohibited them from participating in certain nonregulated areas.
1965	Bell Labs, MULTIplexed Information and Computing System (MULTICS) research begins on the *ultimate* multi-user environment, a joint project with Massachusetts Institute of Technology and General Electric.
1969	Bell Labs, the UNIX system is born — Ken Thompson, during research on file system development creates Space Travel, a program to simulate the motion of bodies in space on a discarded PDP-7 minicomputer. Created a file system, assembler, editor and a simple shell. WHY a PDP7? It had *good graphics*, it was *cheap* when compared to the DEC-10 that supported an interactive, time-sharing interface, and he wanted *convenient, interactive computer service.* Previous work was done on a GE645 mainframe that operated in batch mode and was expensive to access. Programs were originally cross-compiled for the PDP-7 and loaded through paper tape. Due to the Consent Decree, Bell Labs was allowed to research the UNIX system, but could not market, advertise or support any the UNIX system-based products. They were allowed to distribute software to universities for educational purposes only.
1970	Assembler based UNIX system ported to PDP-11/20 (16 bit minicomputer) to research text processing capabilities.
1971	1st edition — Bell Labs Patent Office are the first UNIX system customers. Big advantage for users to not have to go through central computing service. Ken Thompson develops interpreted language B, based on Martin Richards' BCPL language, and subsequently the language NB (new B).
1972	2nd edition — pipes, language support, attempt to write kernel in NB (a predecessor to C). 10 systems. Dennis Ritchie develops the C language.
1973	4th edition — The kernel and shell are rewritten in C. UNIX Systems Group created at Bell Labs for internal support. 25 systems. First unofficial distribution to universities.
1974	5th edition — Officially available to universities for *educational purposes only*. AT&T provides NO support, NO trial period, NO warranty, NO bug fixes, you MUST pay in advance.
1975	6th edition — Licenses are available to government & commercial users. Thompson attends University of California at Berkeley (UCB) on sabbatical. Berkeley development starts.
1977	500 systems, mostly at 125 universities. 1 BSD developed on PDP-11. First ports to non-DEC equipment.

© 2003 Hewlett-Packard Development Company, L.P.



1978	7th edition — portability is a major design goal. Swapping, the K&R C Compiler, the Bourne Shell, and larger files are supported. The UNIX system is ported to VAX 11/780 (32-bit address space, with 4Gb virtual address space). Outcome is UNIX/32V.
1979	3.0 BSD — enhanced UNIX/32V to incorporate virtual memory and support demand paging. Major design goal is the capability to run processes that are larger than physical memory.
1980	4.0 BSD incorporates job control, virtual memory, paging, device drivers for third party (non-DEC) peripherals, terminal independent support for screen-based applications such as vi. Caught the interest of Department of Defense Advanced Research Projects Agency (DARPA) — looking for a non-proprietary operating system standard for networked research systems for CAD/CAM, artificial intelligence, and vision applications. Berkeley's virtual memory development was more advanced than AT&T's.
1981	/usr/group founded — first organization to initiate definition of standards in the UNIX system environment.
1982	System III — combined features from several UNIX system variants developed with AT&T, also integrated some BSD features such as curses, job control, termcap and vi. HP-UX was introduced.
1983	System V Release 1 — AT&T announces official support and lowers the price. AT&T authorizes microprocessor manufacturers to support the UNIX system. BSD 4.2 — released based on DARPA research, incorporates IPC, virtual memory, high-speed file system, network architecture (TCP/IP). Introduction of 16- and 32-bit microcomputers. BSD-IPC, network, fast file system 100,000 UNIX system sites.
1984	Consent decree lifted, Bell divestiture — allows AT&T to compete in the computer business. System V Release 2 — supports paging, shared memory. /usr/group Standard submitted to POSIX. Richard Stallman forms the GNU Project (which becomes the basis of the utilities which make up the Linux operating system)
1985	System V Interface Definition (SVID) — defines the system call interface. System V Verification Suite (SVVS) — test suite that must be passed to be marked SVID compliant.
1986	4.3 BSD — primarily bug fixes, job control, reliable signals.
1987	System V Release 3 STREAMS, IPC, Job Control. X/Open Portability Guide (XPG) — specify kernel interface and many utility programs to promote portability of applications between the UNIX system implementations. 300,000 UNIX systems shipped. 750,000 UNIX systems, total.
1988	SVID Issue 2 — file locking. Open Software Foundation founded an independent company formed to

© 2003 Hewlett-Packard Development Company, L.P.

develop and provide a computing environment that is based on industry standards and the best technologies that are available.

1989 System V Release 4 — POSIX.1 compliance.
XPG/3support POSIX.1 and Common Application Environment, selects standards that will be incorporated for several aspects of the computing environment, not just the operating system interface to promote portability.

1990 SVID Issue 3 — POSIX.1, FIPS 151-1 and C Standard.

1991 HP-UX 8.0 — licensee of System V Release 3, SVID2 compliant, incorporating BSD4.2 and BSD4.3 extensions that have become de facto industry standards, incorporating POSIX-, FIPS-, XPG2-, and XPG3-compliant interfaces.

Linus Torvalds creates the **`linux`** kernel. The availability of this kernel allows the GNU open-source software to be packaged as a complete operating system. The "GNU/Linux" operating system later becomes known by the abridged name: "Linux".

1992 HP-UX 9.0 — licensee of System V Release 3, SVID2 compliant, incorporating X/Open Portability Guide Issue 3, POSIX 1003.1 and POSIX 1003.2, X11R5, FIPS-2 and FIPS-3, POSIX.1, OSF/Motif 1.2, and others.

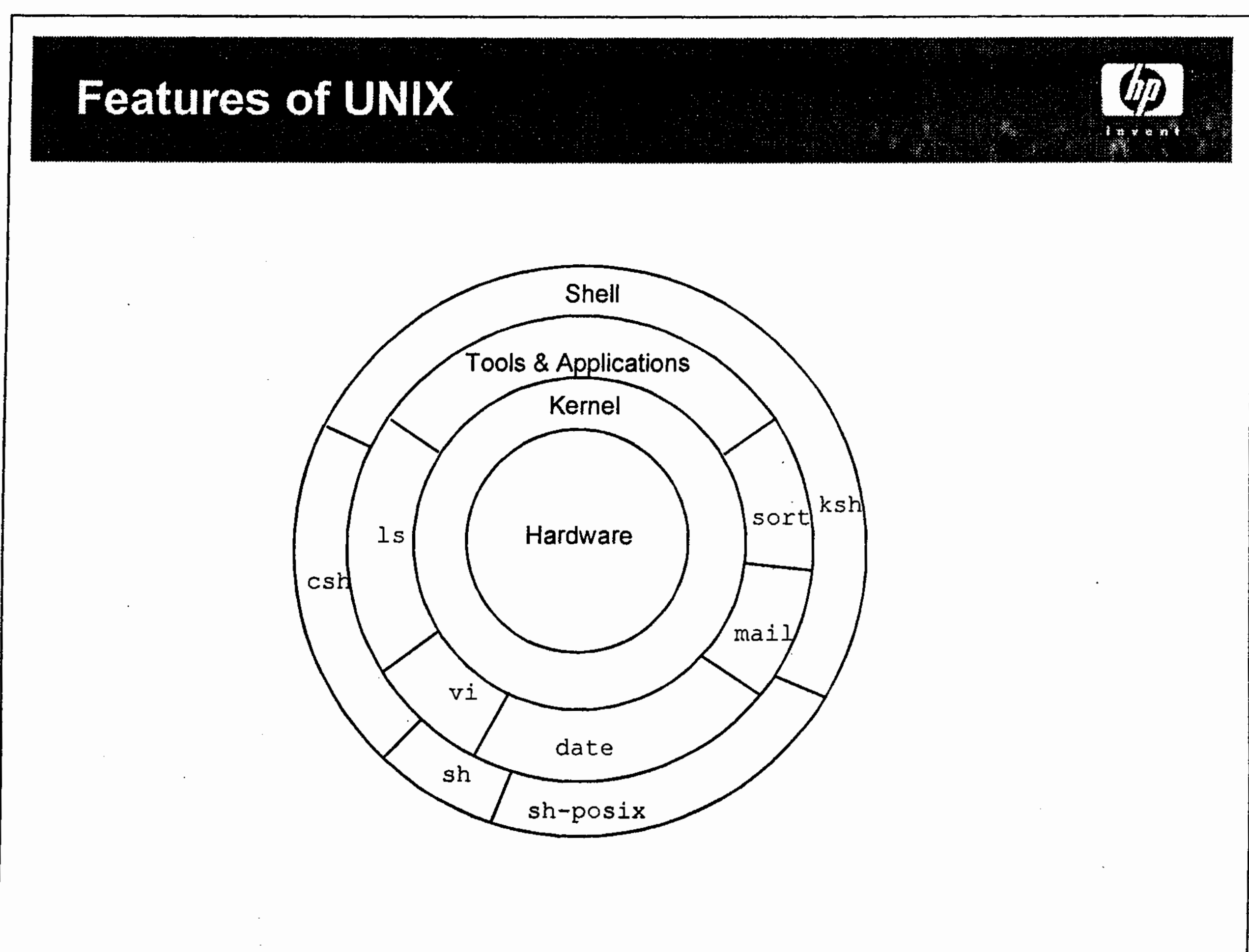
1995 HP-UX 10.0 SVID3 kernel compliance, incorporating X/Open Portability Guide Issue 4, POSIX.4 Realtime Phase 1 and others assuring portability from 9.0 to 10.0. The major difference is that the file system layout has been changed to follow the AT&T SVR4 and OSF/1 paradigm.

1997 HP-UX 11.0 — SVID3 Release 4 — POSIX.2 compliance. IA64 compliant for 64-bit implementations. Implements kernel threads.

1999 Red Hat Inc. release Red Hat Linux version 6.0. This version of Linux quickly becomes a commercial standard and is implemented on a variety of computer manufacturers systems.

© 2003 Hewlett-Packard Development Company, L.P.

1–4. SLIDE: Features of UNIX



Student Notes

The UNIX system provides a time-sharing operating system that controls the activities and resources of the computer, and an *interactive*, flexible operating interface. It was designed to run multiple processes concurrently and support multiple users to facilitate the sharing of data between members of a project team. The operating environment was designed with a modular architecture at all levels. When installing the UNIX system, you only need install the pieces that are relevant to your operating needs, and omit the excess. For example, the UNIX system supplies a large collection of program development utilities, but if you are not doing program development you need only to install the minimal compiler. The user interface also effectively supports the modular philosophy. Commands that know nothing about each other can be easily combined through pipelines, to perform quite complex manipulations.

The Operating System

The **kernel** is the operating system. It is responsible for managing the available resources and access to the hardware. The kernel contains modules for each hardware component that it interfaces with. These modules provide the functionality that allows programs access to the CPU, memory, disks, terminals, the network, and so forth. As new types of hardware are installed on the system, new modules can be incorporated into the kernel.

© 2003 Hewlett-Packard Development Company, L.P.

The Operating Environment

Tools and Applications

The modular design of the UNIX system environment is most evident in this layer. The UNIX system command philosophy is that each command does one thing well, and the collection of commands make up a tool box. When you have a job to complete you pull out the appropriate tools. Complex tasks can be performed by combining the tools appropriately.

From its inception, the UNIX system "toolbox" has included much more than just the basic commands required to interact with the system. The UNIX system also provides utilities for

electronic mail (**mail**, **mailx**)
file editing (**ed**, **ex**, **vi**)
text processing (**sort**, **grep**, **wc**, **awk**, **sed**)
text formatting (**nroff**)
program development (**cc**, **make**, **lint**, **lex**)
program management (SCCS, RCS)
inter-system communications (**uucp**)
process and user accounting (**ps**, **du**, **acctcom**)

Since the UNIX system user environment was designed with an interactive, programmable, modular implementation, new utilities can easily be developed and added to the user's toolbox, and unnecessary tools can be omitted without impairing system operation.

As an example, an application programmer and a technical writer are using UNIX systems. They will use many common commands, even though their applications are very different. They will also use utilities that are appropriate just for their development. The application programmer's system will include utilities for program development and program management, while the technical writer's system will contain utilities for text formatting and processing, and document management. It is interesting to note that the utility that the application developer uses for program revision control can also be used by the technical writer for document revision control. Therefore, their systems will look very similar, yet each user has selected and discarded the modules that are relevant to his or her application needs.

The popularity of the UNIX system can largely be attributed to

- The completeness and the flexibility of the UNIX system allowing it to fit into many application environments.

- The numerous utilities that are included in the operating environment enhancing users' productivity.

- The availability on and portability to many hardware platforms.

The Shell

The **shell** is an *interactive* command interpreter. Commands are entered at the shell prompt, and acted upon as they are issued. A user communicates with the computer through the shell. The shell gathers the input the user enters at the keyboard and translates the command into a form that the kernel can understand. Then the system will execute the command.

© 2003 Hewlett-Packard Development Company, L.P.

You should notice that the shell is *separate* from the kernel. If you do not like the interface provided by the supplied shell, you can easily replace it with another shell. Many shells are currently available. Some are command driven and some provide a menu interface. The common shells that are supplied with the UNIX system include both a command interpreter and a programmable interface.

There are four shells that are commonly available in the UNIX system environment. They are

- Bourne Shell (`/usr/old/bin/sh`) — the original shell provided on AT&T based systems developed by Stephen Bourne at Bell Laboratories. It provides a UNIX system command interpreter and supports a programmable interface to develop shell programs, or scripts as they are commonly called. The programmable and interactive interfaces provide capabilities such as variable definition and substitution, variable and file testing, branching, and loops.

- C Shell (`/usr/bin/csh`) — the shell developed at the University of California Berkeley by Bill Joy, and is provided on BSD-based systems. This shell was referred to as the California Shell, which was shortened to just the C Shell. It was considered an improvement over the Bourne Shell because it offered interactive features such as a command stack which allows simple recalling and editing of previously entered commands, and aliasing which provides personalized alternative names for existing commands.

- Korn Shell (`/usr/bin/ksh`) — is a more recent development from Bell Laboratories developed by David Korn. It can be considered an enhanced Bourne Shell because it supports the simple programmable interface of the Bourne Shell, but has the convenient interactive features of the C Shell. The code has also been optimized to provide a faster, more efficient shell. A GNU version of the Korn shell is available with Linux.

- POSIX Shell (`/usr/bin/sh`) — POSIX-conformant command programming language and command interpreter residing in file `/usr/bin/sh`. This shell is similar to the Korn shell in many respects; it provides a history mechanism, supports job control, and provides various other useful features.

Bash shell (`/usr/bin/bash`) — is a variant of the Bourne shell, which has been updated with POSIX-conformant options and controls. The name bash is an acronym for **B**ourne **A**gain **Sh**ell. This shell provides functions and mechanisms very similar to the POSIX shell.

© 2003 Hewlett-Packard Development Company, L.P.

Table 1. Comparison of Shell Features

Features	Description	Bourne	Bash	Korn	C	POSIX
Command history	A feature allowing commands to be stored in a buffer, then modified and reused.	No	Yes	Yes	Yes	Yes
Line editing	The ability to modify the current or previous command lines with a text editor.	No	Yes	Yes	No	Yes
File name completion	The ability to automatically finish typing file names in command lines.	No	Yes	Yes	Yes	Yes
Alias command	A feature allowing users to rename commands, automatically include command options, or abbreviate long command lines	No	Yes	Yes	Yes	Yes
Restricted shells	A security feature providing a controlled environment with limited capabilities.	Yes	No	Yes	No	Yes
Job control	Tools for tracking and accessing processes that run in the background.	No	Yes	Yes	Yes	Yes

© 2003 Hewlett-Packard Development Company, L.P.

1–5. SLIDE: More Features of UNIX

More Features of UNIX

- Hierarchical file system
- Multi-tasking
- Multi-user

Student Notes

Hierarchical File System

Information is stored on the disk in containers known as **files**. Every file is assigned a name, and a user accesses a file by referencing its name. Files normally contain data, text, programs, and so on. A UNIX system normally contains hundreds of files, so another container, the **directory,** is provided that allows users to organize their files into logical groupings. In the UNIX system, a directory can be used to store files or other directories.

The file system structure is very flexible, so if a user's organizational needs change, files and directories can be easily moved, renamed, or grouped into new or different directories through simple UNIX system commands. The file system, therefore, is like an electronic filing cabinet. It allows users to separate and organize their information into directories that are most appropriate for their environment and application.

Multitasking

In the UNIX system several tasks can be performed at the same time. From a single terminal, a single user can execute several programs that all seem to be running simultaneously. This

© 2003 Hewlett-Packard Development Company, L.P.

means that a user can edit a text file, while another file is being formatted, while yet another file is being printed.

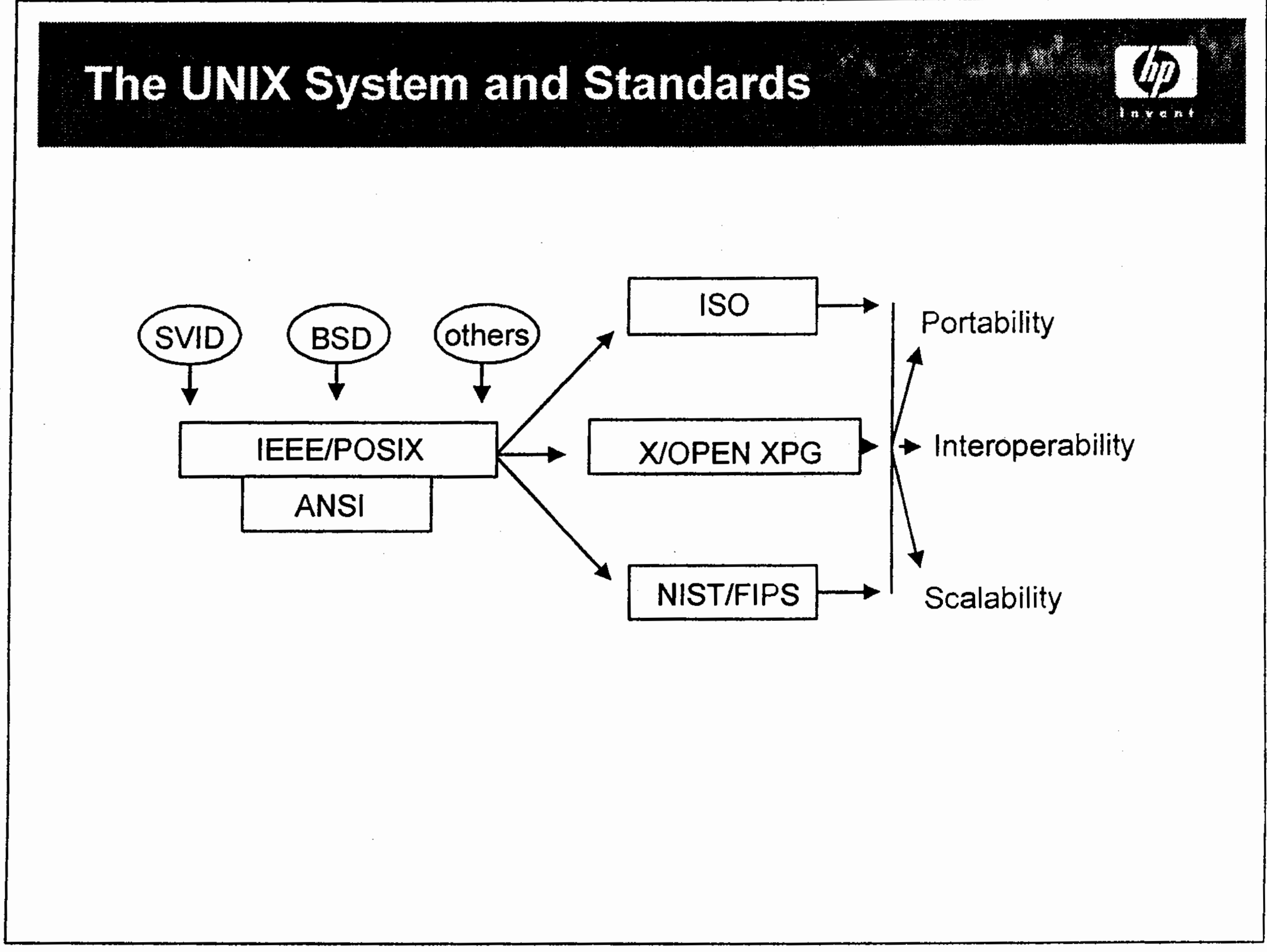
In actuality, the CPU can execute only one task at a time, but the UNIX operating system has the capability to time-share the CPU between multiple processes that are scheduled to run at the same time. So, to the user, it appears that all programs are executing simultaneously.

Multiuser

Multi-user capability allows more than one user to log in and use the system at the same time. Multiple terminals and keyboards can be attached to the same computer. This is a natural extension of the multi-tasking capability. If the system can run multiple programs simultaneously, some of those multiple programs should be able to support other user sessions. In addition, a single user could log in multiple times to the same system through multiple terminals. A big advantage of this architecture is that members of a work group can have access to the same data at the same time, either from a development or a user viewpoint.

© 2003 Hewlett-Packard Development Company, L.P.

1–6. SLIDE: The UNIX System and Standards



Student Notes

From its inception, the UNIX system was developed with a focus on portability. Since most of the operating system and utilities have been written in C (as opposed to assembler), the UNIX system has not been restricted to one processor or hardware platform. On the other hand, since the UNIX system is written in a high level language, it is easy to modify, as exemplified by the over 100 companies that offer UNIX-based implementations (licensed from The Open Group) and the UNIX system clones (new implementations of a UNIX-like interface that do not require an Open Group license). Even though most systems are derived from AT&T UNIX, BSD UNIX, or a combination of both, each implementation may incorporate unique extensions to the operating system, such as real time capabilities, that may negate the compatibility between different UNIX system implementations. (Actually, System V has already incorporated many of the popular features of BSD.) To encourage consistency from implementation to implementation standards are being formulated for the UNIX system operating environment.

The goal of these standards is to promote the following:

1. Portability — the ability to easily transfer an application from one UNIX system implementation to another.

© 2003 Hewlett-Packard Development Company, L.P.

2. Interoperability — the ability for applications running on different UNIX system implementations to share information.

3. Scalability — provide a range of hardware options, from small systems to large systems, users can select from depending on their application needs. Plus allow flexible system upgrade capabilities as application needs grow.

Despite the many implementations of the UNIX system, the differences at the user level are slight, since most have been developed from common origins. Therefore, the standards initially focused on the source code interface to the kernel, and are only recently evaluating standardization of the interactive user interface.

Goals of the Standards Bodies

Define Interface Not Implementation

Standards are not intended to define a totally new interface but to create a well-defined, portable interface based on current UNIX system implementations. It is important to understand that standards are intended to define interfaces to the UNIX system operating environment, not how a standard is to be implemented. Therefore, the UNIX system standards do not dictate that all UNIX system computers be complete duplicates, rather that they will all support a common set of functions that specific implementations can be formed around.

A good analogy would be an automobile. The basic interface defined by the automobile "standard" is

Go — step on accelerator

Stop — step on brake

Change directions — turn wheel

Start engine — turn key

Automobiles that support these standard interfaces can be designed with many different implementations. For example, the automobile could have an electric engine or a gas engine, but stepping on the accelerator would make either go.

An interesting side effect of this philosophy is that it will be possible for non-UNIX operating systems to comply with the defined standards by supporting the prescribed interfaces.

Modularity

The computing environment is continually changing and growing. The standards should be extensible. They should be able to keep up with advances in technology and user demands.

Standards are being defined in a modular fashion so that they can be added to or possibly replaced when a better interface emerges.

© 2003 Hewlett-Packard Development Company, L.P.

AT&T System V Interface Definition (SVID)

AT&T was the first to develop a standard in the UNIX system operating environment. Their standard, based on AT&T's System V, focuses on the function level interface to the operating system (system calls), interprocess communications, the UNIX system shell and some basic utilities.

AT&T also developed the System V Verification Suite (SVVS) to verify SVID compliance.

Although the SVID was the first attempt to develop a standard, the standard was not vendor-neutral, since AT&T was the definitive body. For example, at System V Release 3 (System V.3), AT&T enforced such strict qualifications for implementations that desired System V.3 endorsement, that certain Berkeley extensions would negate System V.3 compliance.

IEEE/POSIX

The Institute of Electrical and Electronics Engineers (IEEE) sponsors the Portable Operating System Interface for computer environments (POSIX). POSIX originated from the 1984 /usr/group Standard, whose goal was to define standards beyond the SVID (/usr/group is the predecessor to UniForum). POSIX 1003 was set up to develop standards for the *complete operating environment*, not just the kernel interface. Unlike AT&T, POSIX defines a programming interface without defining the implementation. Therefore, POSIX-compliant systems can be developed that are not derived from AT&T code.

POSIX has also been submitted to the ISO for inclusion in the international standard. It is associated with the draft proposed standard TC22 WG15.

To advance the standard development, POSIX has been partitioned into several components, and a working group assigned to each.

1003.1 System Interface (formed 1981).
Provides a source code, programmatic interface bound to a high level language that facilitates application portability. POSIX.1 is closely related to SVID Issue 2 (SVID2), but also includes features from BSD 4.3 and additional features that are not supplied with either interface definition.

1003.2 Shells and Utilities (formed 1984).
Defines a shell command language and interactive utilities.

1003.3 Testing Methods (formed 1986).
Defines the general requirements for how test suites should be written and administered. Provides a list of test assertions showing exactly what in the POSIX standard has to be tested. This work group will not be authoring the test suites, and the method of testing is left up to the vendor.

1003.4 Real Time
4a Thread Extensions
4b Language Independent Specification

1003.5 Ada Binding for POSIX

1003.6 Security

© 2003 Hewlett-Packard Development Company, L.P.

1003.7	System Administration
1003.8	Networking
1003.9	FORTRAN Binding for POSIX
1003.10	Supercomputing Application Execution Profile (AEP)
1003.11	Transaction Processing AEP
1003.12	Protocol Independent Interfaces
1003.13	Real Time AEP
1003.14	Multiprocessing AEP

X/Open and The Open Group

X/Open has been an international consortium of information system suppliers, users, system integrators and software developers who joined to define a Common Application Environment. Their mission was not to define new standards, but select from existing standards those that will ensure portability and interworking of applications, and allow users to move between systems without additional training. X/Open also has its origins from SVID, but is a superset of POSIX. X/Open's Portability Guide (XPG, currently revision 4) includes a set of relevant standards that address the entire application environment.

X/Open has recently merged with the Open Software Foundation (OSF) to form The Open Group.

Some elements include the following:

Component	**Defining Standard**
System Calls & Libraries	POSIX 1003.1
Commands & Utilities	POSIX 1003.2
C Language	ANSI
COBOL Language	ANSI/ISO
FORTRAN Language	ANSI
Pascal Language	ISO
SQL	ANSI
Window Manager	X Window System

© 2003 Hewlett-Packard Development Company, L.P.

American National Standards Institute (ANSI)

The coordinating organization for voluntary standards in the USA. IEEE is an accredited standards committee of ANSI.

International Standards Organization (ISO)

Coordinates the adoption of international standards for distributed information systems in an open systems environment (an environment of heterogeneous networked systems). Most notable developments have been in the area of networking and the definition of the seven layer Open Systems Interconnection (OSI) network reference model.

The ISO participants generally come from national standards organizations of the member countries. In the USA, ANSI is an ISO participant.

National Institute of Standards and Technology Federal Information

Processing Standard (NIST/FIPS)

The NIST was formerly the National Bureau of Standards (NBS) and is under the direction of the Department of Commerce. This organization is developing standards requirements for governmental agencies. Their original mission was to evaluate the proposed POSIX.1 standards, and the resulting Federal Information Processing Standard (FIPS) incorporated POSIX plus additional features the POSIX.1 considered optional or did not specify.

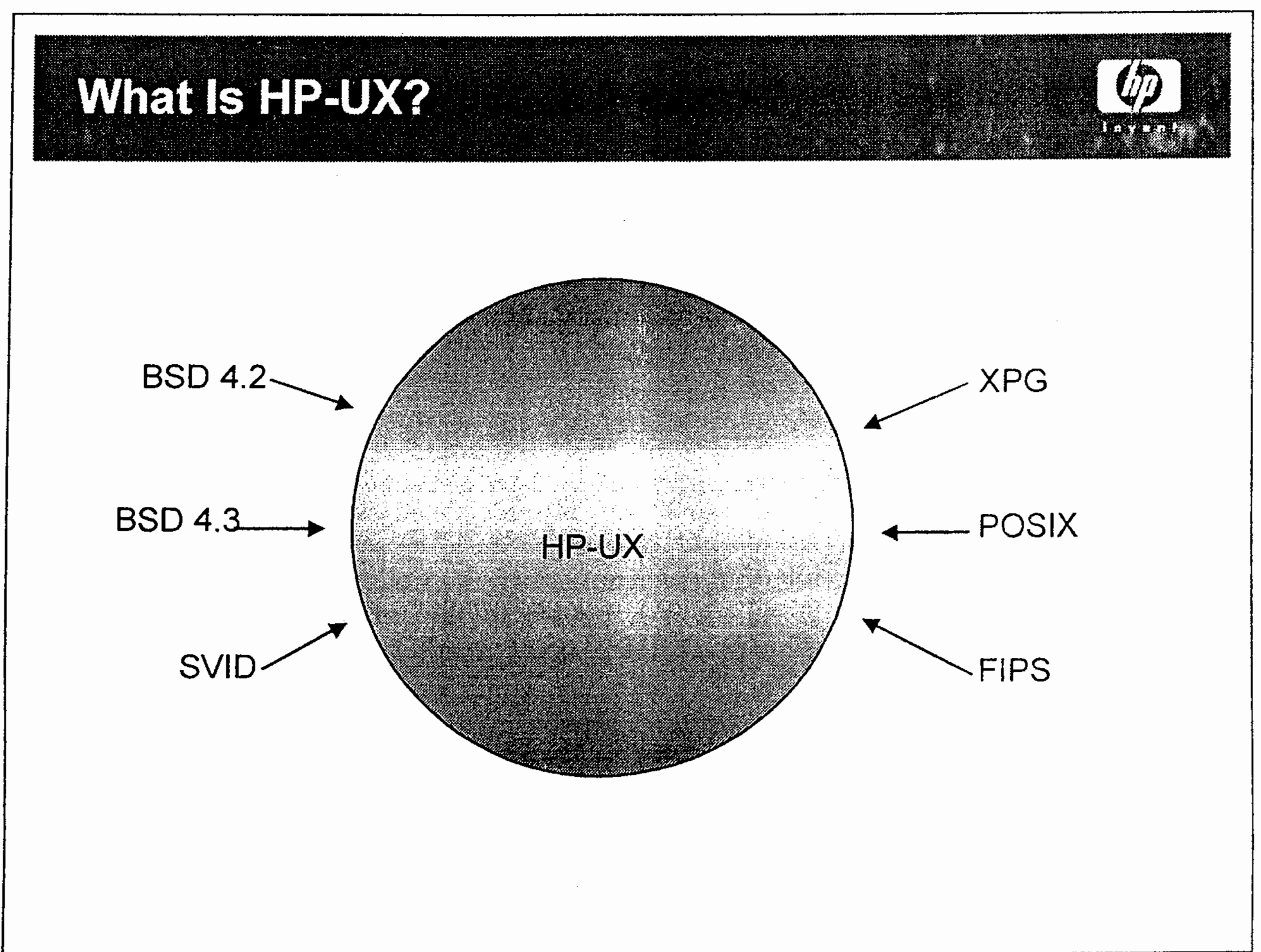
They are also evaluating the other components of POSIX as they are made available.

Linux Standards

A variety of standards bodies have recently been created to oversee the development of the Linux operating system and associated utilities. Details can be found at the following URLs:
www.linux.org
www.gnu.org
www.redhat.com

© 2003 Hewlett-Packard Development Company, L.P.

1–7. SLIDE: What Is HP-UX?



Student Notes

HP-UX 11.0 is an implementation of the AT&T System V UNIX operating system, complying with the following standards:

- POSIX.1:1996 (IEEE Standard 1003.1:1996)
- POSIX.2:1992 (IEEE Standard 1003.2:1992)
- System V Interface Definition (SVID3)
- OSF/Motif 1.2
- X Window System Version 11, Release 4
- X Window System Version 11, Release 5
- X Window System Version 11, Release 6.2
- OSF AES OS Component, Revision A (S300, S400, and HHP 9000 workstations)
- X/Open Portability Guide Issue IV
- FIPS 151-1, FIPS 151-2, and FIPS 189
- ANSI C (ANS X3.159-1989)
- X/OPEN UNIX95 - Branding
- LP64 Industry 64-bit De-facto Standard
- SPEC 1170

© 2003 Hewlett-Packard Development Company, L.P.

The following additional capabilities are also available from Hewlett-Packard:

- X Windows and the Motif graphical user interface
- Common Desktop Environment - Motif-based user interface (CDENext)
- Visual Editor - Motif-based text editor
- Graphics languages
- Native language support
- Menu-based system administration tools (SAM)
- CD ROM-based installation and documentation services
- BIND 4.9
- POSIX.3 (IEEE Standard 1003.3c) Kernel-Based Threads
- Stream-Based transport stack for TCP/IP
- BSD 4.3

The Red Hat Linux 6.x operating system is similar in many respects to HP-UX and other commercial versions of the UNIX operating system. Red Hat has included the following with the workstation and server versions of the Linux package:

- Gnome RPM – Red Hat Package Manager software package management tool
- GNOME Graphical User Environment
- KDE Graphical User Environment
- SAMBA file and print server for Windows 95/98/NT-based systems
- APACHE web server software
- Menu-based system administration tools (linuxconf)

© 2003 Hewlett-Packard Development Company, L.P.

© 2003 Hewlett-Packard Development Company, L.P.

Module 2 — Logging In and General Orientation

Objectives

Upon completion of this module, you will be able to do the following:

- Log in to a UNIX system.
- Log out of a UNIX system.
- Look up commands in the *HP-UX Reference Manual.*
- Look up commands using the online manual.
- Describe the format of the shell's command line.
- Use some simple UNIX system commands for identifying system users.
- Use some simple UNIX system commands for communicating with system users.
- Use some simple UNIX system commands for miscellaneous utilities and output.

© 2003 Hewlett-Packard Development Company, L.P.

2–1. SLIDE: A Typical Terminal Session

A Typical Terminal Session

- Log in to identify yourself and gain access.
- Execute commands to do work.
- Log off to terminate your connection.

Student Notes

To communicate with your computer you will require:

- a terminal with a full American Standard Code for Information Interchange (ASCII) character set a
- data communication line to the computer
- a login ID
- a password

A terminal session begins by logging in through a recognized terminal and ends by logging off. The computer will do work for you in response to the commands that you enter during your terminal session.

The UNIX system identifies the many users on the system by their **user name** (sometimes called the **login ID**). Your login, which is assigned to you by your system administrator, is normally your name or initials. A password may optionally be assigned to your account. Your system administrator may provide you with an initial password that you will be able to change, or you can provide one of your own. Your password is yours alone. You decide what it will be, and *no one* knows or can find out what your password is. If you forget your

© 2003 Hewlett-Packard Development Company, L.P.

password, you will have to ask your system administrator for assistance. Only the system administrator has the authority to delete a user's password from his or her account.

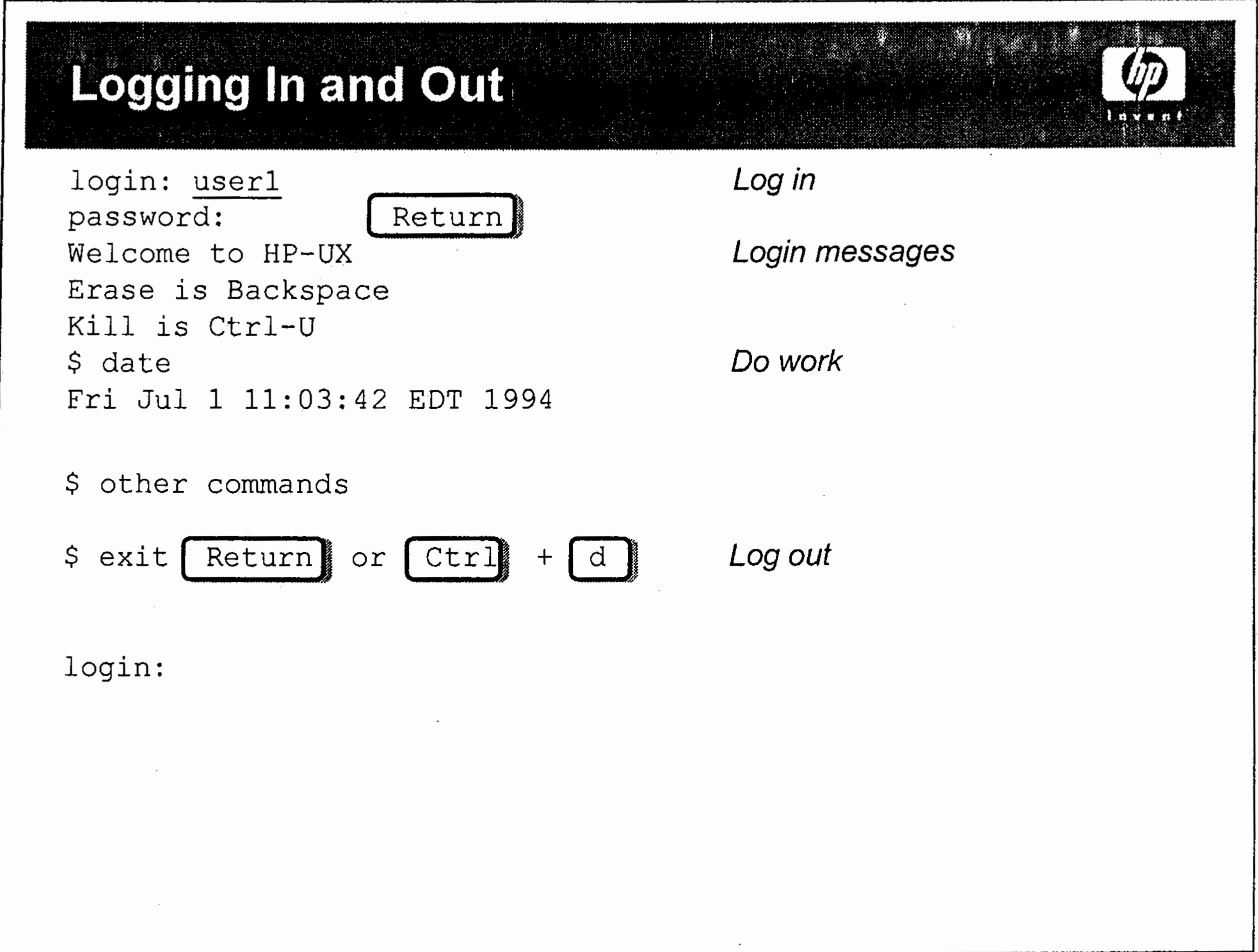
You will enter your user name and password, if required, at the login prompt that will be displayed on your terminal.

Once you are logged in, you can enter commands. The shell will interpret them, and the operating system will execute them on your behalf. Any response generated from the execution of the command will be displayed on your screen.

When you have finished, you terminate your terminal session by logging off. This frees up the terminal so that another user can log in. It is also recommended that you log off when leaving your terminal unattended to prohibit others from accessing your terminal session and user account.

© 2003 Hewlett-Packard Development Company, L.P.

2–2. SLIDE: Logging In and Out



Student Notes

Perform the following steps to log in:

- Turn on the terminal. Some terminals have display timeouts, so you may only have to press a key (Shift for example) to reactivate the display.
- If you do not get the **login:** prompt or if garbage is printed, press Return. If this still doesn't work, press the Break key. The garbage usually means that the computer was trying to communicate with your terminal at the wrong speed. The Break key tells the computer to try another speed. You can press the Break key repeatedly to try different speeds, but wait for a response each time after you try it.
- When the **login:** prompt appears, type your login ID.
- If the **password:** prompt appears, type your password. To ensure security, the password you type will not be printed. For both the login and password, the # key acts as a backspace and the @ key deletes the entire line. Be careful: the keyboard backspace key will not have the deleting function during the login process that it has once you are logged in.

© 2003 Hewlett-Packard Development Company, L.P.

A $ symbol is the standard prompt for the Bourne shell (`/usr/old/bin/sh`), Korn shell (`/usr/bin/ksh`) or POSIX shell (`/usr/bin/sh`) command interpreter. A % symbol usually denotes the C shell (`/usr/bin/csh`). We will be using the POSIX shell, so you will notice a $ prompt. A # prompt is usually reserved for the system administrator's account. This provides a helpful visual reminder while you are logged in as the system administrator, as the administrator can modify (or remove) anything on the system.

Specifying a Password

The first time you log in, your user account may be set up so that you must provide a password. The password that you provide must satisfy the following conditions:

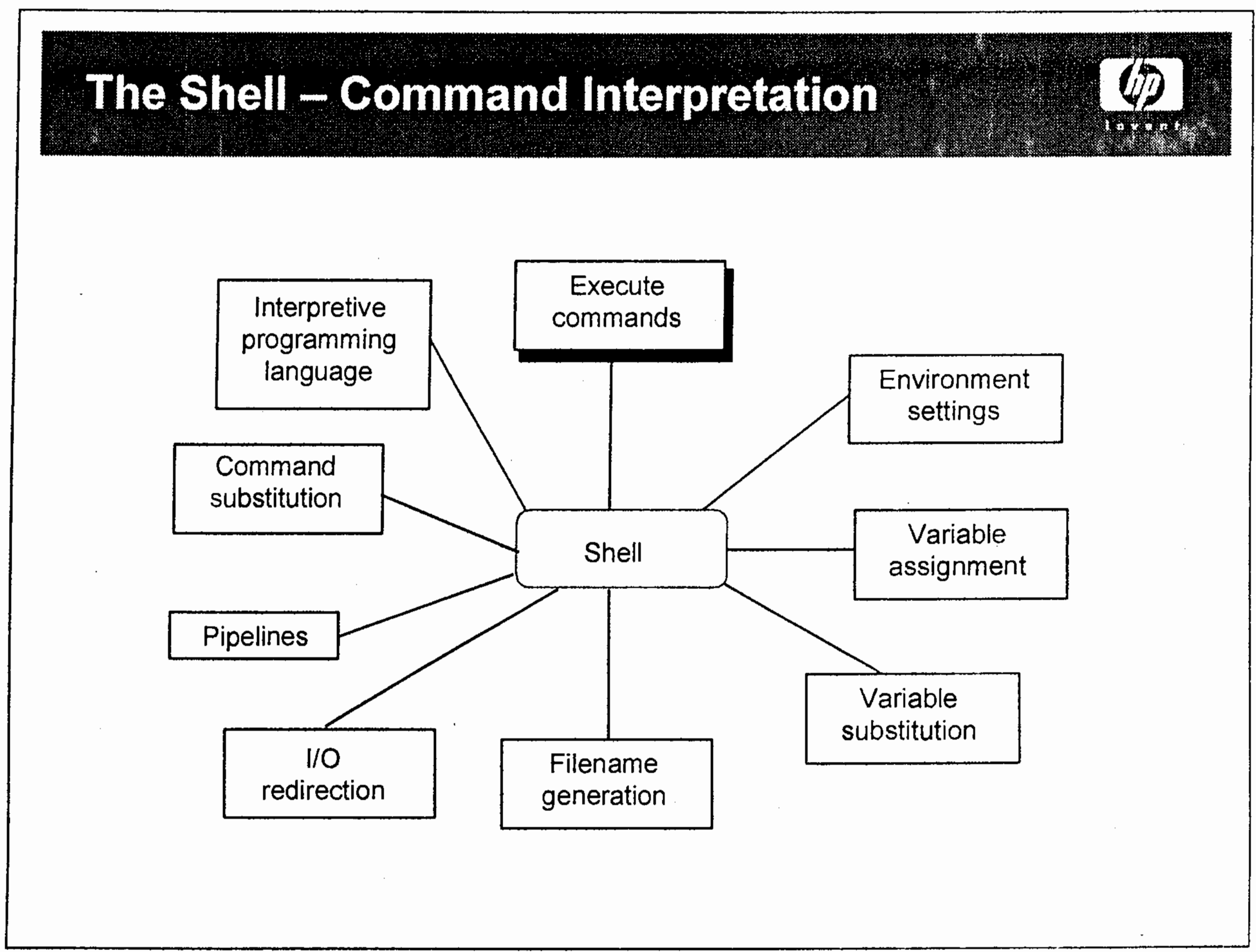
- Your password must have at least six characters.
- At least two of the first six characters must be alphabetic.
- At least one of the first six characters must be non-alphabetic.

After you have entered your password the first time, the system will prompt you to reenter it for verification. Then the system will reissue the log in prompt, and you may complete the login sequence with your new password.

NOTE: When logging in with CDE or HP-VUE, you may have to select (with the mouse) the field in front of login and type in your log name. Then, the field in front of password will be automatically selected if you have a password. So, you have to type in your password that doesn't appear. To correct your log name or password, you can use the [Backspace] key. It is already mapped by the CDE or HP-VUE login process.

© 2003 Hewlett-Packard Development Company, L.P.

2–3. SLIDE: The Shell — Command Interpretation



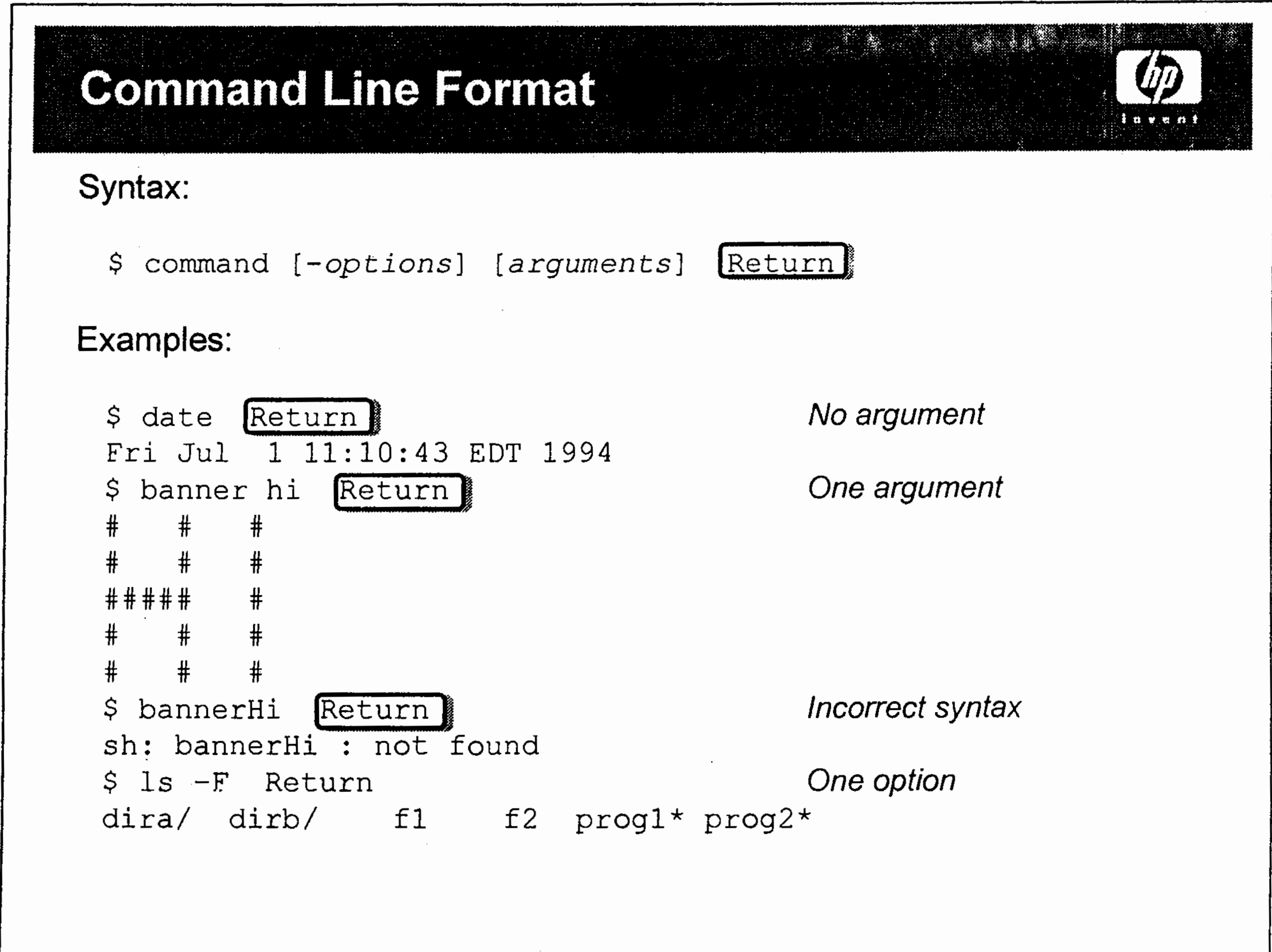
Student Notes

During your login process, a shell is started for you (the POSIX shell in our case). The shell is responsible for issuing the prompt and interpreting the commands that you enter. We will be discussing various commands for the remainder of this module that allow you to access the online reference manual, find out about users who are logged in to your system, and communicate with other users on your system.

As you can see from the slide, the shell has many other functions that supplement command interpretation.

© 2003 Hewlett-Packard Development Company, L.P.

2–4. SLIDE: Command Line Format



Student Notes

After you see the shell prompt (**$**) you can type a command. A recognized command name will always be the first item on the command line. Many commands also accept options for extended functionality, and arguments often represent a text string, a file name, or a directory name that the command should operate upon. Options are usually prefixed with a hyphen (-).

White space is used to delimit (separate) commands, options, and arguments. White space is defined as one or more blanks (Space) or tabs (Tab). Thus, for example, there is a big difference between **`banner Hi`** and **`bannerHi.`** The computer will understand the first one as the command **`banner`** with an argument to the command (**`Hi`**). The second one will be interpreted as a command **`bannerHi`**, which is probably not a valid command name.

Every command will be concluded with a carriage return (Return). This transmits the command to the computer for execution. After this slide, the concluding Return will be understood, and generally will not be presented on the slide.

The terminal input/output supports typing ahead. This allows you to enter a command and then enter the next command(s) before the prompt is returned. The command will be buffered and executed when the current command has finished.

© 2003 Hewlett-Packard Development Company, L.P.

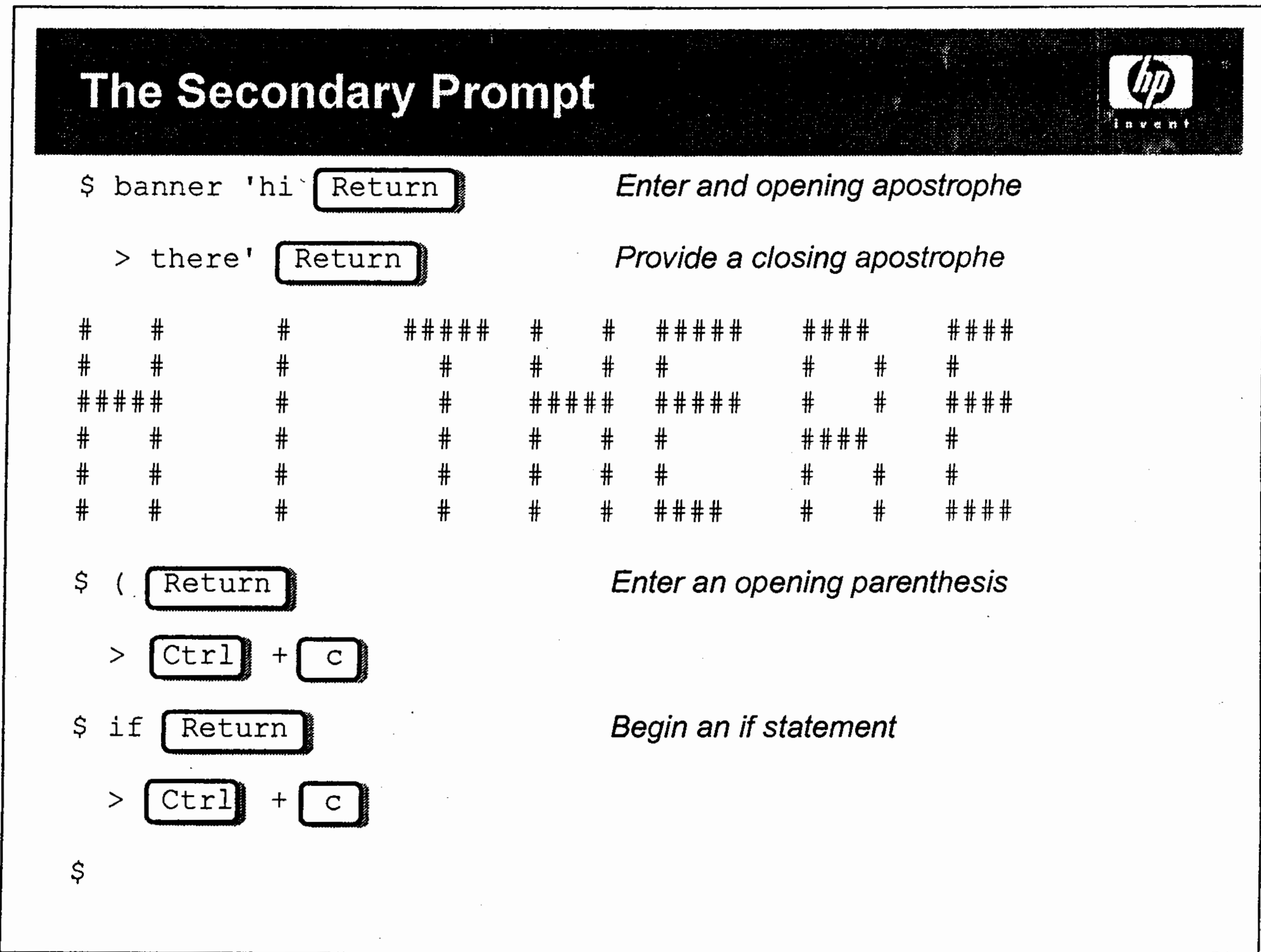
Multiple commands can be entered on one command line by separating them with a semicolon.

NOTE: The UNIX system command input is *case-sensitive*. Most commands and options are defined in lowercase. Therefore, `banner hi` is a legal command whereas `BANNER hi` would not be understood.

NOTE: You can type two commands on a single command line separated by a semicolon (;). For example, `$ ls;pwd`

© 2003 Hewlett-Packard Development Company, L.P.

2–5. SLIDE: The Secondary Prompt



Student Notes

The Bourne, Korn, and POSIX shells support interactive multiline commands. If the shell requires more input to complete the command, the secondary prompt (>) will be issued after you enter the carriage return. Some commands require closing commands, and some characters require a closing character. For example, an opening **`if`** requires **`fi`** to close, opening parentheses require closing parentheses, and likewise an opening apostrophe requires a closing apostrophe.

If you enter a command incorrectly, as illustrated on the slide, the shell will issue you a secondary prompt. A special key sequence should be defined to interrupt the currently executing program. Commonly Ctrl + c will terminate the currently running program and return the shell prompt. You can issue the **`stty -a`** command to confirm the interrupt key sequence for your session.

© 2003 Hewlett-Packard Development Company, L.P.

2–6. SLIDE: The Manual

The Manual

The *HP-UX Reference Manual* contains:

Section	Number and Description
Section 1:	User Commands
Section 1m:	System Maintenance Commands (formerly Section 8)
Section 2:	System Calls
Section 3:	Functions and Function Libraries
Section 4:	File Formats
Section 5:	Miscellaneous Topics
Section 7:	Device (Special) Files
Section 9:	Glossary

Student Notes

"The Manual" is the *HP-UX Reference Manual.* The manual is very useful for looking up command syntax, but was not designed as a tutorial. Also, this was not very useful for learning how to use the UNIX operating system. Experienced UNIX system users refer to the manual for details about commands and their usage. The manual is divided into several sections, as illustrated in the slide.

Following is a brief description of each section:

Section 1 — User Commands
This section describes programs issued directly by users or from shell programs. These are generally executable by any user on the system.

Section 1M — System Maintenance
This section describes commands that are used by the system administrator for system maintenance. These are generally executable only by the user *root*, the login that is associated with the system administrator.

© 2003 Hewlett-Packard Development Company, L.P.

Section 2	System Calls This section describes functions that interface into the UNIX system kernel, including the C-language interface.
Section 3	Functions and Function Libraries This section illustrates functions that are provided on the system in binary format other than the direct system calls. They are usually accessed through C programs. Examples include input and output manipulation and mathematical operations.
Section 4	File Formats This section defines the fields of the system configuration files (such as **`/etc/passwd`**), and documents the structure of various file types (such as **`a.out`**).
Section 5	Miscellaneous Topics This section contains a variety of information such as descriptions of header files, character sets, macro packages, and other topics.
Section 7	Device Special Files This section discusses the characteristics of the special (device) files that provide the link between the UNIX system and the system I/O devices (such as disks, tapes, and printers).
Section 9	Glossary This section defines selected terms used throughout the reference manual.

Within each section, commands are listed in alphabetical order. In order to find a given command, users can reference the manual index.

Linux manual pages are arranged with different section numbers, as shown below:

Section 1	User Commands
Section 2	System Calls
Section 3	Functions and Function Libraries
Section 4	Device Special Files
Section 5	File Formats
Section 7	Miscellaneous Topics
Section 8	System Maintenance
Section 9	*<Not Used>*

© 2003 Hewlett-Packard Development Company, L.P.

2–7. SLIDE: Content of the Manual Pages

Student Notes

It is important to know the format of the manual pages. Throughout the UNIX system documentation, references are given in the format *cmd(n)*, in which *cmd* is the name of the command and *n* is one of the eight sections of the manual. Thus, `date(1)` refers to the date command in section 1 of the manual. In each section, the commands are listed alphabetically. Because of the way the manual is maintained, page numbering is not used. Each command starts on a page 1.

Each manual "page" (some commands take up more than one page) has several major headings. Manual pages do not always have all the headings on them.

The following lists each heading and gives a description of its contents:

NAME	This contains the name of the command and a brief description. The text in this section is used to generate the index.
SYNOPSIS	This indicates how the command is invoked. Items in **boldface** are to be typed at the terminal exactly as shown. Items in square brackets ([ ]) are optional.

© 2003 Hewlett-Packard Development Company, L.P.

Items in regular type are to be replaced with appropriate text that you choose. Ellipses (...) are used to show that the previous argument may be repeated. If in doubt about the meaning of the synopsis, read the DESCRIPTION.

DESCRIPTION	This contains a detailed description of the function of each command and each option.
EXTERNAL INFLUENCES	This provides information on programming for various spoken languages, which is useful for international support.
NETWORKING FEATURES	This lists network-feature-dependent functionality.
RETURN VALUE	This describes values returned on the completion of a program call.
DIAGNOSTICS	This explains error messages that the command may issue.
ERRORS	This lists error conditions and their corresponding error message or return value.
EXAMPLES	This provides examples of the command use.
WARNINGS	This points out potential pitfalls.
DEPENDENCIES	This points out variations in the UNIX system operation that are related to the use of specific hardware.
AUTHOR	This describes the developer of the command.
FILES	This describes any special files that the command uses.
SEE ALSO	This refers to other pages in the manual or other documentation containing additional information.
BUGS	This discusses known bugs and deficiencies and occasionally suggests fixes.
STANDARDS CONFORMANCE	This describes standards to which each entry conforms.

© 2003 Hewlett-Packard Development Company, L.P.

2–8. TEXT PAGE: The Reference Manual — An Example

banner(1) banner(1)

NAME

banner - make posters in large letters

SYNOPSIS

banner strings

DESCRIPTION

banner prints its arguments (each up to 10 characters long) in large letters on the standard output.

Each argument is printed on a separate line. Note that multiple-word arguments must be enclosed in quotes in order to be printed on the same line.

EXAMPLES

Print the message ``Good luck Susan'' in large letters on the screen:

```
banner "Good luck" Susan
```

The words Good luck are displayed on one line, and Susan is displayed on a second line.

WARNINGS

This command is likely to be withdrawn from X/Open standards. Applications using this command might not be portable to other vendors' platforms.

SEE ALSO

echo(1).

STANDARDS CONFORMANCE

banner: SVID2, SVID3, XPG2, XPG3

© 2003 Hewlett-Packard Development Company, L.P.

2–9. SLIDE: The Online Manual

The Online Manual

Syntax:

```
man [-k | X ]  keyword | command
```

in which *X* is the number of one of the manual sections

Examples:

`$ man date`	*Display the "date" man page.*
`$ man -k copy`	*Display entries with keyword "copy".*
`$ man passwd`	*Display the "passwd" man page-Section 1.*
`$ man 4 passwd`	*Display the "passwd" man page-Section 4.*

Use `Space` to view next page.

Use `Return` to view next line.

Use `q` to quit the man command.

Student Notes

There is another way of retrieving information from the manual.

On most UNIX systems, the manual is available online. The online manual is accessed using the **man** command.

The syntax is:

```
man -k keyword
```

or

```
man [12345791m] command
```

In which

man -k *keyword*	This lists all commands that have the string keyword in their description.

© 2003 Hewlett-Packard Development Company, L.P.

man [12345791m] *command*	This displays the manual page for *command* in the specified section of the manual.
man *command*	This displays the default manual entry for command. There may be an entry in more than one section for the command.

All of these commands require that the system administrator has installed the online manual correctly. In the first example of the slide, **man passwd** shows the command to change the password. **man 4 passwd** is password file layout. When the first page of the manual entry for the specified command has been displayed, the following keys can be used at the **Standard input** prompt:

[Return]	Displays the next line
[Space]	Displays the next page
[Q] or [q]	Exits the man command and returns to the shell

Occasionally, when accessing the online manual, you will get the message:

```
Reformatting Entry.  Wait...
```

This message means the manual page for the specified command needs to be uncompressed because it is being used for the first time during the current session. The message will not appear the next time that the command is referenced.

Screen Control

Special keys are available on Hewlett-Packard keyboards to assist you in viewing the output of your man command, or any other command. NOTE: these keys are a function of the HP keyboard and terminal emulator products and not a feature of the UNIX system.

[Prev]	Scroll the display back to the previous screen.
[Next]	Scroll the display forward to the next screen.
[Shift] +[↓]	Scroll down line by line.
[Shift] +[↑]	Scroll up line by line.
[Home]	Move the cursor to the first row, first column.
[Clear Display]	Clear the display from the cursor to the end of the screen.
[Home] [Clear Display]	Clear the entire display.

© 2003 Hewlett-Packard Development Company, L.P.

Multiple Manual Entries

Some commands have an entry in more than one section of the reference manual. You can use the **whereis** command to display the sections that provide a manual reference. For example:

```
$ whereis passwd

passwd: /sbin/passwd /usr/bin/passwd /usr/share/man/man1.Z/passwd.1

/usr/share/man/man4.Z/passwd.$


whereis nothere

nothere:
```

This reports that there is a manual entry for the **passwd** command in sections 1 and 4, and there is no manual entry for a command called **nothere**.

© 2003 Hewlett-Packard Development Company, L.P.

2–10. SLIDE: Some Beginning Commands

`id`	Display your user and group identifications.
`who`	Identify other users logged on to the system.
`date`	Display the system time and date.
`passwd`	Assign a password to your user account.
`echo`	Display simple messages to your screen.
`banner`	Display arguments in large letters.
`clear`	Clears terminal screen.
`write`	Sends messages to another user's terminal.
`mesg`	Allows/denies messages to your terminal.
`news`	Display the system news.

Student Notes

We will present some basic commands that allow you to practice submitting simple commands to the UNIX system shell. Most of the commands presented have many options in addition to those presented in the student workbook. Refer to the `man` pages for these commands if you would like to investigate other options.

© 2003 Hewlett-Packard Development Company, L.P.

2–11. SLIDE: The `id` Command

The id Command

Syntax:

`id` Displays user and group identification for session

Example:

```
$ id
uid =303 (user3) gid=300 (class)
```

Note: The '**gid**' is the primary group. If the user belongs to additional groups, these are listed at the end as 'groups'

Student Notes

In order to access files and execute programs, the UNIX system must know your **user** and **group** identifications. The computer maintains numerical identifiers; corresponding text names are provided for the user's convenience. Your identification will be defined initially when you log in. After you are logged in, you may have authorization to change your user and/or your group identifiers. The **id** command will display your current user and group identifiers.

All of the user identifications recognized by the computer are stored in the file **/etc/passwd**, while all of the group identifications are stored in the file **/etc/group**.

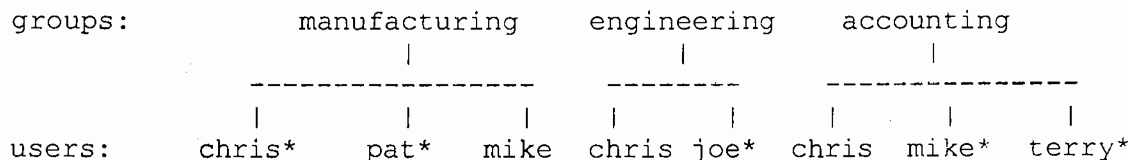
Groups

Groups provide a method for a subset of users to share access to a file.

Users to be included in a specific group are defined by the system administrator, and each user can be a member of one or more groups. Groups are normally formed using the normal work groups already defined in an organization.

© 2003 Hewlett-Packard Development Company, L.P.

For example, an organization may include manufacturing, engineering, and accounting groups. The user structure within these groups may be defined as follows:



* Denotes the group identification at login

- **chris** is a member of all three groups.
- **mike** is a member of two groups.

With this organization, **chris** could access the files that are associated with the manufacturing, engineering, and accounting groups. **mike** could access files that are associated with the **manufacturing** and **accounting** groups. All other users can access only the files associated with their login group.

© 2003 Hewlett-Packard Development Company, L.P.

2–12. SLIDE: The who Command

The who Command

Syntax:

`who [am i]`	Reports information about users who are
`whoami`	currently logged on to a system

Examples:

```
$ who
root     tty1p5  Jul 01 08:01
user1    tty1p4  Jul 01 09:59
user2    tty0p3  Jul 01 10:01

$ who am i
user2    tty0p3  Jul 01 10:01

$ whoami
user2
```

Student Notes

The **who** command reports which users are logged into a system, what terminal port each is connected to, and login time information. **whoami** just reports the user name and port information of the local terminal session. Finally, the **whoami** command reports the user name that the system associates with the local terminal port. Authorization to execute a command is dependent upon a user's identification, and a user may be able to change his or her user identification interactively to access additional commands or programs.

In the Linux environment, the **whoami** command also displays the hostname of the computer system, followed by an exclamation character (**!**), before the user name at the start of the output line.

© 2003 Hewlett-Packard Development Company, L.P.

2–13. SLIDE: The `date` Command

The date Command

Syntax:

`date` Reports the date and time

Example:

```
$ date
Fri Jul  1 11:15:55 EDT 1998
```

Student Notes

The **date** command is used to report the system date and time. It accepts arguments that allow the output to be formatted.

The **date** command is usually used with no options or arguments, and that is how we present it here.

The manual page — see **date(1)** also shows a first argument that can be used to set the date. *Only the system administrator is authorized to modify the system time and date.*

© 2003 Hewlett-Packard Development Company, L.P.

2–14. SLIDE: The passwd Command

Syntax:

```
passwd        Assigns a login password
```

Example:

```
$ passwd
Changing password for user3
Old password:
New password:
Re-enter new password:
```

Password Restrictions:

- minimum of six characters
- at least two alpha characters
- at least one non-alpha character

Student Notes

On many systems, the system administrator controls the users' passwords. Under the UNIX system however, the system administrator can allow users to retain direct control of their own password. The **passwd** command can be used to change your password. The syntax is

```
passwd
```

You will be asked for your current password (old password). This is to prevent someone from changing your password if you leave your terminal unattended while you are logged in. Then you will be asked for your new password, and you will be asked to confirm it by retyping the new password. This is to prevent you from changing your password to one that has a typographical error in it. Your new and old passwords must differ by at least three characters.

The characters of the old and new passwords will not be displayed to the screen as you type them in.

© 2003 Hewlett-Packard Development Company, L.P.

Password Restrictions

Your password must have at least six characters. At least two of the first six characters must be alphabetic and at least one of the first six characters must be nonalphabetic.

The system administrator is not held to these conditions, so if the system administrator assigns a password to your account, it may not follow these rules.

© 2003 Hewlett-Packard Development Company, L.P.

2–15. SLIDE: The echo Command

The echo Command

Syntax:

```
echo [arg ...]
```

Writes argument(s) to the terminal

Examples:

```
$ echo how are you
how are you

$ echo    123       abc
123 abc
```

Student Notes

The **echo** command gives you the ability to display command-line arguments, that is, a command such as:

```
echo hello
```

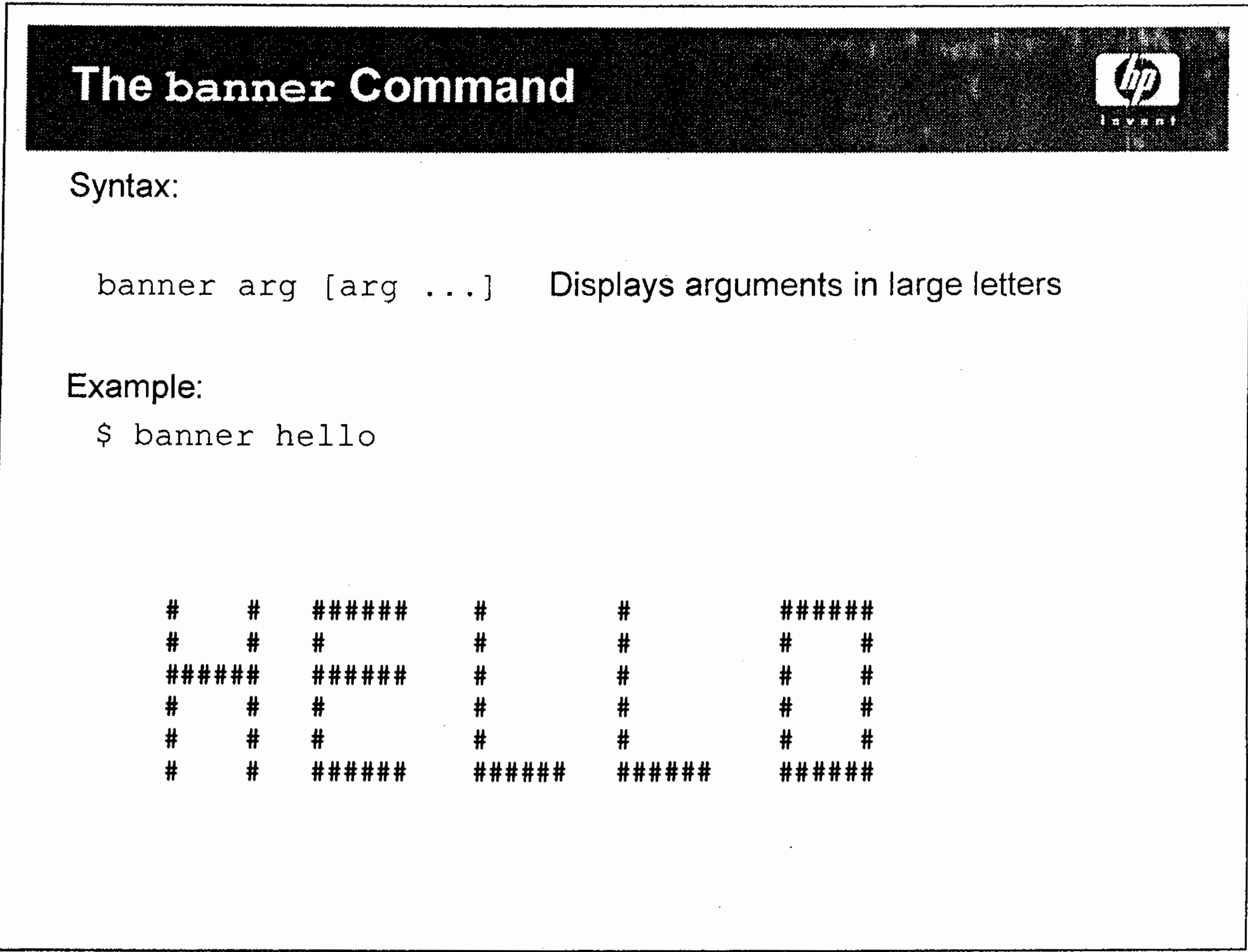
Produces the output:

```
hello
```

This command may seem rather trivial, but it is commonly used in shell programs to display messages to users or see the value of a shell variable. For shell programming we will use the **echo** command extensively.

© 2003 Hewlett-Packard Development Company, L.P.

2–16. SLIDE: The banner Command



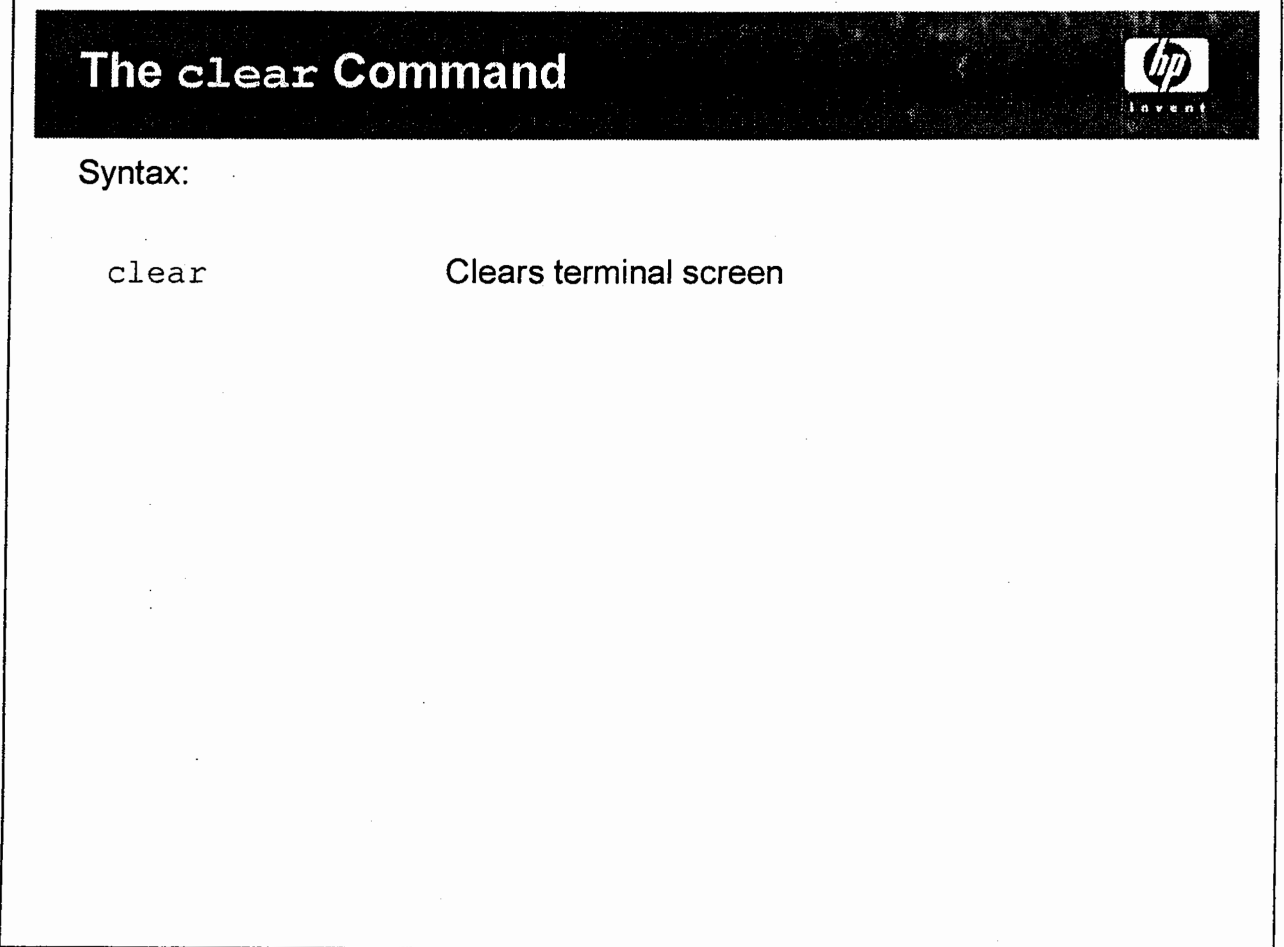
Student Notes

The **banner** command was originally developed, and is still used, for labeling the output from line printers. The **banner** command displays the command line arguments in large capital letters, one argument per line.

The Linux **banner** command is intended to be used for printed output, only. By default, a user would not be able to execute the **banner** command on a Linux system unless the command is invoked using the appropriate file pathname (for example, **/usr/games/banner**).

© 2003 Hewlett-Packard Development Company, L.P.

2–17. SLIDE: The `clear` Command

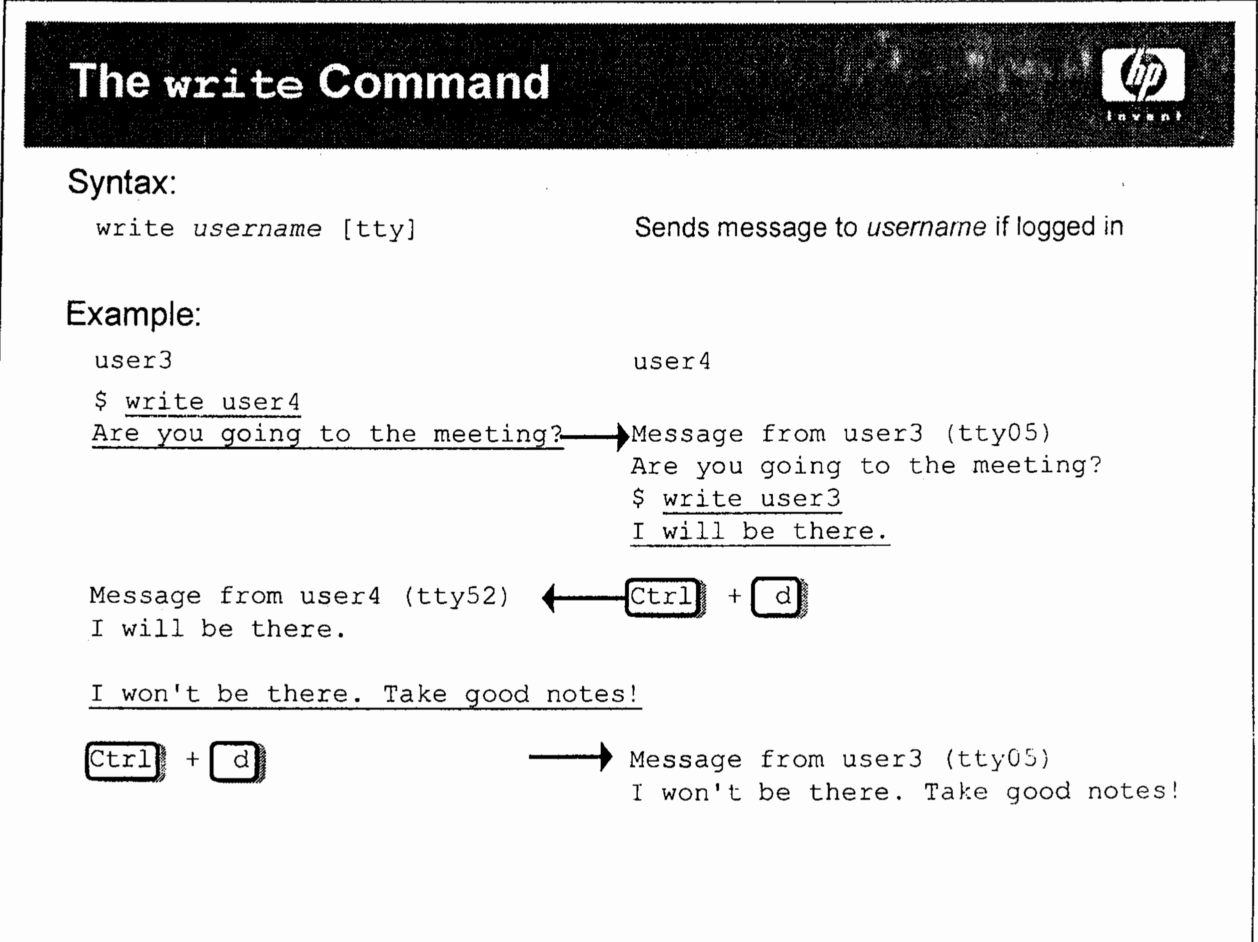


Student Notes

The **`clear`** command clears the terminal screen if it is possible to do so. This command only clears the current screen, so it is possible for the user to scroll up to retrieve previous screens. To erase all screens, position the cursor home, by pressing the [HOME] key, and then type the **`clear`** command.

© 2003 Hewlett-Packard Development Company, L.P.

2–18. SLIDE: The `write` Command



Student Notes

The **`write`** command can be used to send a message to another user's terminal that is currently logged in to the same UNIX computer system. When invoked, the write command gives you the opportunity to input your message. Every time you press [Return], that line is transmitted to the recipient's terminal. The recipient can **`write`** back to you, and you can hold an interactive conversation through your terminals. When you are done typing your message, press [CTRL] + [d]. This will conclude your end of the conversation.

NOTE: Unless you disable the capability, messages can be sent to your terminal *at any time*. Therefore, if you are in a utility such as **`man`**, **`mail`**, or an editor, and someone writes a message to you, it will be displayed on your terminal, and can be disruptive.

If the person to whom you wish to write is not logged on, you will get the message: ***`user`*** **`is not logged on`**, in which ***`user`*** is the user name of the person you tried to reach.

© 2003 Hewlett-Packard Development Company, L.P.

2–19. SLIDE: The mesg Command

Student Notes

You can use the **mesg** command to disable other users from sending messages to your terminal. If you write to someone who has disabled messaging to his or her terminal, you will get a **Permission Denied** error.

mesg n	Denies "writes" to your terminal. This is the default value in HP-UX 10.0 and HP-UX 11.00.
mesg y	Allows "writes" to your terminal.
mesg	Reports whether "writes" are allowed (**y**) or disallowed (**n**).

Even when you disable messaging, the system administrator can still send messages to your terminal.

© 2003 Hewlett-Packard Development Company, L.P.

2–20. SLIDE: The news Command

The news Command

Syntax:

`news [-a] [-n] [headline]` Displays the system news

Example:

`$ news`	*Displays new news*
`$ news -a`	*Displays all news*
`$ news -n`	*Displays new headlines*

Student Notes

Messages that are of interest to all users on the system can be broadcast through the **news** facility. The system administrator often uses the **news** to inform the user community of system-specific items, such as system shutdown times, backup times, or new hardware that will be available.

You can read the **news** by issuing the news command. When issued with no options, only the items that you have not yet read will be displayed.

The options to **news** are:

-a	Reads all the news there is regardless of whether it has been read
-n	Displays only the headlines of unread news items

Each user who accesses the **news** utility will have a **.news_time** file in their *HOME* directory. Every file in the UNIX system has a time stamp, which records the last time the file was modified. The time stamp on the **.news_time** file is updated to match the time stamp of the last news message that you read. If a new message is added to the news facility,

© 2003 Hewlett-Packard Development Company, L.P.

news knows that it has not been read because the time stamp on your `.news_time` file is earlier than the new news message.

There is no equivalent to the **news** command in the Linux environment.

© 2003 Hewlett-Packard Development Company, L.P.

2–21. LAB: General Orientation

Directions

Complete the following exercises and answer the associated questions. You may need to use the *HP-UX Reference Manual* in order to complete some of the exercises.

1. Log in to the system using the user name and password that the instructor assigned to you. Did you have any trouble?

2. Now log out of the system using [CTRL]+[d] or **exit**. What did you notice, if anything? Log back into the system.

3. Which of the following commands are syntactically correct? Try typing them in to see what the output or resulting error message would be.

```
$ echo
$ echo hello
$ echohello
$ echo HELLO WORLD
```

HP-UX systems only
Try the following commands to see what the output or resulting message would be:

```
$ banner
$ banner hello
$ BANNER hello
```

4. Assign a password to your account, or change the password, if one is already defined. Remember the requirements for user passwords.

© 2003 Hewlett-Packard Development Company, L.P.

5. Using variations of the `who` command or the `whoami` command, determine each of the following with separate command lines. What commands did you use?

   Who is on the system?

   What terminal device are you logged in on?

   Who does the system think you are?

6. Can another user send messages to your terminal? What command did you use to find out?

7. Determine if your partner is logged in, and then write a message to your partner's terminal. Establish a two-way conversation. Have fun.

   What happens if you try to write to your partner and he or she is not logged in? What happens if your partner has disabled messaging to his or her terminal?

8. Read the system's news. What command did you use? Can you display the news *after* you have read a message? (Note that this works only in HPUX.)

9. Execute the **`date`** command with the proper arguments so that its output is in a *mm-dd-yy* format. Hint: look at the examples provided in the reference manual entry for **`date(1)`**.

© 2003 Hewlett-Packard Development Company, L.P.

10. Using the online `man` pages, find the `cp` command. What is its function? What is the minimum number of arguments that it requires?

11. Using the online `man` pages, find the `ls` command. What is its function? What is the minimum number of arguments that it requires?

12. Issue the command `ls -l /usr/bin`. You will see several screens worth of data scroll by.

    HP-UX Only

    Use the up arrow key to move the cursor up to the top of the screen. Issue the `clear` command. Is there any data remaining on the screen? Using the [Shift] key together with the down arrow screen, scroll down. Do you see a partial listing of the `ls -l` command?

© 2003 Hewlett-Packard Development Company, L.P.

Module 3 — Using CDE

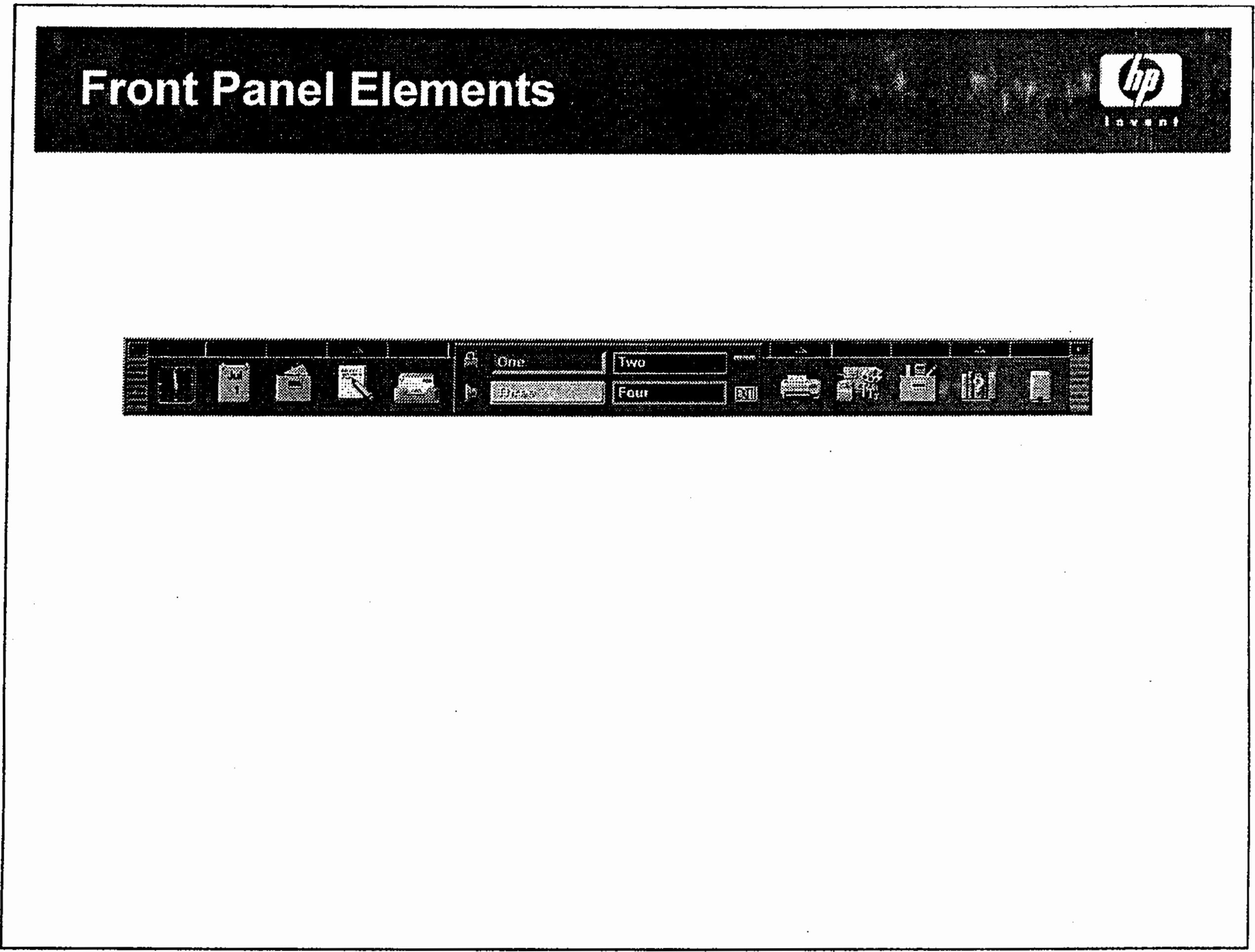
Objectives

Upon completion of this module you will be able to:

- Describe the Front Panel Elements.
- Understand how the Front Panel Pop-Up Menus work.
- Describe the Workspace Switch.
- Describe the Subpanel Controls.
- Understand how to use the Help System.
- Describe the File Manager.
- Understand how to use the File Manager Menu.
- Locate files using the File Manager.
- Delete files.
- Print files using the Front Panel, the File Manager, and the Print Manager.
- Display Print Spooler Information.
- Understand Printer Management.
- Use the Text Editor.
- Run Applications using the Application Manager.
- Use the Mailer and the Mailer Options, as well as how to create Mailboxes.
- Use the Calendar Manager to Schedule Appointments and To Do Items.
- Describe how to Browse Other Calendars on the Network.
- Describe how to Grant or Prevent Access to Your Calendar.

© 2003 Hewlett-Packard Development Company, L.P.

3–1. SLIDE: Front Panel Elements



Student Notes

The Front Panel

The Front Panel is a special window at the bottom of the display of each workspace, which contains a collection of frequently, used controls, indicators, and subpanels from which users can manage all aspects of a session (except initial login).

Many controls in the Front Panel, like the Mail Utility, start applications when you click on them. Others, like the clock, are merely indicators and do not respond when you try to activate them by clicking. Depending upon the actions that the applications perform, they may or may not be used as a drop zone. For instance, the Mailer, Print Manager, and Trash Can can all be used to drop files from the File Manager.

Arrow buttons over the Front Panel controls identify subpanels - click the arrow and a subpanel menu appears.

Main components of the Front Panel include:

Clock	Displays the current time of day based on the system time.

© 2003 Hewlett-Packard Development Company, L.P.

Calendar	Displays the current date. This icon activates the Calendar application, which allows users to manage, schedule, and view appointments. This utility also provides the ability to view other calendars across the network and schedule appointments across the network, if access is permitted.
File Manager	Displays the files, folders, and applications on the system as icons. Users can work with files without having to learn complex commands. Activities such as copying, moving, deleting, printing, and changing permissions on a file, to name just a few, can easily be done with the File Manager menu bar.
Text Editor	Starts a simple text editor with common functionality including clipboard interaction with other applications. This application can also be used as a drop zone from the File Manager. The default Front Panel displays the Text Editor icon. It also has a Personal Application subpanel that can activate a terminal or the icon editor.
Mailer	Activates a GUI interface to the electronic mail facility. This tool can be used as a drop zone for files or calendars to be mailed to others on the network.
Lock Button	Allows users to lock the screen if unattended. This can be configured to be automatic after a certain time period has elapsed. The user password is needed to regain access to the Desktop.
Workspaces	By default, allows four separate screens of windows, however the number of workspaces is configurable. Applications can be organized into a specific, custom named workspace. In addition, windows present in one workspace can be copied to another workspace.
Exit	Allows users to log out of the Desktop. All work not saved will be lost. By default, users will be prompted to confirm logout.
Print Manager	A simple GUI print job manager that allows the scheduling and management of print jobs on any available printer.
Style Manager	Allows users to easily customize the desktop resources such as colors, backdrops, font size, and system behavior.
Application Manager	Provides access to applications in icon form. Users can click on a specific application icon to execute the application. Application Manager is comprised of Application Groups, which is a way of organizing applications according to specific functions. Users have the ability to create their own Application Group and to put their own applications in new or existing Application Groups.
Help Manager	Online help is available for each of the standard applications in CDE. The pop-up sub-menu provides access to: Help Manager, which is a special help volume that lists all the online help registered on the system; Desktop Introduction, which helps users understand how to navigate around the Desktop; Front Panel Help, which provides help

© 2003 Hewlett-Packard Development Company, L.P.

on the contents of the front panel; and On Item Help, which is interactive, allowing users to move the pointer to a specific item and click the item to display the corresponding help. In addition, other applications installed on the desktop may take advantage of using the Desktop's Help System.

Trash Can — Collects the files and folders that users delete. They are not actually removed from the system until the trash is emptied. Until that time users can restore files that have been deleted. This control can be used as a drop zone.

CDE Reference Manuals

- B1171-90101 *CDE 1.0 User's Guide*

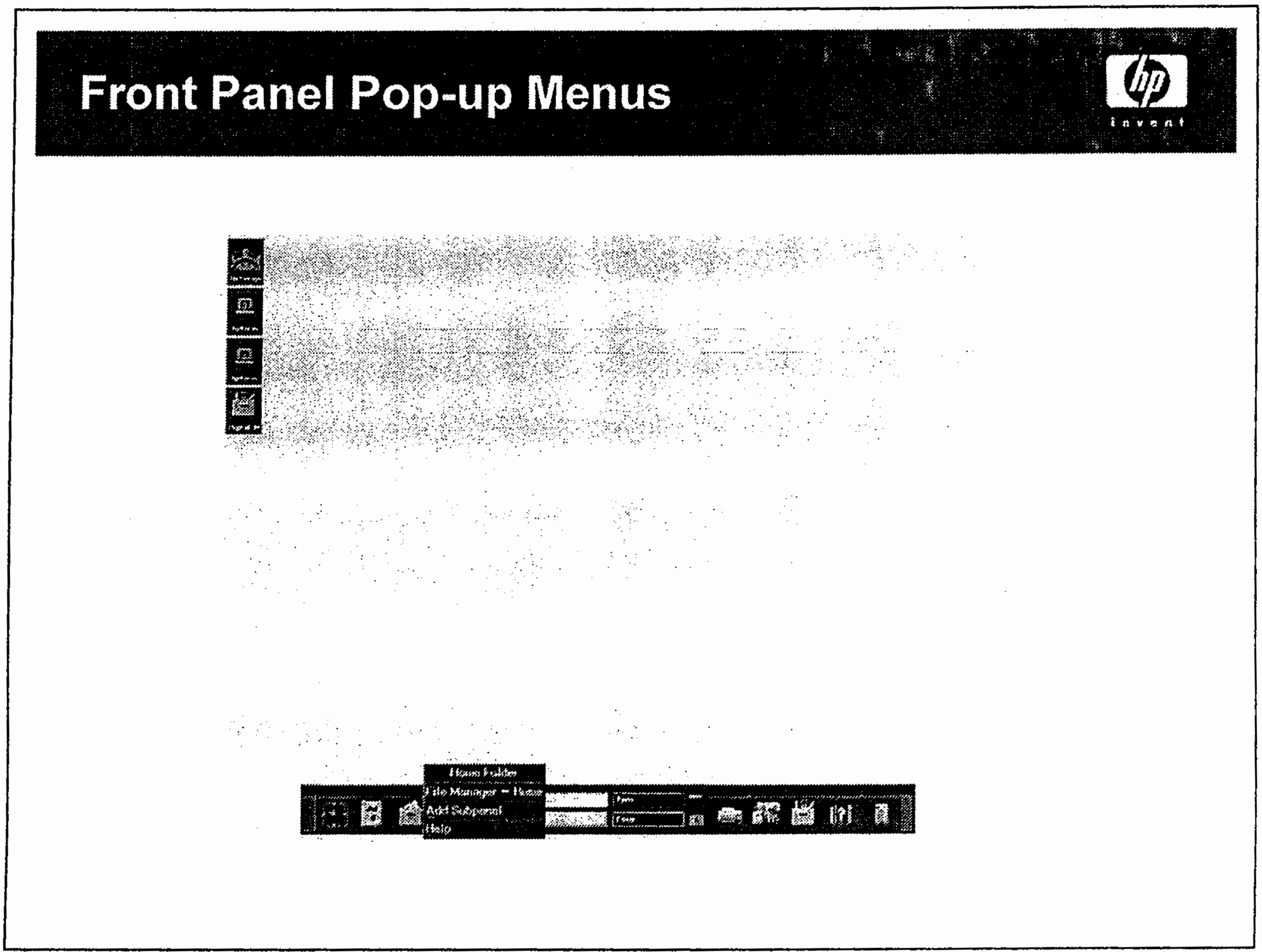
Linux Systems

The Red Hat Linux system is supplied with both the Gnome and KDE graphical user environments. KDE has many similarities to CDE. It is intended to be used primarily by Linux end-users. However, the differences between the CDE and KDE environments are too many to be mentioned in these notes. The "look and feel" of the KDE environment is sufficiently similar to CDE such that an end-user should have little difficulty becoming familiar with the various applications and utilities provided with KDE after completing this course module.

For the purposes of this course, only the CDE environment will be covered in any detail.

© 2003 Hewlett-Packard Development Company, L.P.

3–2. SLIDE: Front Panel Pop-Up Menus



Student Notes

A pop-up menu is one that "pops up" when you click on mouse button 3 in an application window or on a workspace object. Each control in the Front Panel has a pop-up menu, which is different for each control. To display a Front Panel pop-up menu, point to the control and press down mouse button 3.

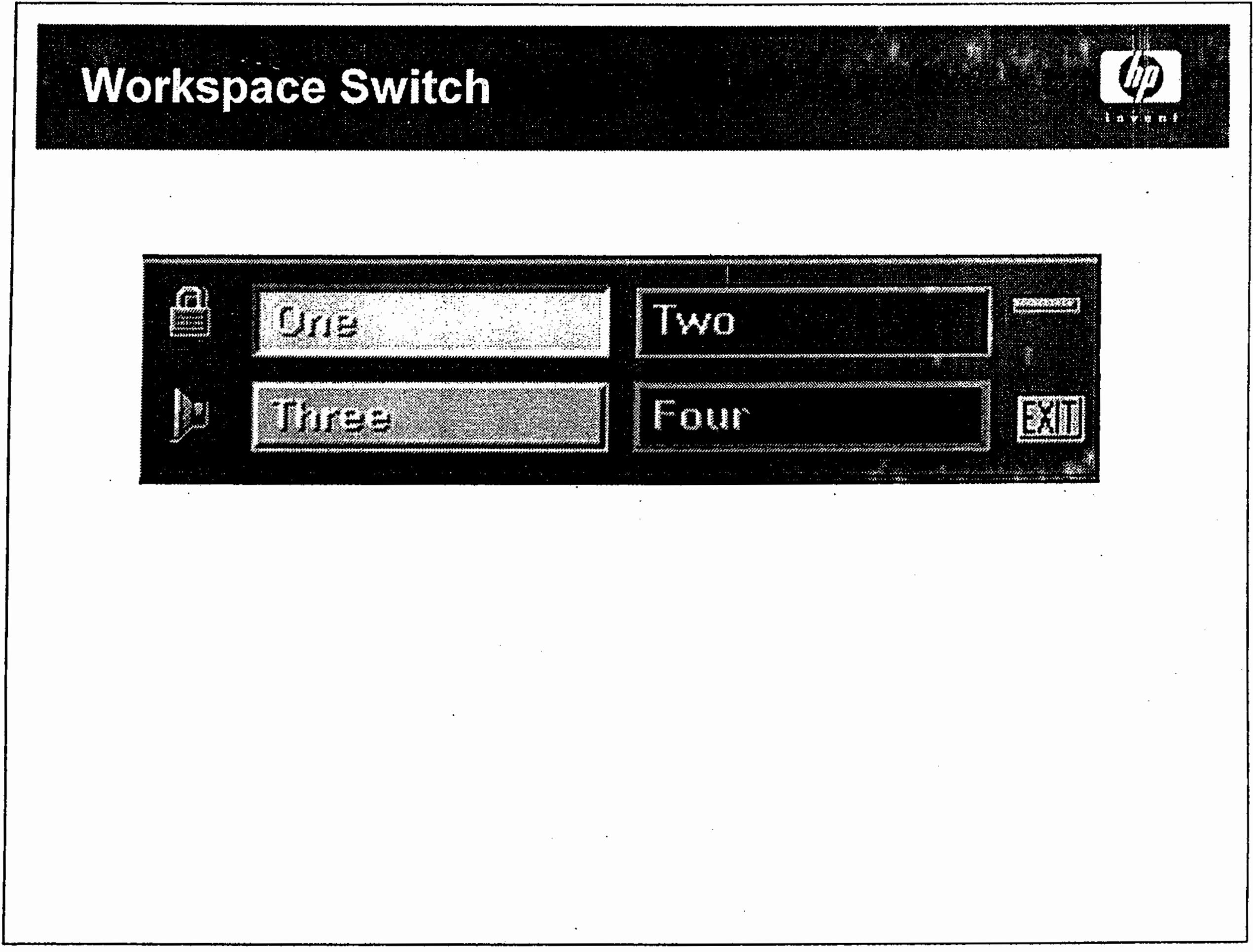
Depending upon the control, pop-up menu contents will vary. For example, if the control is an application, the first entry in the menu is the command to start that application. Figure 4-1 shows the pop-up menu for Application Manager. If the object is not an application, a different set of choices will be available depending upon the purpose of the object.

In addition to the Main Panel, subpanel elements also have pop-up menus. For example, if you click on the arrow above the Text Editor control, the Personal Applications subpanel will be displayed. Position the cursor next to the Terminal, and press mouse button 3. You will get a pop-up menu which will allow you to copy the control to the Main Panel, delete the control altogether, or get help.

Pop-up menus are also available within applications. For example within File Manager, action can be taken upon the displayed objects (files or directories, for example) by pointing the cursor to the object, and pressing mouse button 3.

© 2003 Hewlett-Packard Development Company, L.P.

3–3. SLIDE: Workspace Switch



Student Notes

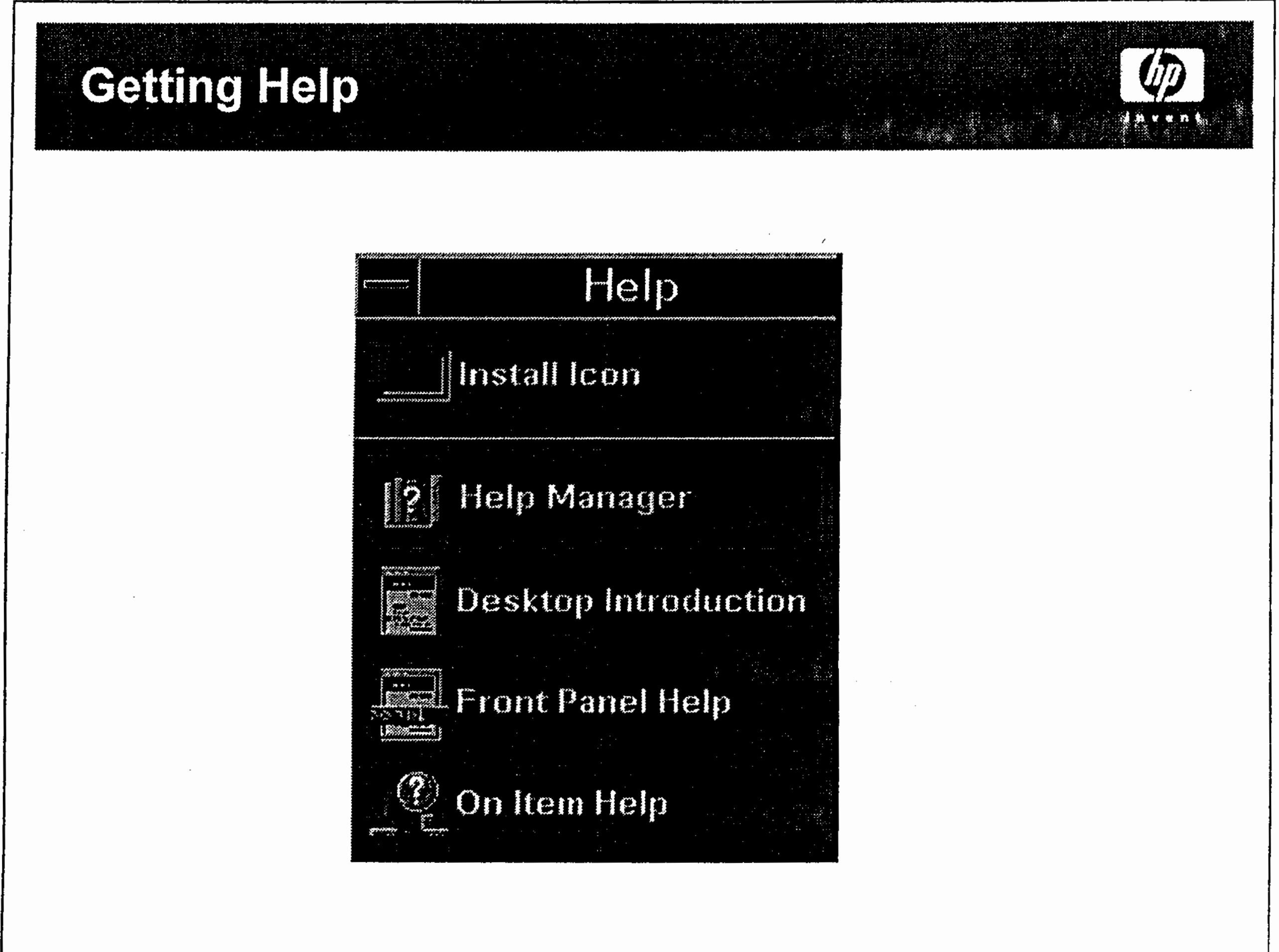
By default there are four workspaces. Each workspace covers the entire display. The workspace switch contains the buttons used to change from one workspace to another. The button that represents the current workspace will appear to be pressed in. To switch to another workspace, simply click on the button that refers to the new workspace. Work done in one workspace is preserved when switching to another workspace.

A pop-up menu is available for the workspace switch by clicking on a portion of the workspace switch that is not occupied by other controls or workspace buttons.

In addition, each workspace button has a pop-up menu that can be used add another workspace, or rename or delete the workspace being pointed to.

© 2003 Hewlett-Packard Development Company, L.P.

3–4. SLIDE: Getting Help



Student Notes

Online help is available for each standard application within CDE. The menu bar of each application has a Help menu selection on the far right side, or users can press F1 in most applications to get context-sensitive help. In addition, on the Front Panel, there is a Help Manager control, which is a special volume that lists all the online help registered on the system. By choosing any underlined titles you can view an additional layer of help for that subject.

The Help System subpanel also gives access to help on the Desktop and the Front Panel, as well as **On Item** help which enables you to move the pointer to a specific item and click to display a corresponding help page.

Help Windows

The Help System that is built into each of the CDE applications provides two types of help windows, general help and quick help. Quick help has just a topic display area which displays the help of a requested subject. General help windows have two areas: topic tree and topic display area. The topic tree is basically an outline of a help volume's major topics. Subtopics are displayed beneath main topics. You can choose a topic by clicking on the topic within the topic tree.

© 2003 Hewlett-Packard Development Company, L.P.

To open the general help window, activate a CDE application (for example File Manager). In the upper right corner of the menu bar, you will see the **`Help`** menu selection. Click on **`Help`**, followed by **`Overview`** to open a general help window. Above the Help text, and below the menu bar, you will see the topic tree for the File Manager Help System.

Another way to jump between help subjects is by using hyperlinks. When a topic display has a word underlined, for example, the help facility will "jump" to that related topic when a user clicks on the underlined word.

Searching for Topics Using the Help Index

Once users have opened a general help window, they can search based on key words or pattern searches by using the Help Index as follows:

1. To open the index within an application, click on the [Index] button. The index allows you to browse all the entries for the current help volume, all help volumes, or just selected help volumes.

2. Select the **`Entries with`** field, type the word or phrase you are looking for and press [Return].

3. Select the index entry you want to view. If the entry has a + prefix, the list will expand to show additional choices. Select a help topic to view.

Pattern Searches

The help facility recognizes the following pattern search characters when searching for topics:

* (asterisk)	Matches any string of characters (including no characters)
? (question mark)	Matches any single character
. (period)	Matches any character
| (vertical bar)	Specifies two search patterns and matches either pattern (logical OR)
() (parentheses)	Encloses a pattern expression

Displaying a Man Page

Displaying a man page is done from the Application Manager, rather than the Help System.

1. Click the Application Manager control in the Front Panel.

2. Double-click the Desktop_Apps icon.

3. Click the Man Page Viewer icon.

© 2003 Hewlett-Packard Development Company, L.P.

4. Type the name of the man page you want to see and press Return.

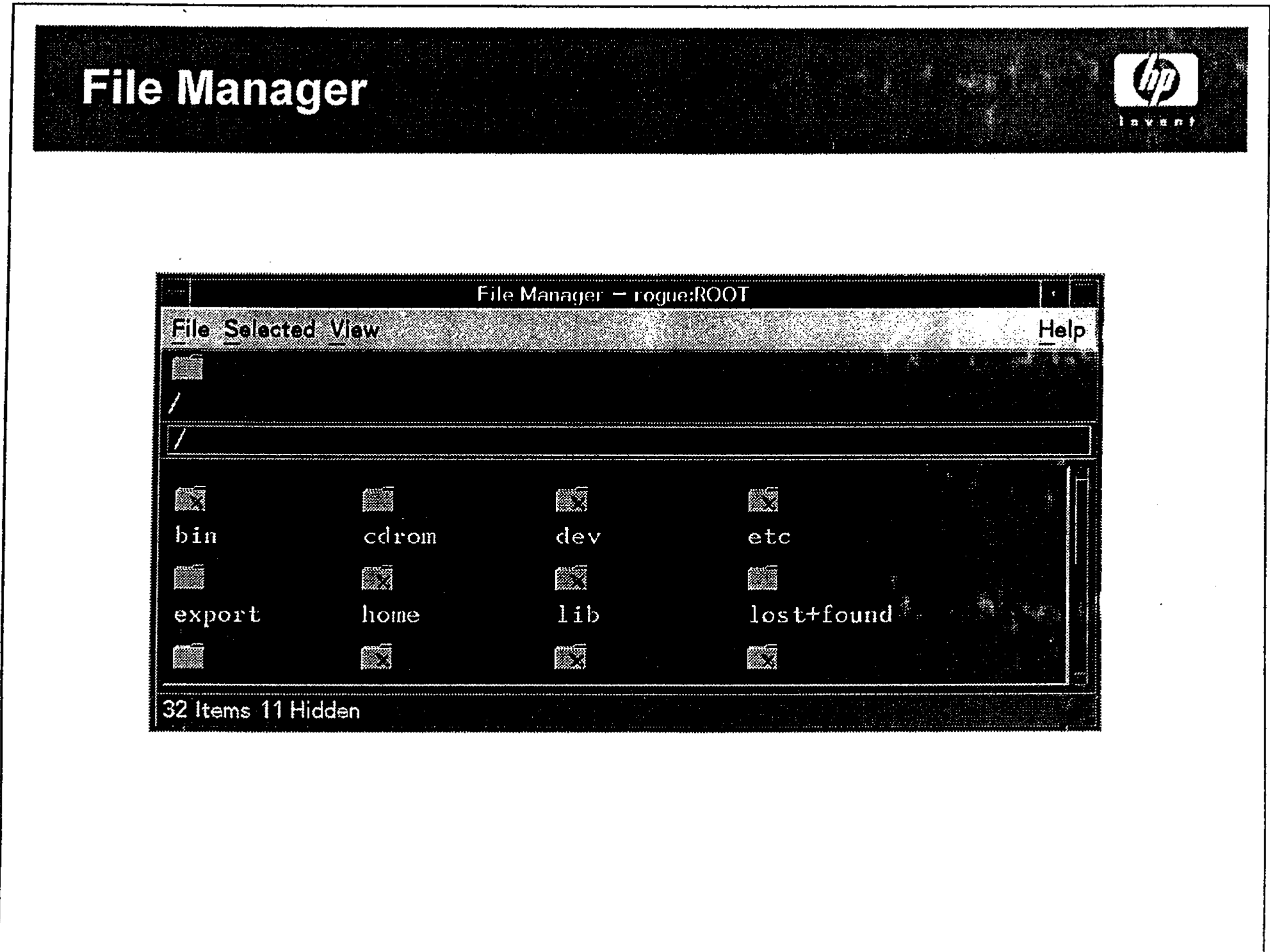
5. Click close to dismiss the man page.

Printing a Help Topic

Once you have displayed the topic page you want help on, you can print it by clicking on `File-> Print` on the menu bar.

© 2003 Hewlett-Packard Development Company, L.P.

3–5. SLIDE: File Manager



Student Notes

The File Manager allows you to manipulate files (i.e. moving, copying printing, etc) on your system without having to learn complex commands. Files, folders (directories), and applications are displayed as icons. Clicking on the File Manager control in the Front Panel activates the File Manager.

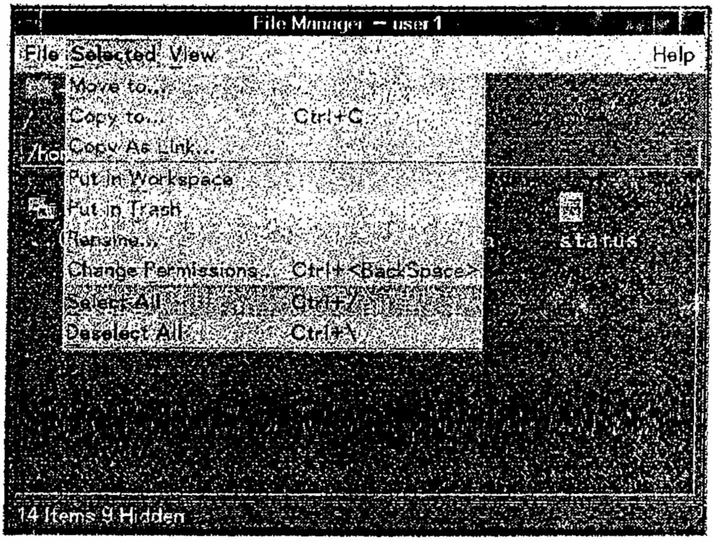
When you first get into File Manager, you are placed in your Home directory. Directories are referred to as folders. Whatever folder you happen to be in at any given time is known as the *current folder*, and the object viewing area will show the objects (files and folders) in the current folder.

Highlighting an Object

By simply double clicking on a file, you will open the Text Editor (if the file is ASCII) with the file opened for editing. By double clicking on a folder, the File Manager will bring you into that folder. Other tasks involve using the File Manager menu bar. To access a file or folder, you must first select it by clicking on it and highlighting it. In order to highlight multiple files or folders, simply press mouse button 1 in a blank area of the object viewing area, drag the mouse to draw a box around the icons you want to select, then release the mouse button. Once you have selected your object(s), you may then take action by using the File Manager Menu. If you do not select at least one object, some of the menu actions will be unavailable.

© 2003 Hewlett-Packard Development Company, L.P.

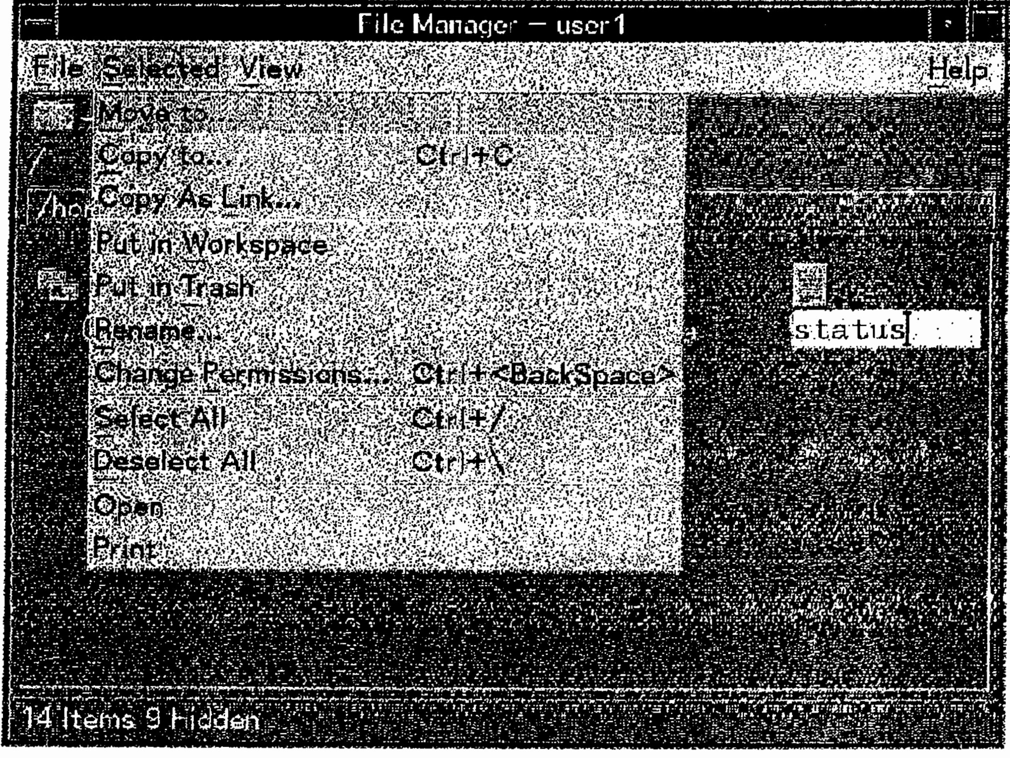
For example, on the `File Manager Selected` menu many of the actions are not highlighted because there is no selected object to take action upon.



a576125

Figure 1

Once at least one object is selected, all actions are available to choose from other tasks become highlighted and are available for the highlighted object.



a576126

Figure 2

© 2003 Hewlett-Packard Development Company, L.P.

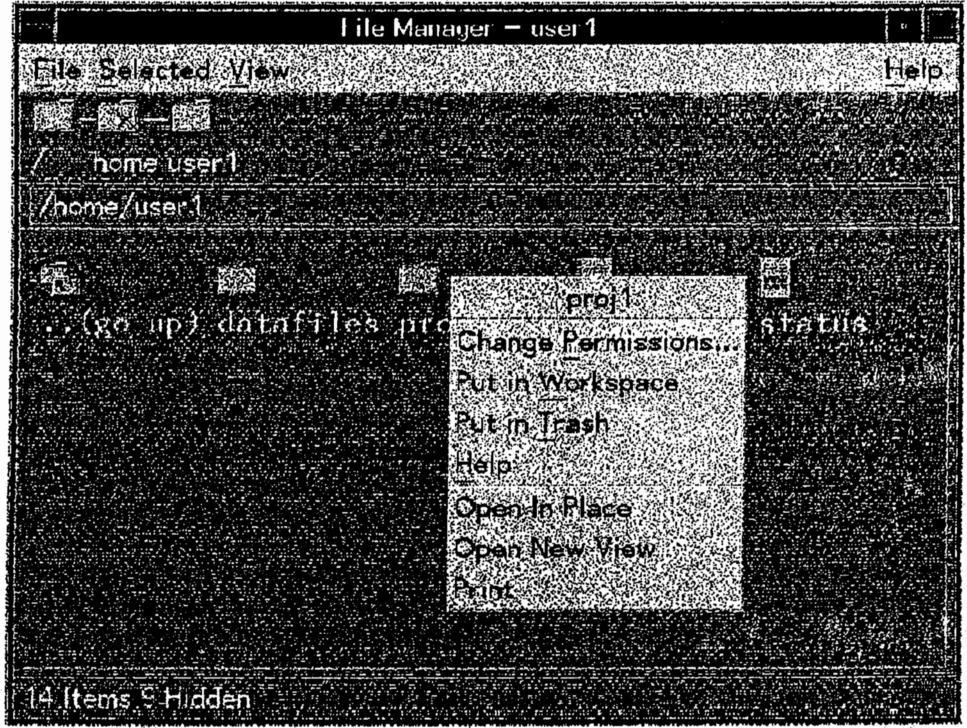
Drag-and-Drop

Objects can be dragged to other areas, either within the File Manager (to another folder for example) or to another area of the Desktop, such as the printer, trashcan, or workspace. This is done by

1. Positioning the pointer over the file(s) or folder(s) you wish to drag.
2. Press and *hold* mouse button 1.
3. Drag the icon to where you want to drop it. (i.e. printer, trash can, another folder, etc)
4. Release the mouse button.

Using Pop-up Menus

Pop-up menus can be used in lieu of the menu bar. Position the mouse pointer over the icon you wish to access, press mouse button 3 and choose the menu item.

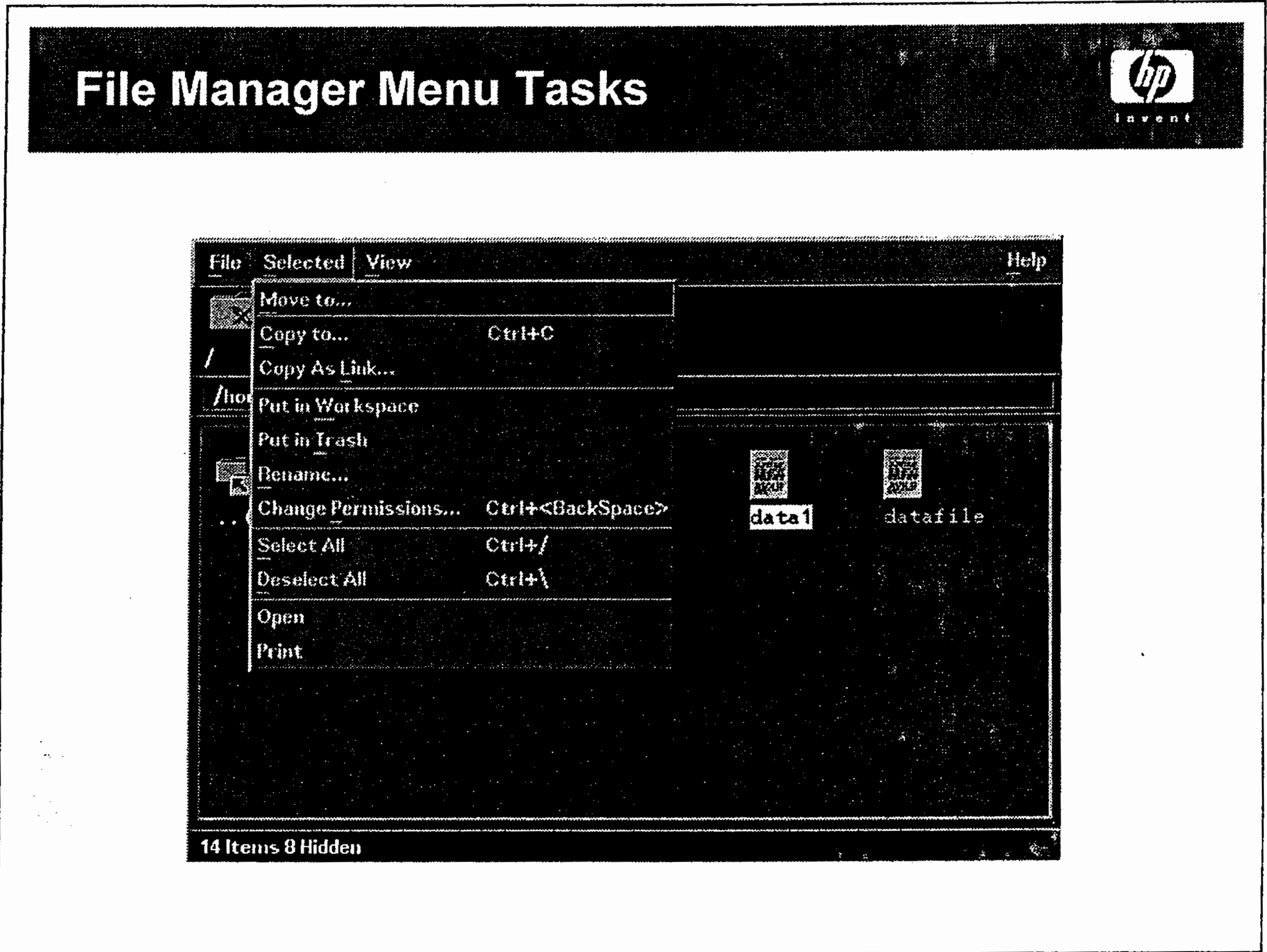


a576127

Figure 3

© 2003 Hewlett-Packard Development Company, L.P.

3–6. SLIDE: File Manager Menu Tasks



Student Notes

File related tasks that can be done at the command line, can be accomplished using the File Manager Menu Bar. In order to take action specific files or folders, they must first be selected (once they are selected, the name of the file or folder will be highlighted). If you fail to do this, many of the tasks will be unavailable. After highlighting the specific files or folders, choose `Selected` to display the list of tasks available. The `View` menu allows you to manipulate how the File Manager information is displayed.

Moving or Copying a File or Folder

1. Go into the parent folder of the file or folder you want to copy or move.
2. Select the file or folder to copy or move.
3. Choose either **`Copy to`** or **`Move to`** from the **`Selected`** menu. A pop-up dialogue box will appear prompting you for the destination folder. In the case of a copy it will also ask you for the name of the file. The default name will remain the same as the original.
4. Fill in the destination name.

© 2003 Hewlett-Packard Development Company, L.P.

5. Press OK.

Copy as Link

Copying a file as a link (this will be a symbolic link) does not actually copy the data, rather it makes a copy of the original file's icon. Any changes you make after opening the link icon will also appear when you access the file or folder using the original icon. This applies not only to the contents of the file, but also to the properties of the file as well. Therefore if you change the permissions or ownership on the original file, you also change the permissions and ownership on the linked file. To link the file:

1. Go into the folder that contains the file you want linked.
2. Select the file or folder.
3. Choose `Copy as Link` from the `Selected` menu. A pop-up dialogue box will appear prompting you for the destination folder and file name.
4. Fill in the destination name.
5. Press OK.

Change Permissions

You must be the owner or system administrator to change the permissions or ownership on a file or folder.

1. Go to the folder of the file or folder whose properties you want to change.
2. Highlight the desired file or folder.
3. Choose `Change Permissions ...` from the `Selected` menu.
4. A pop-up dialogue menu will appear with the current ownership and permission information, including the size and last modified information.
5. Fill in the information as needed.
6. Press OK.

Rename File or Folder

1. Go to the folder of the file or folder to be renamed.
2. Select the file or folder.
3. Choose `Rename...` from the `Selected` menu. The cursor will be positioned at the end of the current name of the file or folder.
4. Backspace as far back in the name as necessary, and type in new name.

© 2003 Hewlett-Packard Development Company, L.P.

5. Press Return.

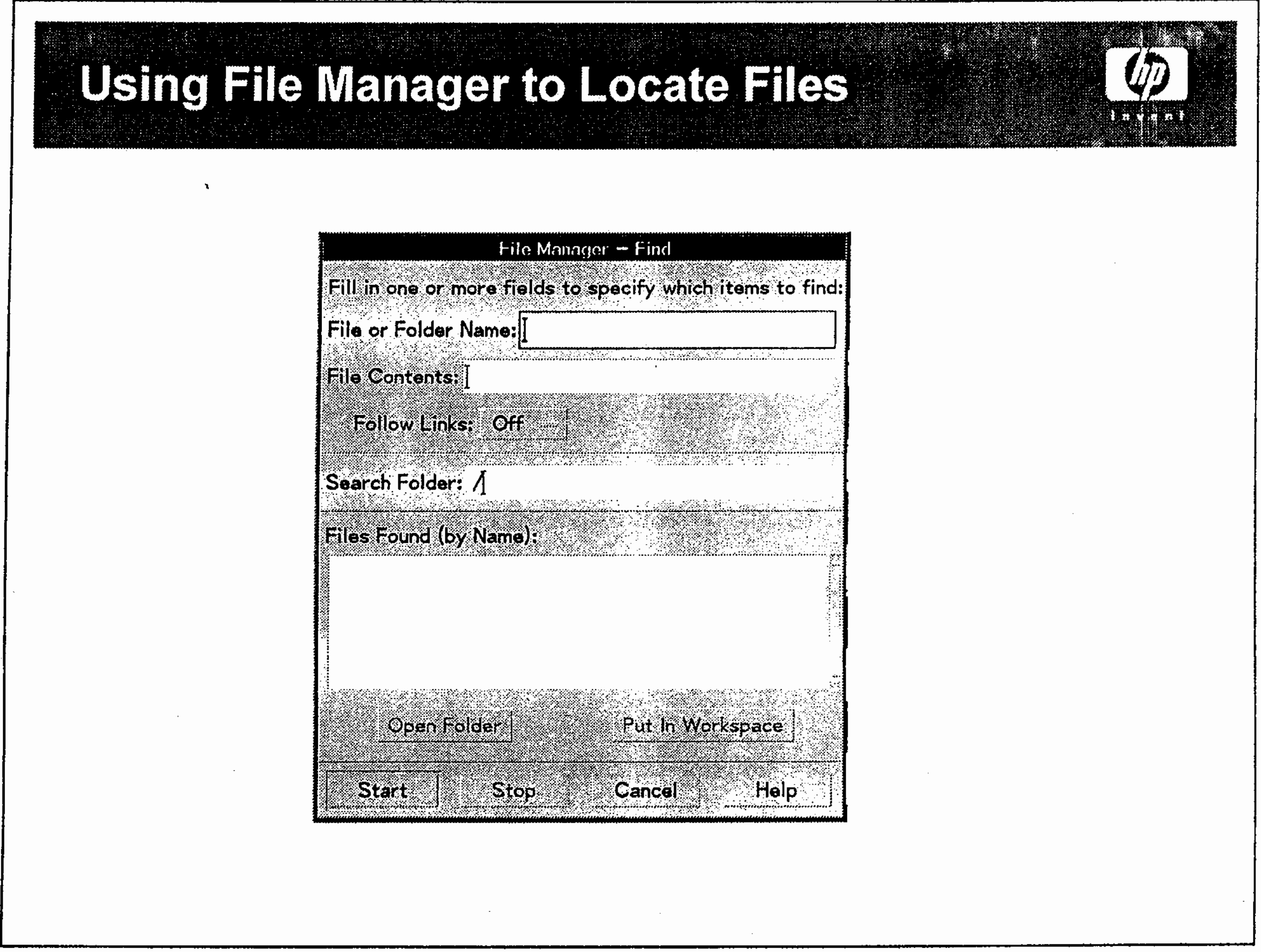
Other Tasks from Selected Menu

Depending upon the type of object the icon represents, not all of these choices will be available.

`Put on Workspace`	Places the icon of the file or folder on the backdrop of the workspace for easy access. To remove, position the mouse cursor on the icon, press mouse button 3, and choose **`Remove From Workspace`**.
`Put In Trash`	Removes the icon from the current folder to the Trash Can.
`Select All`	Selects all icons in the current folder for action to be taken upon.
`Deselect All`	Deselects all icons in the current folder.
`Open`	The action taken depends upon the type of object selected. Opening a folder will display the contents of the folder. Opening a text file will display the contents of the file in the text editor.
`Print`	If the file is a text file, the contents will be printed to the printer chosen in the dialogue box. If it is a folder, a long list of files and directories will be printed to the printer of choice.
`Run`	Applies to executable files. This action causes the file to be executed as a command.
`Imageview`	Deposits the bitmap image into the icon editor.

© 2003 Hewlett-Packard Development Company, L.P.

3–7. SLIDE: Using File Manager to Locate Files



Student Notes

File Manager gives you the ability to search for a file or folder by name or by contents of the file.

1. Click on **`File`** menu and choose **`Find`**.

2. Type the name of the file or folder you want to find in the **`File`** or **`Folder Name:`** field. Wildcards are allowed in the name:

* asterisk	Matches zero or more of a given character. For example, if you wanted to find all the files that began with the string *prog*, you would enter prog*. This would find the file prog, progA, prog1, prog.data, etc.
? question mark	Matches any single character. Using the same example, if you entered prog?, only the file progA, and prog1 would be found.

3. Type the text string you want to search for in the **`File Contents`** field. (Case is ignored)

© 2003 Hewlett-Packard Development Company, L.P.

NOTE:	Both File Name and File Contents both do not need to be filled in. By filling both in, however, it speeds up the search, because the search criteria has been narrowed.

---

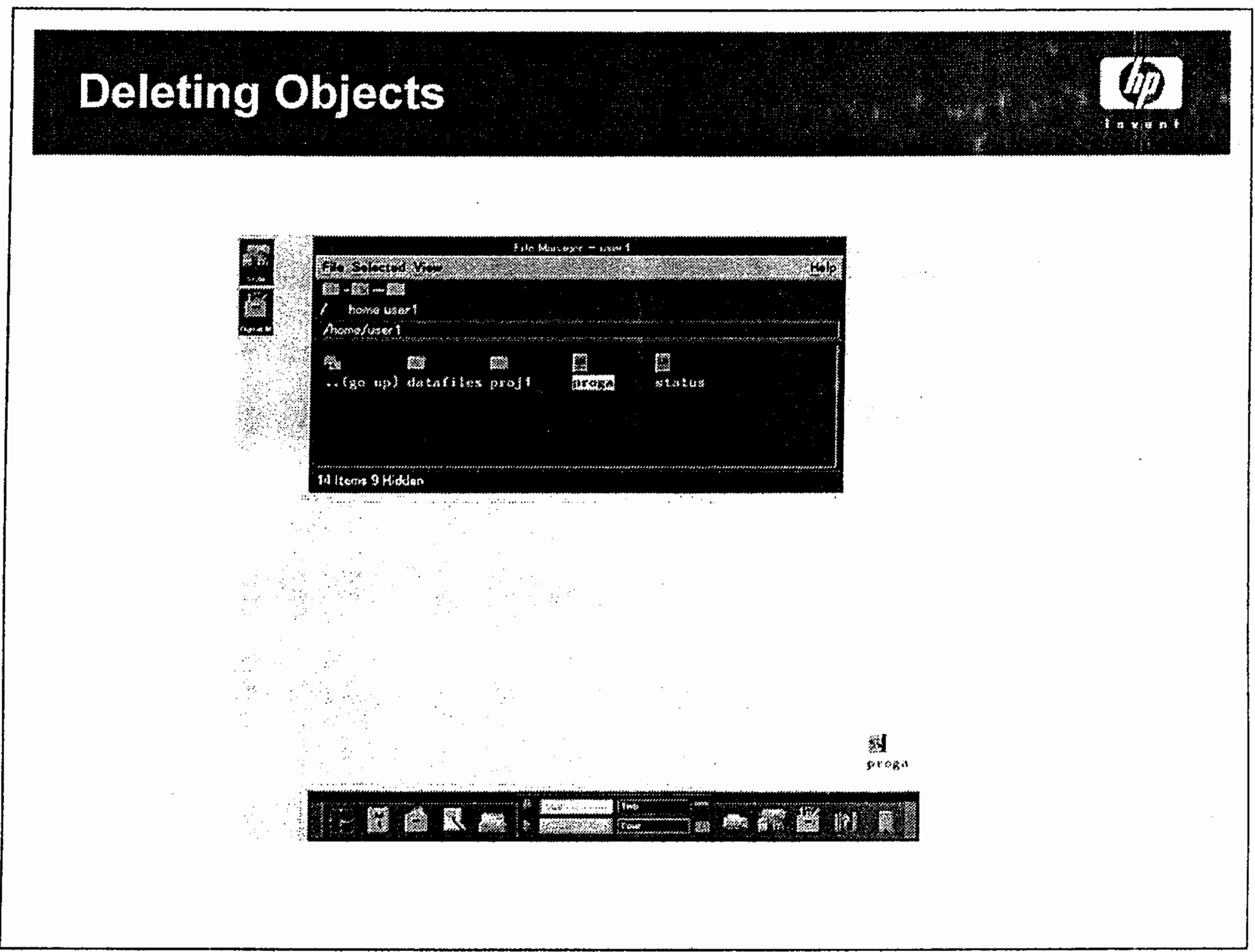
4. Type the name of the folder where you want to begin the search. Find will search this folder and any subfolders beneath it.

5. Click `Start`

When a match is found, the name of the file is placed in the **`Files Found`** window. Once the files are located, you can highlight the file name and either press **`Open Folder`** or **`Put in Workspace`**, which will place the appropriate icon on the backdrop of the workspace.

© 2003 Hewlett-Packard Development Company, L.P.

3–8. SLIDE: Deleting Objects



Student Notes

Objects can be deleted in two ways: with the File Manager Selected menu, or by dragging an object down to the Trashcan and dropping it in. The files are not actually deleted from the system, but are held in the Trashcan until it is explicitly emptied. (The Trashcan control indicates if there is trash to be emptied by a piece of paper hanging out of the Trash Can lid). Until the Trashcan is emptied, the objects can be restored to the File Manager.

To Place an Object in the Trash Can

1. Go to the folder that contains the object to be deleted to the Trashcan.
2. Highlight the object to be deleted.
3. Either choose the `Put in Trash` menu item from the `Selected` menu OR
4. Click and hold mouse button 1 while dragging the icon down to the Trashcan. Release the mouse button. You will see the Trashcan open and close indicating that the trash was deposited.

© 2003 Hewlett-Packard Development Company, L.P.

To Retrieve an Object from the Trash Can

1. Double click on the `Trashcan` icon in the Front Panel to open the `Trashcan` window. You will see a list of objects that have previously been deleted displayed.

2. Select the object you want to restore.

3. Click on `File` to open the menu bar, then select `Put Back`. The file will return to its original location.

To Delete Objects Permanently

This procedure actually deletes the file or folder from the system. Therefore it can never be retrieved, unless it has been backed up on external media, or copied elsewhere on the system.

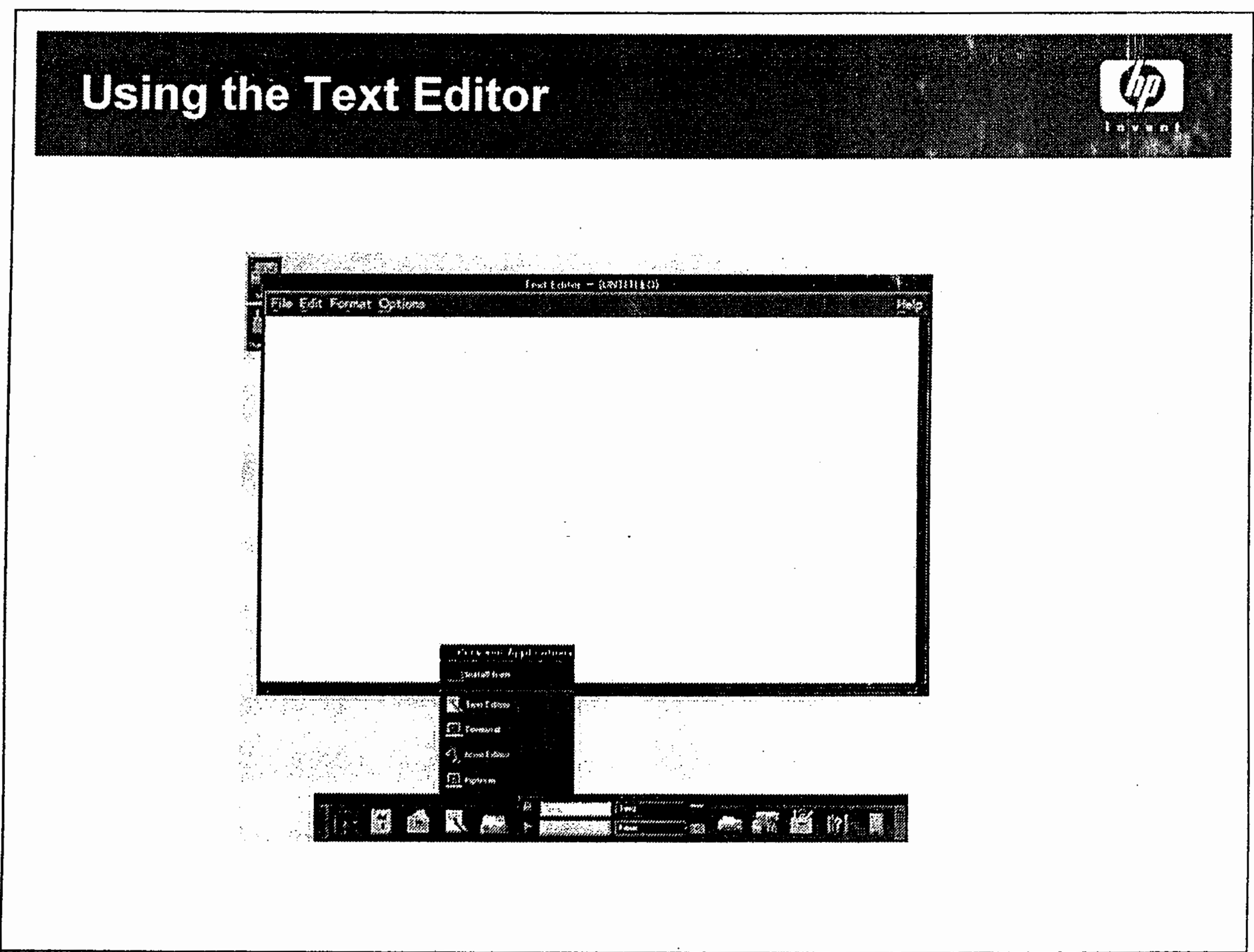
1. Open the `Trashcan` window by clicking on the `Trashcan` icon on the Front Panel.

2. Highlight the object you wish to delete. If you wish to delete all objects, open the `File` menu and `Select All`.

3. From the `File` menu, choose `Shred` to destroy the objects. You will be prompted with a confirmation dialogue box. If you are sure you want to proceed, press [OK]. *Remember these objects are now irretrievable!*

© 2003 Hewlett-Packard Development Company, L.P.

3–9. SLIDE: Using the Text Editor



Student Notes

By default, the Text Editor control will be available on the Front Panel. If this has been changed, for example to a Terminal icon, open the Personal Applications subpanel and choose Text Editor. This control activates a program called `/usr/dt/bin/dtpad`, which can be run from the command line by typing `dtpad` ***`filename`*** `&`.

Once the Text Editor window is opened, you can either create a new document by clicking on `File` on the menu bar, and then selecting `New`. To open an existing document, click on `File` on the menu bar and select `Open`. A dialogue box will prompt you for the name of the file.

© 2003 Hewlett-Packard Development Company, L.P.

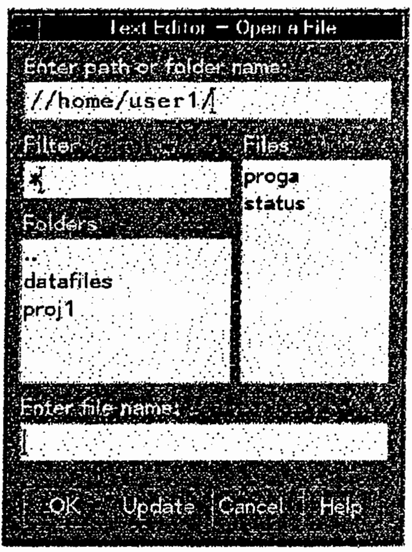


Figure 4

Double click on the name of the file, or enter the name of the file and press OK. The title bar displays the name of the current document. A new document is named (UNTITLED).

You can also include a separate document into the current one by selecting `Include` from the File menu. This does not affect the file that was included, but does update the file opened.

Editing Text

Moving Text (Cut and Paste)

1. Select the text to be moved by positioning the mouse cursor to the beginning of where you want to move, press and hold mouse button 1 while dragging the cursor across the area to be moved. Release the button.

2. Choose `Cut` form the Edit menu. The text is removed from the document and stored on a clipboard where it can be accessed later.

3. Move the cursor to where you want the text inserted.

4. Choose `Paste` from the Edit menu.

Copying Text

1. Select the text to be moved by positioning the mouse cursor to the beginning of where you want to move, press and hold mouse button 1 while dragging the cursor across the area to be moved. Release the button.

2. Choose `Copy` from the Edit menu. A copy of the text is stored on a clipboard.

3. Position the cursor where you want to insert the text.

© 2003 Hewlett-Packard Development Company, L.P.

4. Choose `Paste` from the Edit menu. If you are copying to multiple locations in the file, you only need to do the copy once, followed by multiple pastes.

Delete Text

1. Select the text to be moved by positioning the mouse cursor to the beginning of where you want to move, press and hold mouse button 1 while dragging the cursor across the area to be moved. Release the button.

2. Choose `Delete` from the Edit menu or press the [Delete] key.

Clear Text

Clearing text replaces the selected text with spaces or blank lines.

1. Select the text to be moved by positioning the mouse cursor to the beginning of where you want to move, press and hold mouse button 1 while dragging the cursor across the area to be moved. Release the button.

2. Choose `Clear` from the Edit menu.

Find and Changing Text

1. Choose `Find/Change...` from the Edit menu

2. Type the text you want to find in the `Find` field.

3. Type the replacement text in the `Change To` field. If you want to delete the text altogether, this field can be left blank.

4. Press [Return] or click [Find] to begin the search.

5. If a match is found, the cursor will be positioned at the match. To activate the change, click [Change]. If you do not want this instance changed, but want to continue searching, click `Find`. To make the change globally, click [Change All].

6. Click [Close] when done.

To Undo an Edit

1. Choose Undo from the Edit Menu. This reverses the last cut, paste, clear, delete, change, include, or format.

© 2003 Hewlett-Packard Development Company, L.P.

Correct Misspelled Words

1. Choose Check Spelling from the Edit menu. The Spell Dialogue Box will be displayed. (Figure 5)

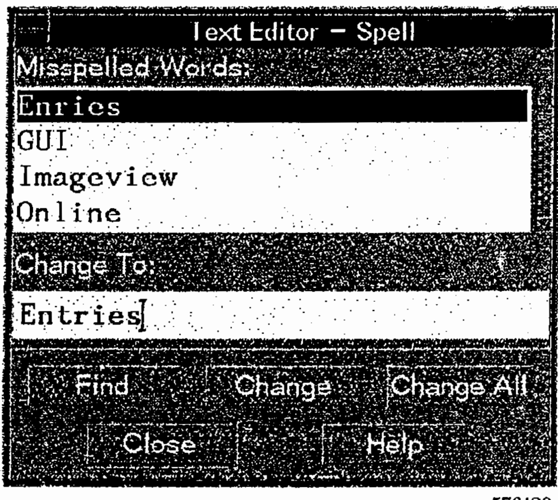


Figure 5

2. Type the correct word into the **Change To** field.

3. Click [Change] to make a single change or [Change All] to make a global change. If you simply want to locate the misspelled words and not make the changes, click [Find].

4. Click [Close] when you are done.

Formatting The Document

1. Choose Settings from the Format menu to display the Format Settings dialog box

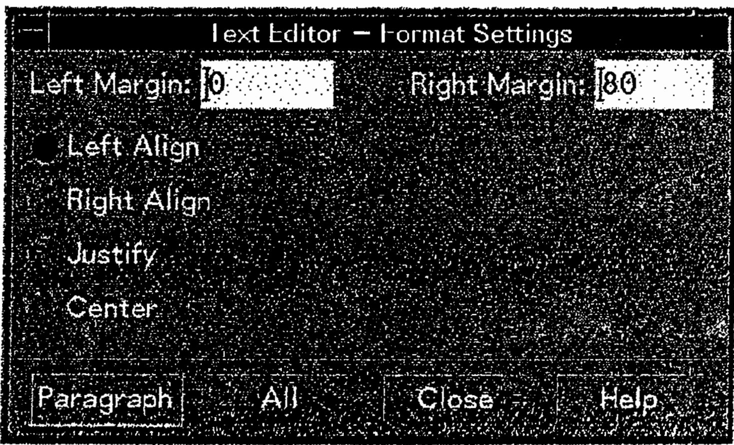


Figure 6

© 2003 Hewlett-Packard Development Company, L.P.

2. Enter margins.

3. Select left, right, justify (block style), or center alignment.

4. Determine the scope of the formatting:

   - To format a single paragraph, place the cursor in the paragraph, then click [Paragraph].
   - To format the entire document, click [All].

5. After closing the dialog box, choose [Paragraph] or [All] from the Format menu to apply the settings.

Other Text Editor Options

Overstrike Insert

Choosing `Overstrike Insert` from the Options menu will allow you to type over existing characters, rather than entering new ones. When this is no longer desired, press `Overstrike Insert` again to toggle the option off.

Wrap to Fit

Choosing `Wrap to Fit` from the Options menu controls whether lines are dynamically wrapped to fit the width of the window. When turned on, lines are broken automatically at the edge of the window. When the size of the window is changed, the line breaks are adjusted accordingly.

Status Line

Choosing `Status Line` from the Options menu creates a status line at the bottom of the document that displays the current line number and the total number of lines in the document. It also indicates when Overstrike mode is turned on.

The Status Line can also be used to go to a specific line number easily.

1. Display the Status Line.

2. Click in the Line field of the Status Line.

3. Type the line number to which you want to go and press [Return].

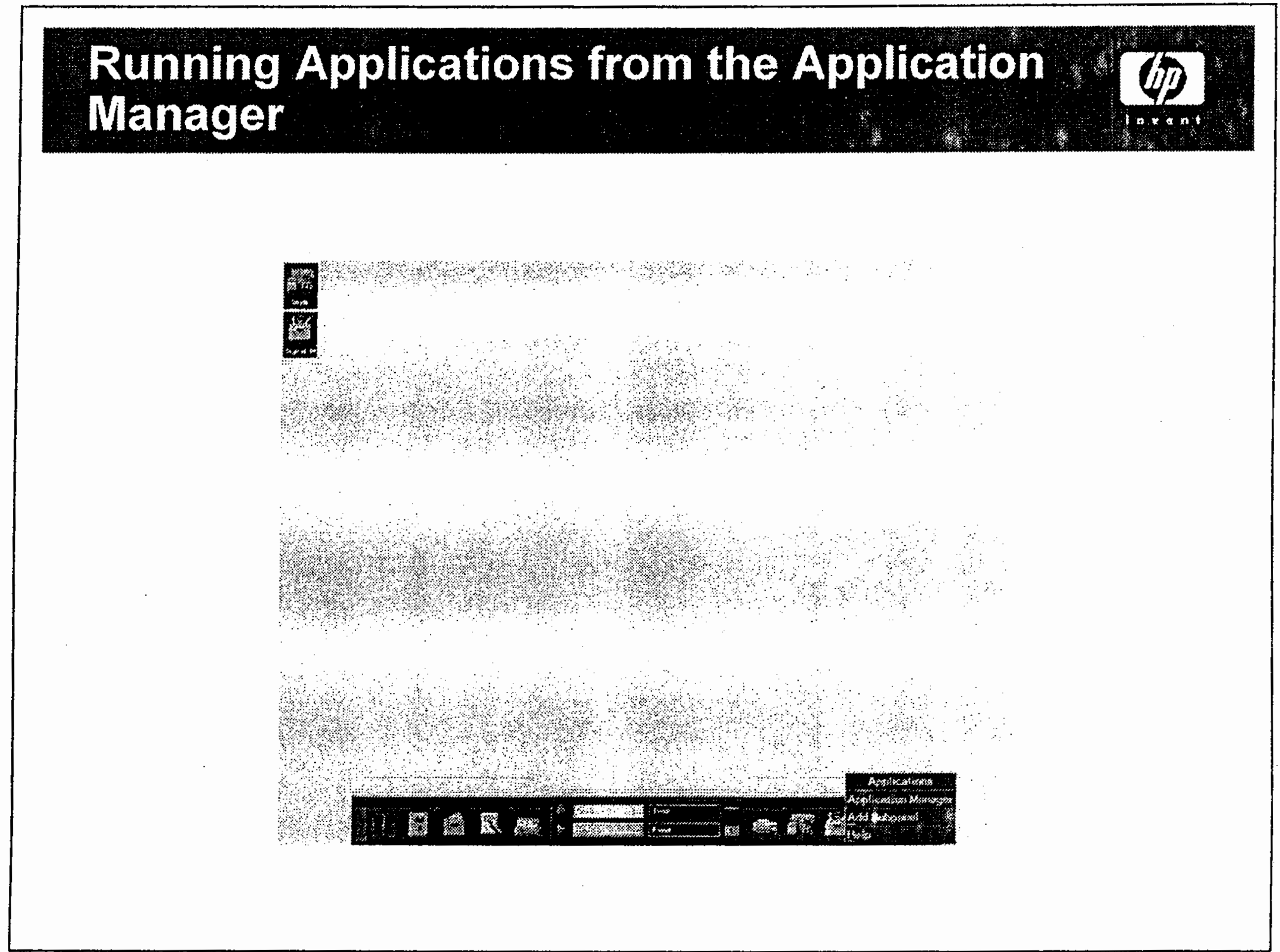
Printing a Document

1. Choose `Print` from the File menu. A printer dialog screen will appear where you can control which printer, the number of copies, banner page title, whether you want page numbers, and other printer commands.

2. Click [Print].

© 2003 Hewlett-Packard Development Company, L.P.

3–10. SLIDE: Running Applications from the Application Manager



Student Notes

Application Manager is a container for the applications and other tools available on your system. Most of the applications and tools in Application Manager are built into the Desktop. Customization can be done at the system level by the system administrator, or on a personal level by individual users.

To open the Application Manager, click on the Application Manager control on the Front Panel.

The top level of Application Manager contains the folders for the Application Groups available to the user. Applications are never directly stored in the top level of Application Manager, but instead in the Application Groups, which is a way of organizing applications according to specific functions.

To run an application from Application Manager

1. Open Application Manager.
2. Double-click the application group's icon to display its contents.
3. Double-click the application's action icon to execute the application.

© 2003 Hewlett-Packard Development Company, L.P.

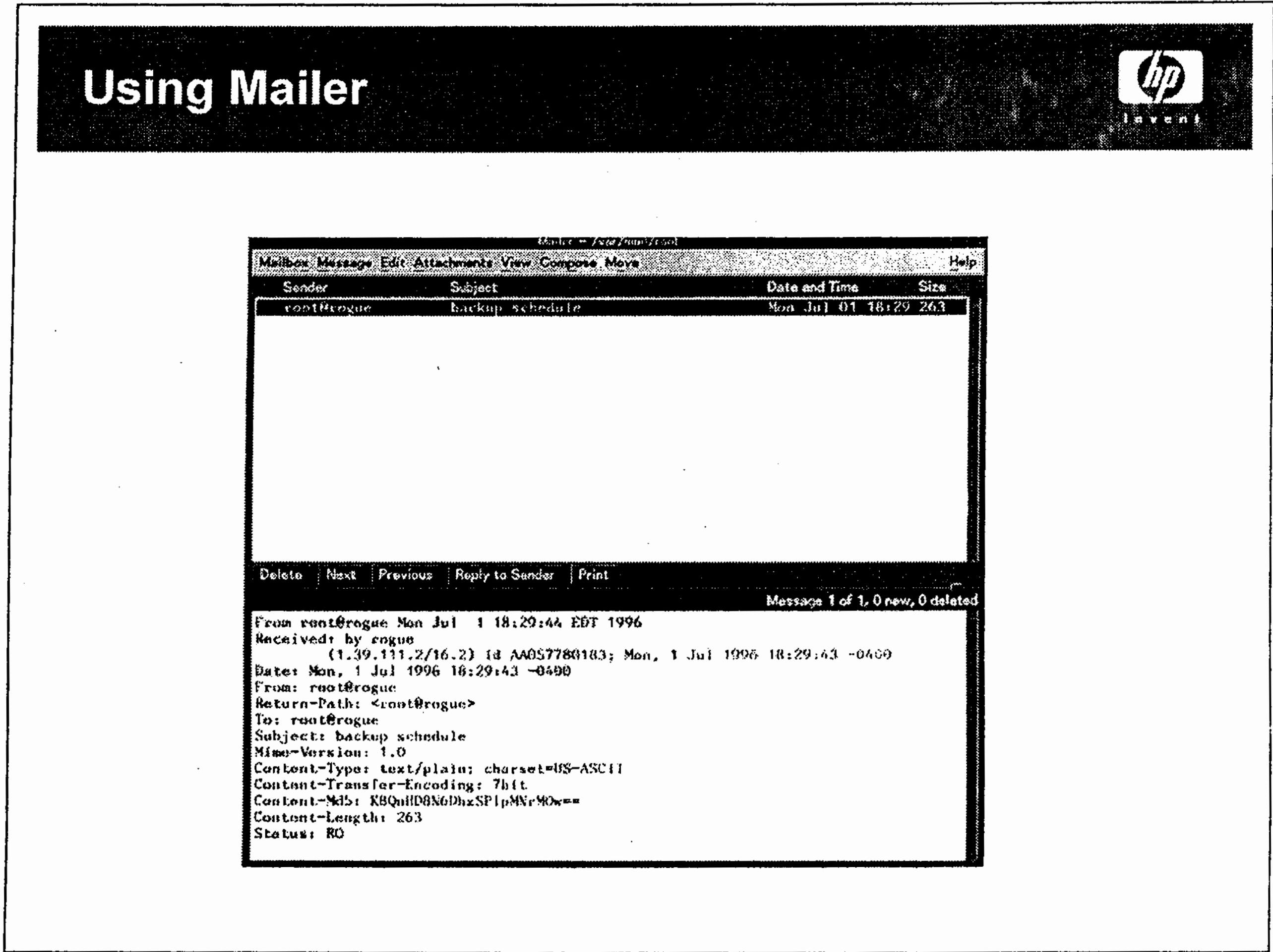
Built-in Application Groups

The Desktop provides these built-in application groups that are containers for various tools and utilities available on your system:

Desktop_Apps	Desktop applications such as File Manager, Style Manager, and Calculator
Desktop_Tools	Desktop administration and operating system tools such as Reload Application, `vi`, and Check Spelling
Information	Icons representing frequently used help topics
System_Admin	Tools used by system administrators
Digital_Media	Tools for audio, screen captures and image viewing.

© 2003 Hewlett-Packard Development Company, L.P.

3–11. SLIDE: Using Mailer



Student Notes

The CDE Mail utility allows you to send, receive, and manage your electronic mail from the Desktop. The Mailer icon on the Front Panel will change when there is new mail to be read. The icon in Figure 7 indicates that there is no unread mail, while the icon in Figure 8 indicates that there is mail to be read.

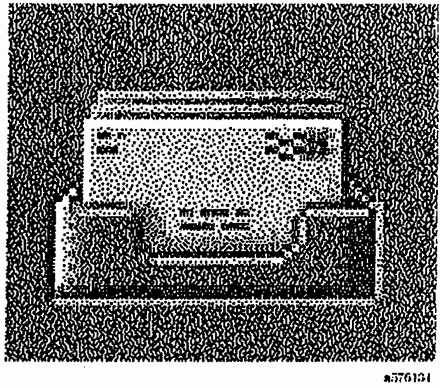


Figure 7

© 2003 Hewlett-Packard Development Company, L.P.

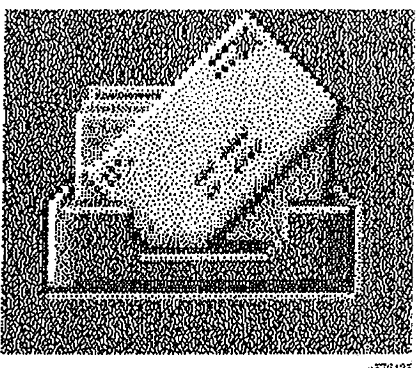


Figure 8

The Mailer main window will display the headers of any messages, whether they have been read or not. New messages are preceded by `N`. Whichever message is highlighted is the current message, and its contents are displayed in the Message View area. If the sender included an attachment, such as a calendar, or graphic, its icon will be shown in the Attachment list.

Reading Messages

To read a message, select the message from the Message Header List. The text of the message appears in the message view area. To open this into a single window, double-click the message or choose `Open` from the Message menu.

Sorting Messages

By default the messages are displayed in the order they arrived (Date/Time). You may rearrange them differently by selecting from the View menu.

Finding Messages

You can search for specific messages based on the contents of the To:, From:, Subject:, and CC: fields by choosing Find from the Message menu.

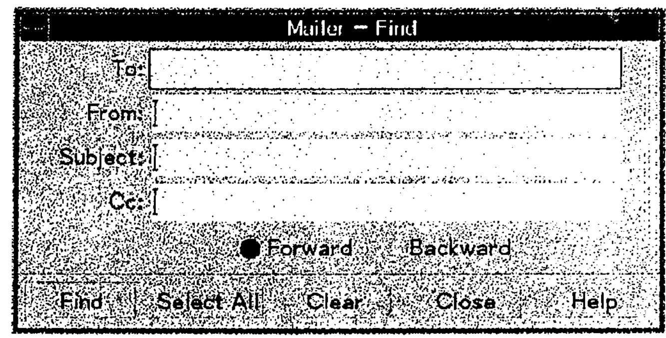


Figure 9

You can narrow the search by specifying search criteria in multiple fields (for example the From: and the Subject: fields). Once you have entered the search criteria, click [Find].

© 2003 Hewlett-Packard Development Company, L.P.

All messages, regardless of whether they match the search or not will remain in the Message Header List. The first message that matches the criteria will be highlighted. As you continue to click Find, subsequent messages matching the search criteria will be highlighted.

Taking Action Upon a Message

Once you have read a message you probably want to do something with it, such as reply, save the message into a file, delete the message once you have read it, or forward it on to someone else with your comments.

Replying to a Message

1. Select the message for reply.
2. From the Compose menu choose one of the following:
   a. Reply to Sender - will reply to sender only.
   b. Reply to All - will reply to sender and all other recipients of the message.
   c. Reply to Sender,Include - will reply to sender only, but will include a copy of the message.
   d. Reply to All,Includes - will reply to the sender and all other recipients, and will include a copy of the message.
3. Enter the reply.
4. Click Send.

Forwarding a Message

1. Select the message to forward.
2. Choose `Forward Message` from the Compose menu. The entire message, along with attachments is included. To remove an attachment, highlight it and choose `Delete` from the Attachments menu.
3. Enter the mail address for the recipients in the To: and CC: fields.
4. Include your own comments if desired.
5. Click Send.

Saving a Message into a File

1. Select the message to be saved by highlighting it.
2. Choose `Save as Text` from the Message menu.
3. Type the file name and directory in the dialog box.

© 2003 Hewlett-Packard Development Company, L.P.

4. Click [Save].

Deleting/Undeleting a Message

Deleting Messages

1. Select a message for deletion.

2. Choose `Delete` from the Message menu.

Undeleting Messages

Even if a message has been deleted, it can be retrieved unless you made your deletions permanent.

- To restore the last deleted message, choose `Undelete Last` from the Message menu.

- To restore a message deleted prior to the last deleted message, choose `Undelete from List` from the Message menu. Select one or more messages to be restored. Click [Undelete].

Destroying Deleted Messages When Closing a Mailbox

You can choose to permanently destroy the messages you have deleted when you close your mailbox. Once mail is permanently destroyed you cannot undelete the messages.

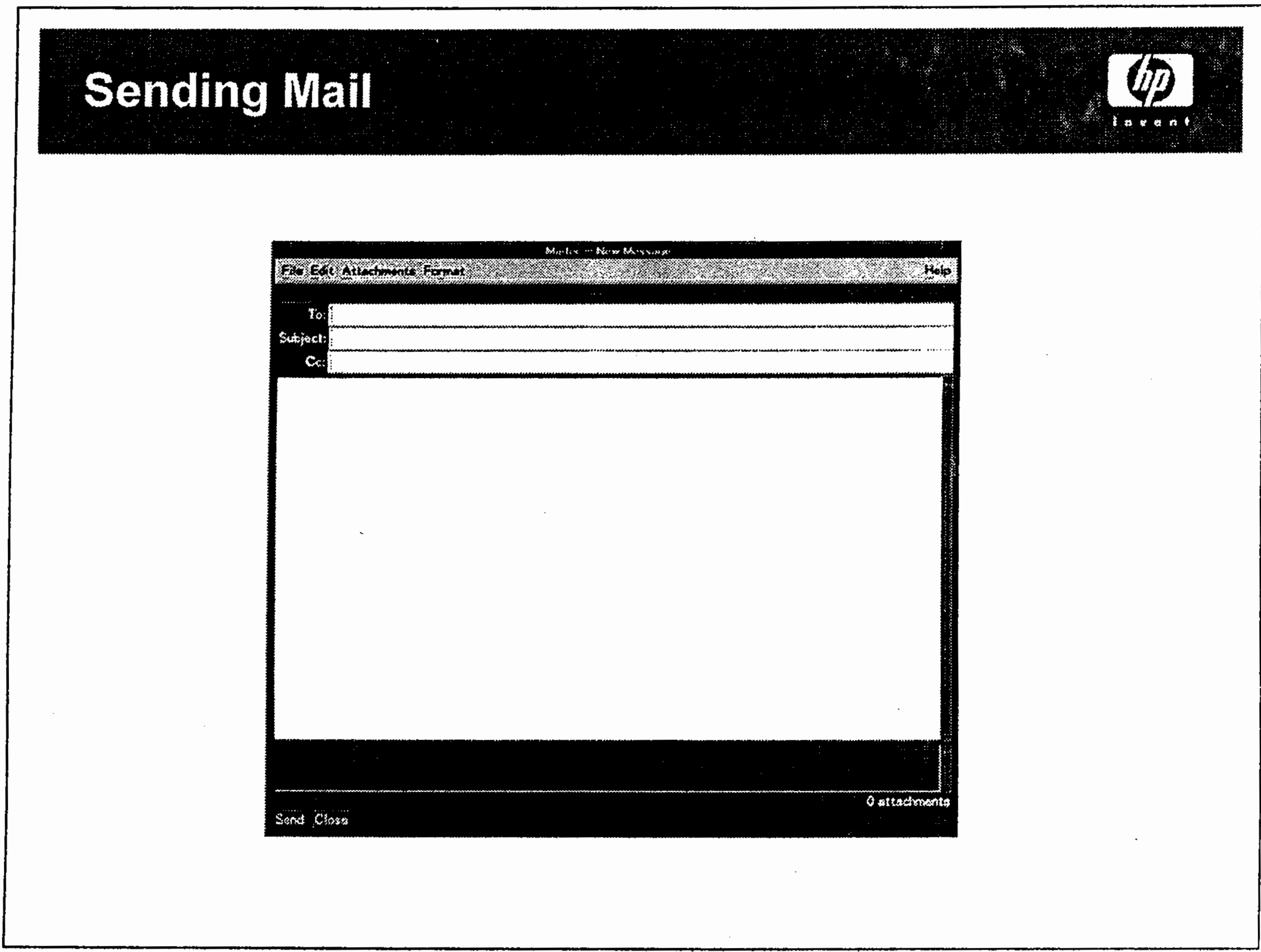
1. Choose `Message Header List` from the Category menu of the Mail Options dialog box.

2. Select `When I close the mailbox` under Destroy Deleted Messages.

3. Click [OK] or [Apply] to make your changes take effect.

© 2003 Hewlett-Packard Development Company, L.P.

3–12. SLIDE: Sending Mail



Student Notes

In order to send an email message, you need to have an email address for the recipient. The format of the email address is *username@location*. To send a message:

1. Choose `New Message` from the Compose menu (indicated in the slide).

2. Enter the recipient's email address in the `To` field, the subject of the message in the `Subject` field, and the email address of anyone you want copied on the message in the `Cc` field.

3. Once you have addressed the message, press [Return] to go to the text area and compose the message. Editing a message in the Mailer utilizes the same menu bar functions as the Text Editor.

4. Click the [Send] button. If you wish to not send the message until a later time, you can save the message by choosing `Save as Text` from the Compose menu.

You can easily include an existing text file or template into a message.

© 2003 Hewlett-Packard Development Company, L.P.

Creating a Template

You may want to create a template that contains text you frequently use when composing a message. To create a template:

1. Use the Text Editor to create the template.
2. In the Mailer, click on `Mail Options` from the `Mailbox` menu bar.
3. Click `Category` button and choose `Templates`. The Template dialog box will appear.
4. Type the name of the template in `Menu Label` field.
5. Type file path name in `File/Path`: field.
6. Click [Add] to include the template in list of templates.

Using a Template

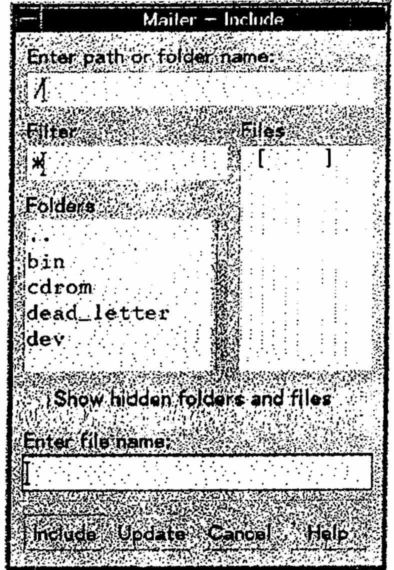
1. Choose `New Message` from the Compose menu.
2. Choose `Templates` from the Format menu.
3. Select template name to use from list available.

Including a File in a Mail Message

To mail an existing file to someone else:

1. Choose `New Message` from the Compose menu.
2. Choose `Include` from the File menu in the Compose window. (Figure 10)

© 2003 Hewlett-Packard Development Company, L.P.



a576132

Figure 10

3. Traverse through the file system and select desired file to include.
4. Click [OK].
5. Add additional text if necessary. Click [Send] when ready to mail.

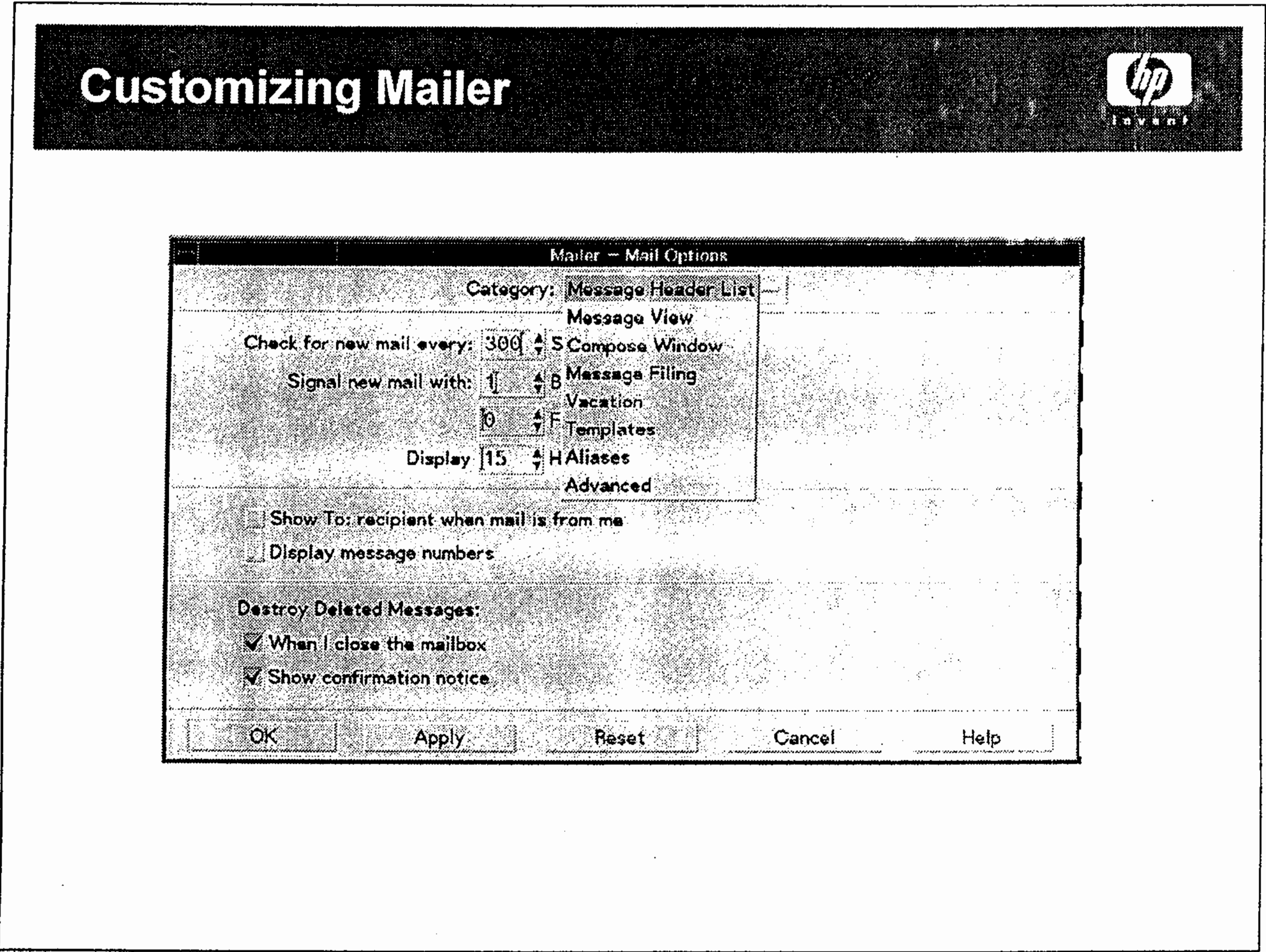
To include a non-text file you must include an attachment. To do this:

1. Choose **`New Message`** from the Compose menu if not already there.
2. Choose **`New File`** from the Attachments menu.
3. Select the desired file to include.
4. Click [OK].
5. Send as usual.

Rather than seeing the text of the file in the Message View area, you will see an attachment at the bottom of the screen with the file name. Double clicking on the attachment icon will open a Text Editor session with that included file.

© 2003 Hewlett-Packard Development Company, L.P.

3–13. SLIDE: Customizing Mailer



Student Notes

Mailer can be highly customized from the default settings. The slide depicts the Mail Options dialog box, which can be selected from the Mailbox menu. Customizations include:

Message Header List

- Frequency of new mail checks.
- Whether or not to signal if new mail arrives with a beep and/or a flashing icon.
- The number of headers that can be displayed in the message view. Whether or not the message sent displays the recipient's name, or the sender's name.
- If message numbers are displayed or not.
- Whether to automatically destroy deleted messages, or to confirm deletion upon exiting the Mailer.

© 2003 Hewlett-Packard Development Company, L.P.

Message View

- The number of lines and characters per line in Message View Area.
- Add, Delete, or Change header fields when displaying messages.

Compose Window

- Whether or not to show attachments.
- Customize the indent string for replies.
- Directory to save messages you are composing in the event that the system crashes in the middle of composing a message. The system automatically saves the messages every 10 minutes.
- Customize available options under Format menu of New Message window to include custom header fields.

Message Filing

- Determine where messages are stored.
- Specify where to begin looking for messages.
- How many mailboxes to display (according to how recently they were visited).
- Whether to log a copy of sent messages.

Vacation

- Allows users to reply with a message to all other users who send a mail message that the recipient is out of the office for a specified period of time.
- Vacation mail can be given a lower priority than other mail.

Templates

- Used to create text frequently used in composing messages.

Aliases

- Allows users to create their own private distribution list with shortened names Many users can be included in one alias name.

Advanced

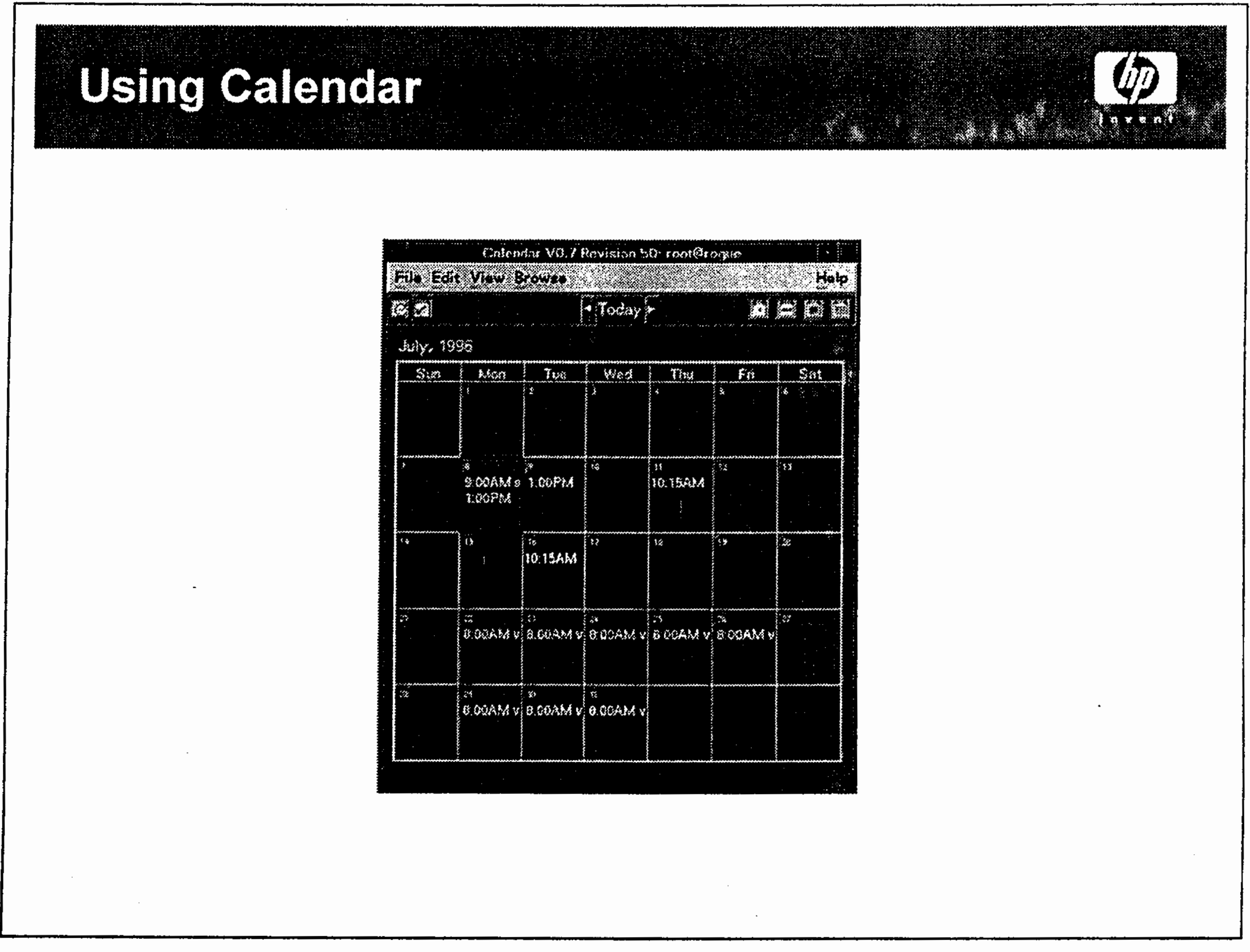
- Control how frequently the mailbox is updated.

© 2003 Hewlett-Packard Development Company, L.P.

- Whether or not to show confirmation notice when making changes (if they have not been saved). If this is not chosen, the changes will automatically be incorporated.
- Whether or not to use MIME encoding which is necessary if the recipients are running a fully compliant MIME package. This should generally be not chosen unless you are certain.
- Network aware file locking prevents two instances of Mailer from opening the same mail message.
- Whether or not to include the sender's address in the Reply to All field, and if so, whether or not the host name is included.

© 2003 Hewlett-Packard Development Company, L.P.

3–14. SLIDE: Using Calendar



Student Notes

The Calendar application allows users to schedule appointments, To Do lists, and reminders, as well as browse other calendars across the network, and schedule group appointments, if access is granted. The calendar icon is generally located on the Front Panel, and can be accessed by clicking on it.

By default, the current calendar month is displayed, with the current date highlighted. This view can be altered using the View menu bar or the controls on the right hand side of the Calendar Tool Bar (just below the menu bar). In addition to the **month view**, the displays available include:

- **Day view** displays a specific day's appointments on an hourly basis. It also provides a three-month minicalendar, which displays the current, previous, and next month. Any of these days can be displayed by clicking on the individual days.

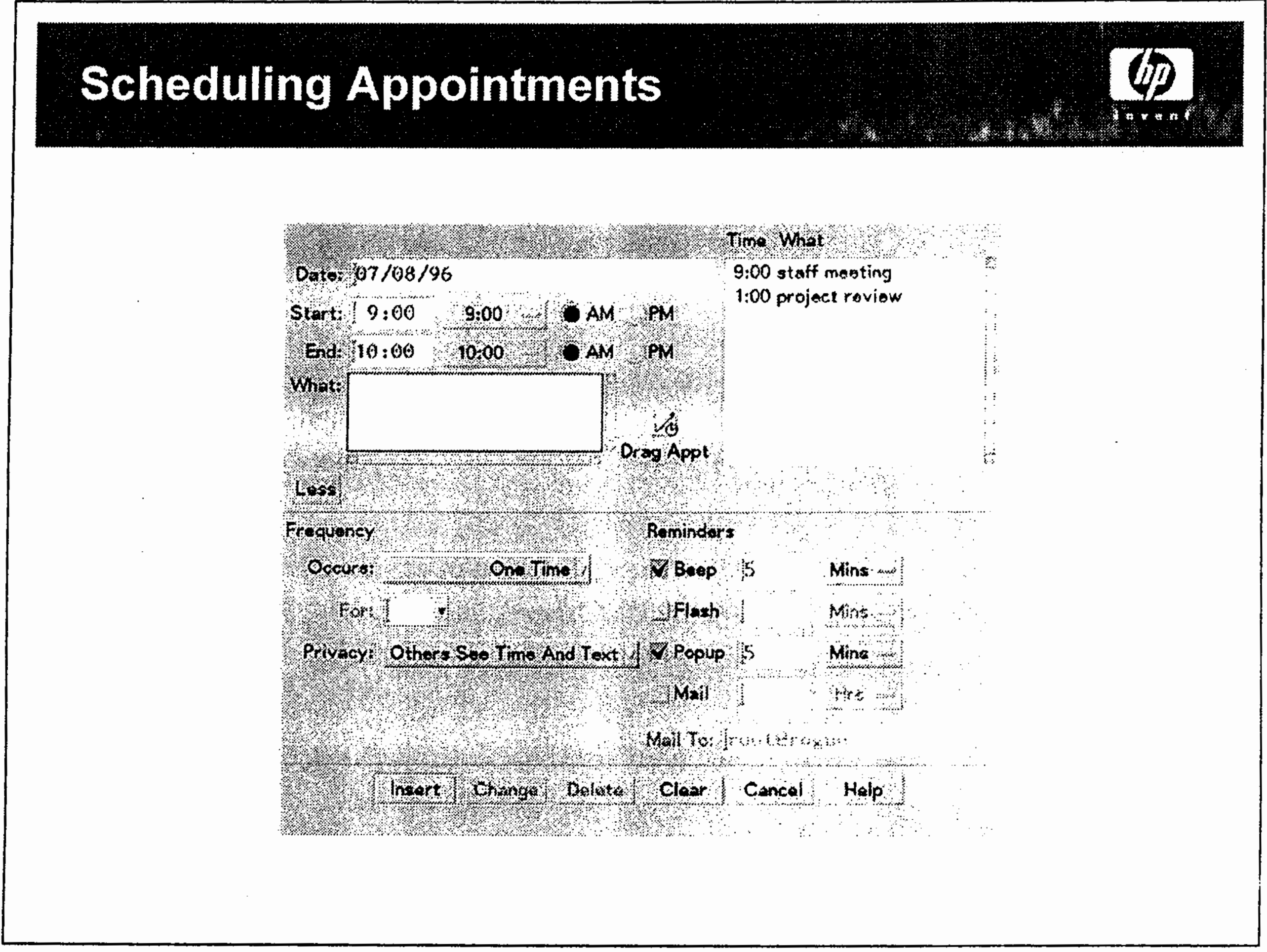
- **Week view** displays a specific week's appointments on a daily basis. A grid with an hourly breakdown is displayed indicating scheduled times, with a shaded area, and available times, with an unshaded area. By default the week displayed is the current week. Users can scroll to other weeks by clicking the left and right arrows surrounding the `Today` button.

© 2003 Hewlett-Packard Development Company, L.P.

- **Year view** displays the year calendar. Because of the amount of time covered, appointments are not displayed.

© 2003 Hewlett-Packard Development Company, L.P.

3–15. SLIDE: Scheduling Appointments



Student Notes

From the menu bar, you can include appointments on the day that is currently displayed (therefore, this cannot be done from the yearly view). To schedule an appointment:

1. Display the day you want to make the appointment.
2. Choose `Appointment...` from the Edit menu to activate the Calendar Appointment Editor.
3. Choose Start and End times.
4. Specify what the Appointment is (this will be truncated in the calendar square).
5. If you want additional options, click [More] to display choices, such as how often this will occur (is this something that must be done the first of every month, for example), whether or not you want reminders sent via beeps, flashes, or mail messages, and to what extent you want to keep this private.

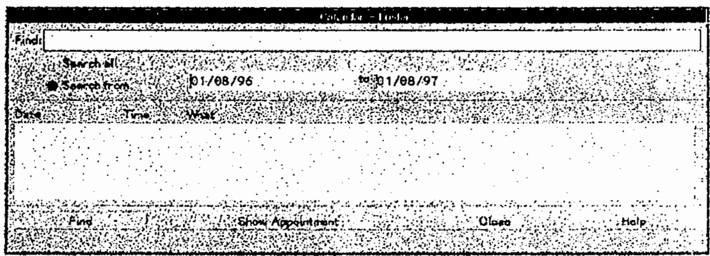
© 2003 Hewlett-Packard Development Company, L.P.

Changing or Deleting Appointments

1. Display the day you want to change an appointment.
2. Activate the Appointment Editor.
3. Highlight the appointment you want to delete or change.
4. If making a change click on the area you want to change (i.e. start/stop time).
5. Click either [Change] (which will be highlighted only if you made a change) or [Delete].

Finding an Appointment

Suppose you know an appointment is scheduled, but do not know when. Rather than scrolling through each month's calendars, you could use the Find selection from the View menu.



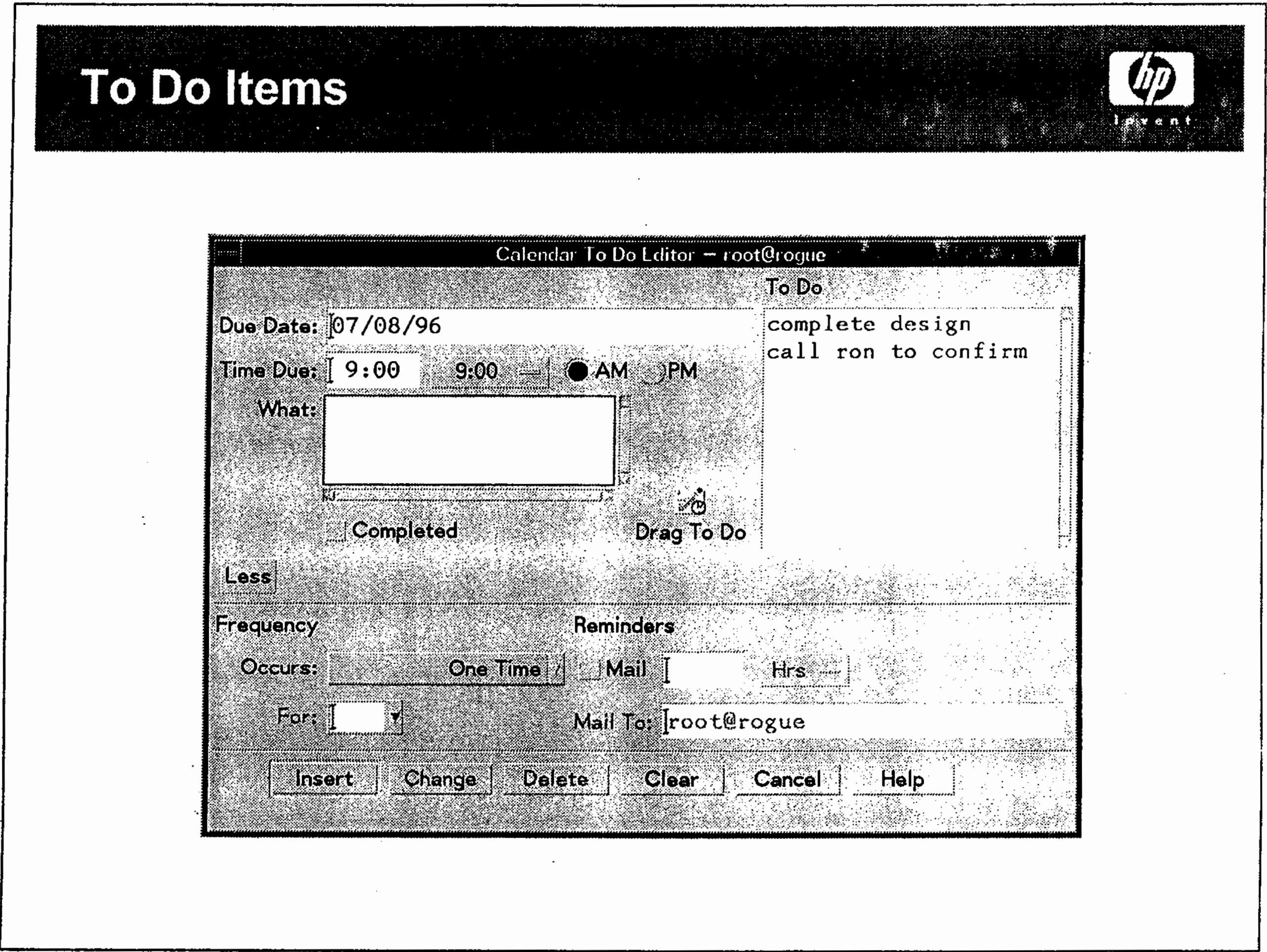
a576133

Figure 11

1. Enter a keyword in the Find field.
2. If desired, alter the range of search dates. By default, it will search the past and next six-month period.
3. Click [Find].
4. Click desired appointment from the resulting list.
5. Click [Show Appointment] to display entire appointment.
6. Click [Close].

© 2003 Hewlett-Packard Development Company, L.P.

3–16. SLIDE: To Do Items



Student Notes

Calendar gives you the capability to schedule To Do items, which can then be marked as complete, or pending. They are listed chronologically and show date, time, and description. To activate the Calendar To Do Editor select `To Do ...` from the Edit menu, or click on the To Do icon on the Calendar Tool Bar (with the check mark and pencil).

1. By default the due date will be whatever date is highlighted on the calendar, if desired edit the due date.
2. Type a description of the To Do item in the What field.
3. Click Insert.
4. Click Cancel to close the To Do editor.

© 2003 Hewlett-Packard Development Company, L.P.

Marking an Item Complete

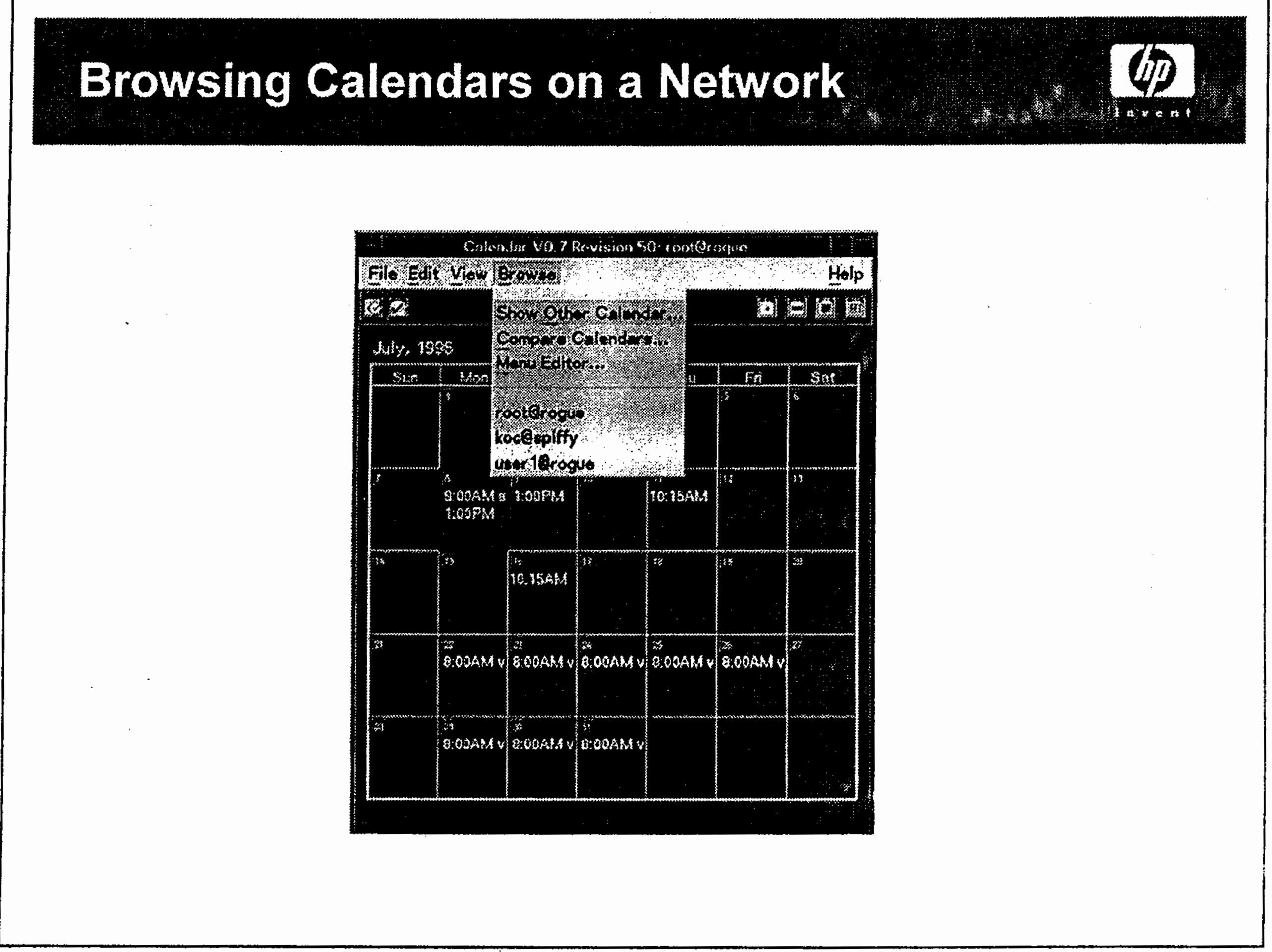
1. Select `To Do List...` from View menu bar.
2. The dialog box will display all To Do items, Completed To Do items, or Pending To Do items. Select which you want displayed.
3. Click on the box to insert a check, which marks complete the To Do item. To remove, click again to toggle off.
4. Click [OK].

Alternative method

1. Select `To Do ...` from Edit menu to display a given day's To Do items.
2. Click [Completed] to mark the item complete. (This can be toggled on or off).
3. Click [Change].
4. Click [Cancel] to close the editor.

© 2003 Hewlett-Packard Development Company, L.P.

3–17. SLIDE: Browsing Calendars on a Network



Student Notes

The desktop Calendar application provides the capability to browse other calendars across the network, as long as you know the names of the other calendars (in the form *calendar-name@hostname*), and you have been granted access to the calendars. This gives users the capability to scan a group of calendars to find an open time slot to schedule an appointment, for example.

Before you can browse a calendar, you must add it to the Browse List. To do this:

1. Choose Menu Editor from the Browse menu.
2. Type the *calendarname@hostname* in the User Name field.
3. Click [Add Name].
4. Click [OK].

Once you have added a calendar name to the list, you can browse the calendar.

1. Select `Choose Calendars` from the Browse menu.
2. Select the name of the calendar(s) you want to view.

© 2003 Hewlett-Packard Development Company, L.P.

3. Calendars are overlaid, one on top of another. Busy times are shaded, available times are unshaded.

To schedule an appointment on other calendars

Once you are in the **`Compare Calendar`** screen, you can schedule appointments, provided you have been granted access to do so.

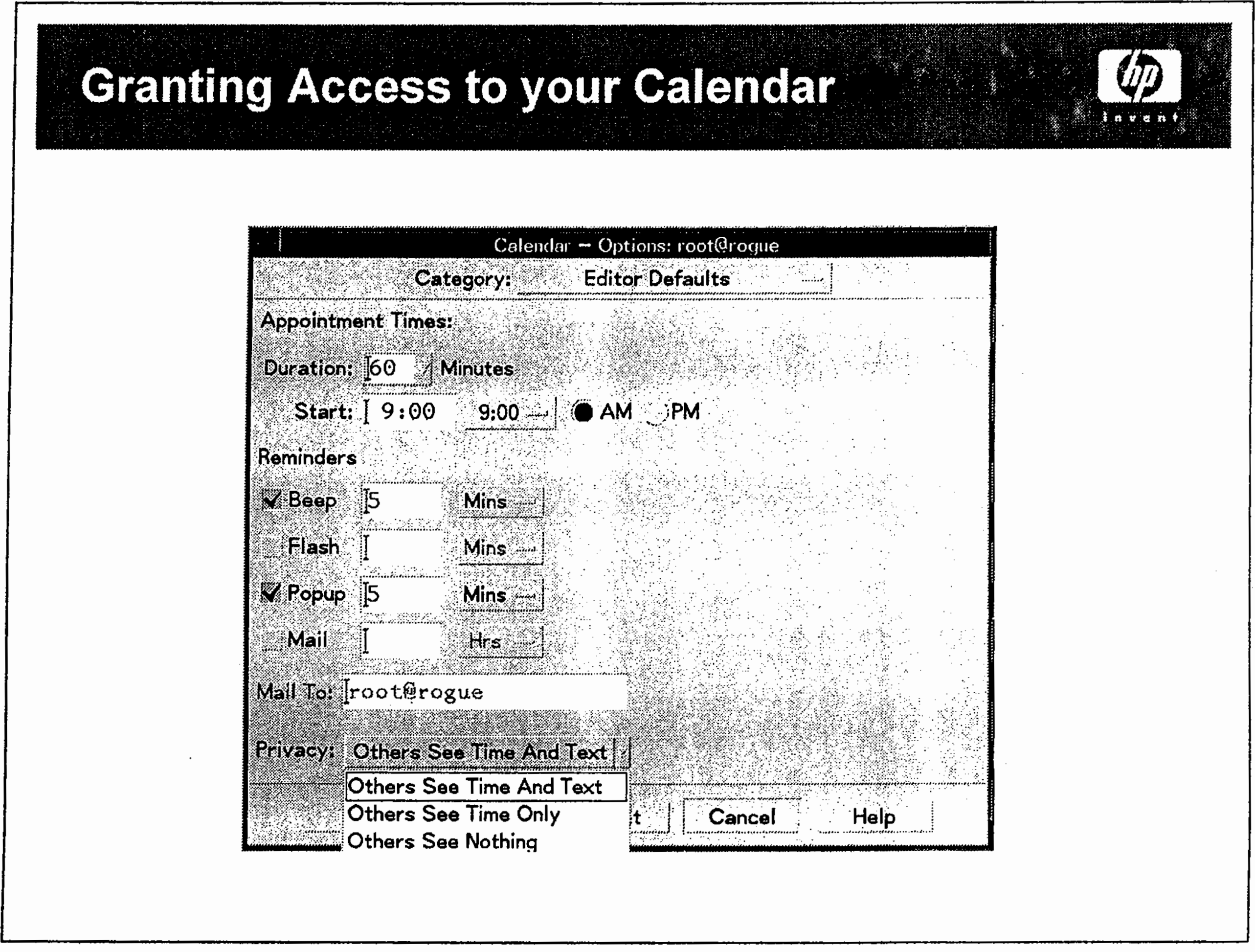
1. Select one or more entries in the Browse list.

2. Click on an unshaded area available to all entries.

3. Click [Schedule]. The Calendar Group Appointment Editor will be displayed, which will provide a Calendar Access list. A **`Y`** in the Access column means that you have insert access to update their schedule. An **`N`** means that you don't. The owners will have to grant you access in order for you to schedule appointments in their calendars.

© 2003 Hewlett-Packard Development Company, L.P.

3–18. SLIDE: Granting Access to Your Calendar



Student Notes

By default, the calendar is configured giving the world browse permission. Only the calendar owner can insert and delete appointments. The owner can change these permissions by:

1. Choose `Options...` from the File menu.
2. From the Category menu, choose **`Access List`** to display the Access List and Permissions dialog box as depicted on the slide.
3. In the **`User Name`** field, type **`calendar-name@hostname`** for the user to whom you want to grant access.
4. Select View, Insert, and/or Change permissions.

`Public`	Enables another calendar to display the time and text of your appointments marked Others See Time and Text.
`Semiprivate`	Enables another calendar to display the time and text of your appoints marked Others See Time Only.

© 2003 Hewlett-Packard Development Company, L.P.

`Private` — Enables another calendar to display the time and text of your appointments marked Others See Nothing.

5. Click [Add] to add the calendar to the Access list with the permissions you've chosen.
6. Click [OK] or [Apply].

© 2003 Hewlett-Packard Development Company, L.P.

3–19. LAB: Using CDE

Objective: To become comfortable using the basic elements of the CDE Desktop Environment.

Linux Systems:

If you are using KDE windowing on your Linux system, try to find applications or options equivalent to those that have been described for the CDE environment.

HP-UX Systems:

Directions

1. Using the File Manager, change to the `class` folder. Select the file `cde_intro` and copy to a file called `cde_intro2`.

2. Move the `cde_intro2` file to the `cde_dir` folder.

3. Change to the `cde_dir` folder. Change the permissions on the file `cde_intro2` to be read only.

4. Return to your home folder. Use File Manager to search for all the files that contain the contents *graphical environment*. Use the search folder `class`.

© 2003 Hewlett-Packard Development Company, L.P.

5. Use File Manager to search for all files that begin with *data*.

6. Wildcard searches can be performed using a question mark (?) to find any single character. Use File Manager to search for all files that begin with data followed by a single character.

7. Use File Manager to delete the file `cde_intro2` from the `cde_dir` folder.

8. Retrieve the `cde_intro2` file from the trash.

9. Use File Manager to permanently delete the file `cde_intro2` from the `cde_dir` folder.

10. Using the text editor open the file *$HOME*`/class/cde_intro` for editing. Copy the first paragraph to be included at the end of the document.

© 2003 Hewlett-Packard Development Company, L.P.

11. Change all except the first and third occurrences of *CDE to Common Desktop Environment.*

12. Correct all misspelled words in the file.

13. Using the pop-up menu on the workspace switch, add another workspace and call it **`CDE Class`**.

14. Using subpanel menus and pop-up menus, change the icon from the Text Editor to the Terminal for the Personal Applications.

15. From the Application Manager, use the **`Man Page Viewer`** to execute the man page for **`ls`**.

16. Choose a partner to send mail to. If you are on separate systems, you must know the partner's user name and host name of the system. This information is necessary to formulate the user's email address, which is in the format **`username@hostname`**.

    Send your partner a message, and have them send you a message. (If you are having

© 2003 Hewlett-Packard Development Company, L.P.

trouble finding a mail partner, you can send the message to yourself).

17. Once your partner has sent you a message, reply to the message, and forward the original message to a third partner.

18. Using the Text Editor, create a template of your status report that will be used to send your monthly status report to your manager every month. Save the file as `status`. In the Mailer, create a template called `monthly` containing this newly created file. Use this template to send a status report to your mail partner.

19. Use the Calendar function to create five or six appointments in the current month. You can have the appointment occur only once, or on a regular basis, such as every month.

20. Set up your calendar configuration so that you can browse your calendar and your mail partner's calendar by adding both your calendars to the Browse List.

21. You want to schedule an important meeting with your mail partner, but want to first check their calendar for their availability. Browse the calendars so you can see both your mail partner's and your own calendar at the same time.

© 2003 Hewlett-Packard Development Company, L.P.

22. Grant your mail partner Insert and Change access to your calendar so that they will be able to schedule appointments with you when necessary.

23. Schedule a meeting with your mail partner, and mail a reminder to your partner.

© 2003 Hewlett-Packard Development Company, L.P.

© 2003 Hewlett-Packard Development Company, L.P.

Module 4 — Navigating the File System

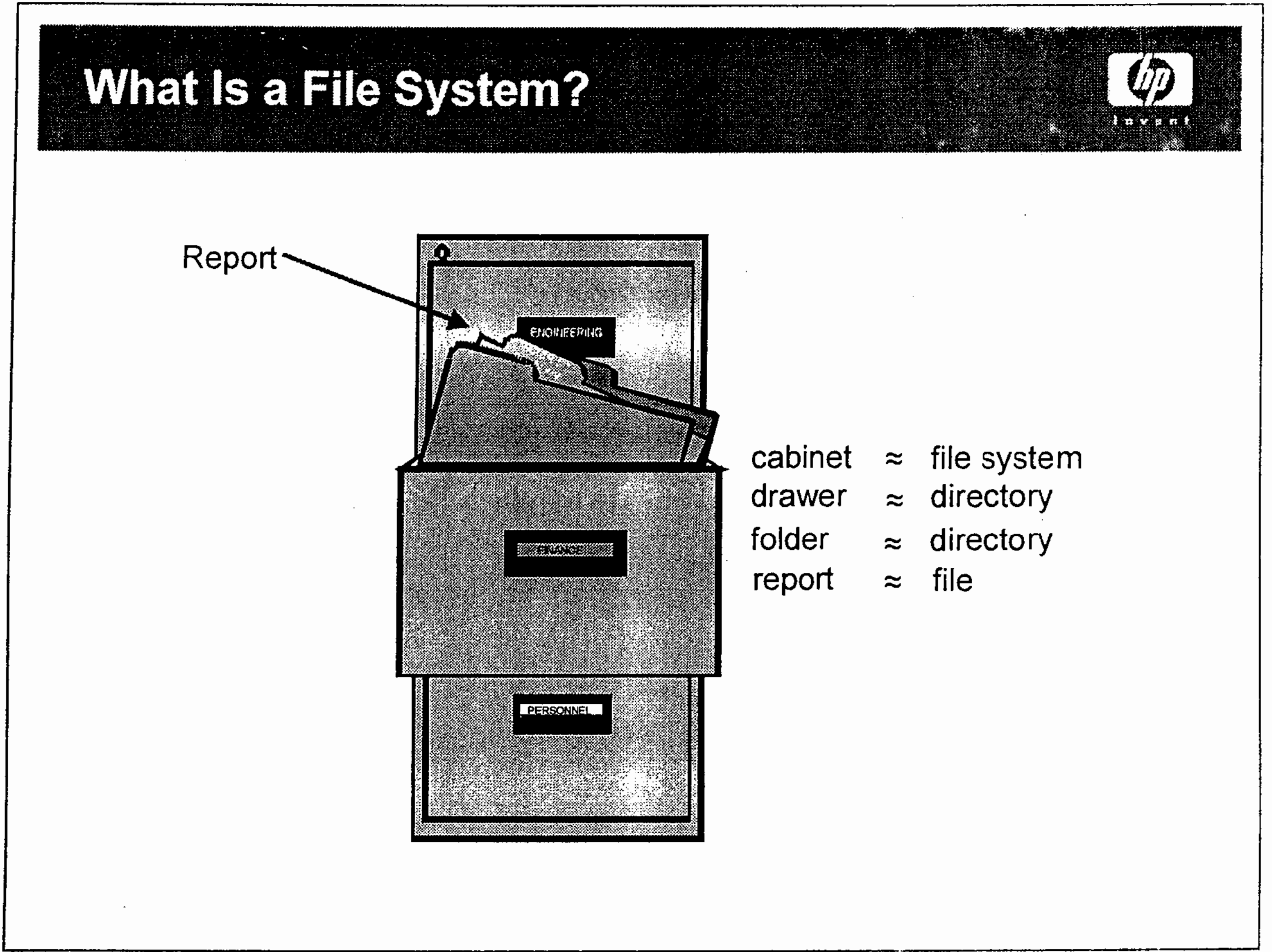
Objectives

Upon completion of this module, you will be able to do the following:

- Describe the layout of a UNIX system's file system.
- Describe the difference between a file and a directory.
- Successfully navigate a UNIX system's file system.
- Create and remove directories.
- Describe the difference between absolute and relative path names.
- Use relative path names (when appropriate) to minimize typing.

© 2003 Hewlett-Packard Development Company, L.P.

4–1. SLIDE: What Is a File System?



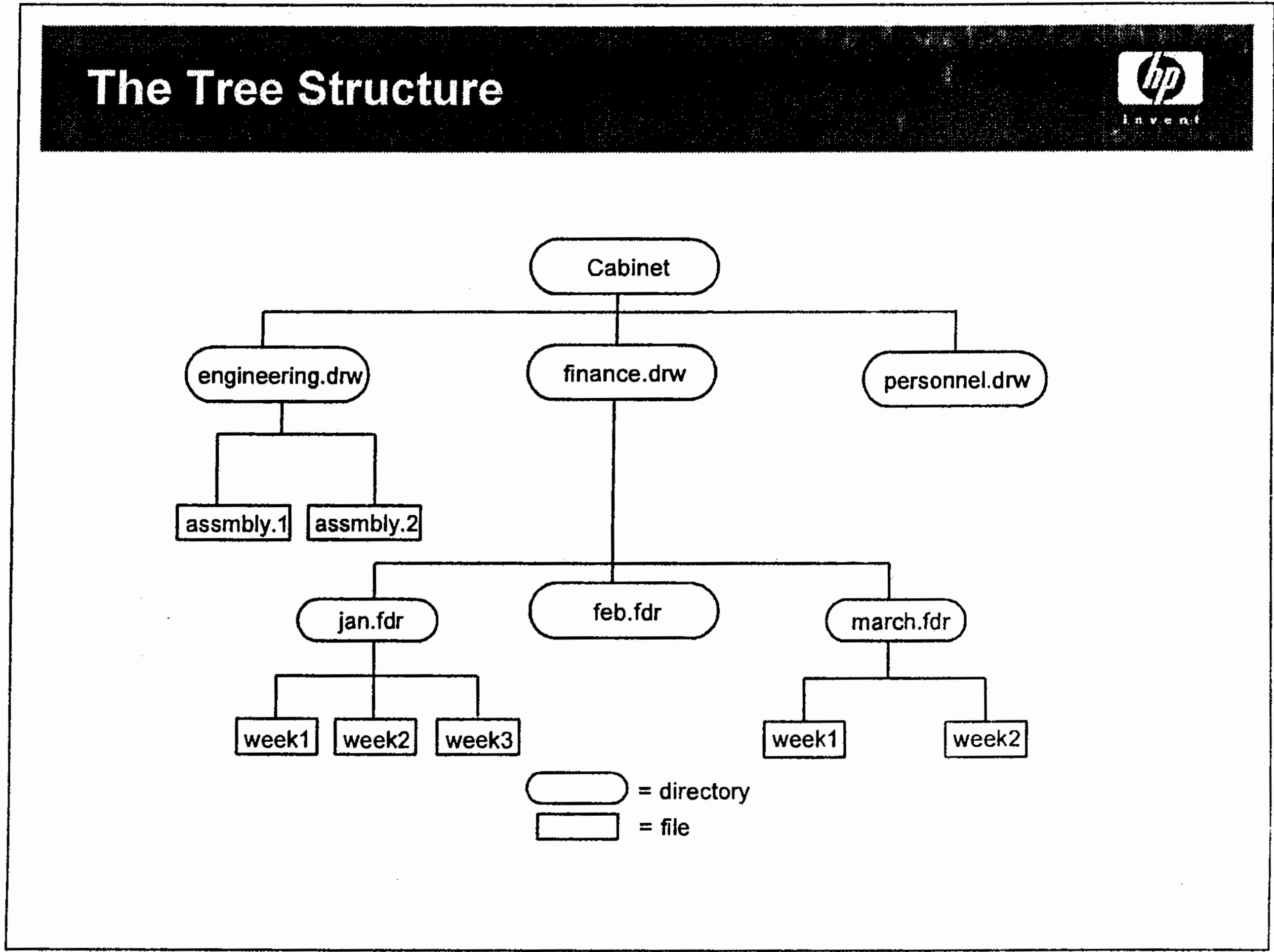
Student Notes

The UNIX system provides a **file system** to manage and organize your files and directories. A **file** is usually a container for data, while a **directory** is a container for files and/or other directories. A directory contained within another directory is often referred to as a **subdirectory**.

A UNIX system's file system is very similar to a file cabinet. The entire file system is analogous to the file cabinet, as it contains all of the drawers, file folders, and files. A drawer is similar to a subdirectory in that it can contain reports or file folders. A file folder would also represent a subdirectory, as it contains reports. A report would represent a file, as it holds the actual data.

© 2003 Hewlett-Packard Development Company, L.P.

4–2. SLIDE: The Tree Structure



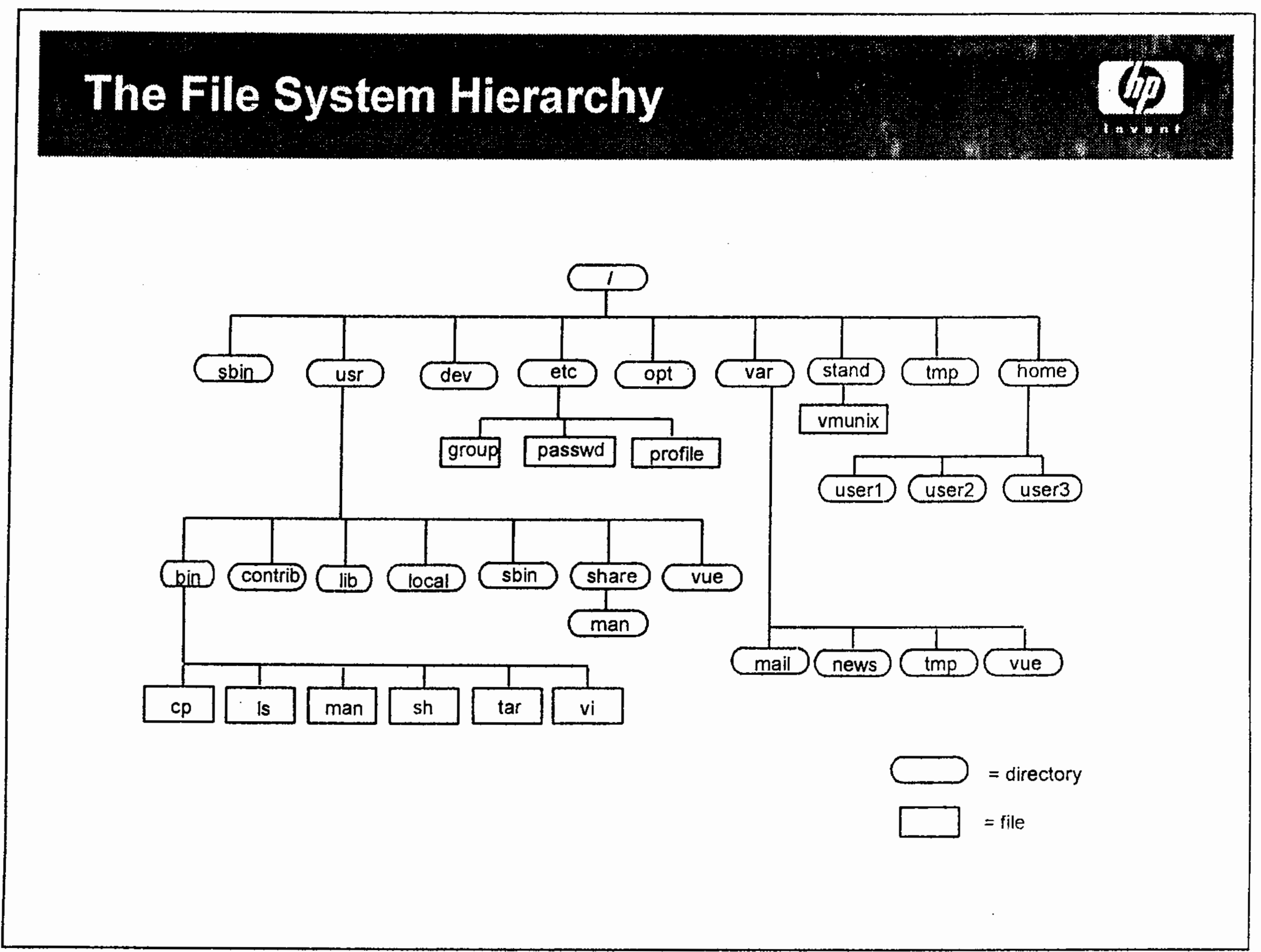
Student Notes

The directory organization can be represented graphically using a hierarchical **tree structure**. Every item in the tree will be either a directory or a file. Directories are represented by ovals, and files are represented by rectangles so that they may be easily distinguished in the diagram.

The slide illustrates a graphical tree representation of the filing cabinet from the first slide.

© 2003 Hewlett-Packard Development Company, L.P.

4–3. SLIDE: The File System Hierarchy



Student Notes

Like the filing cabinet, a UNIX system's file system hierarchy provides an easy, effective mechanism to organize your files. Since a UNIX system distribution normally contains hundreds of files and programs, a hierarchy convention has been defined so that every UNIX system supports a similar directory layout. The top of the hierarchy is referred to as the `root` directory (because it is at the top of the inverted tree), and is denoted with a single forward slash (`/`).

The UNIX system also provides commands that allow you to create new directories easily, as your organizational needs change, as well as commands to move or to copy files from one directory to another. It's as easy as adding a new file folder to one of the drawers in your file cabinet or moving a report from an old folder to a new folder.

With the release HP-UX 10.0, the file system has been reorganized into two major parts: static files and dynamic files.

Static Files (These are shared.) There are three important directories in this part: `/opt`, `/usr`, and `/sbin`.

© 2003 Hewlett-Packard Development Company, L.P.

`/opt`	Contains applications and products. Developers and administrators of the HP-UX system will use it to install new products or local applications.
`/usr/bin`	Contains the programs for all reference manual section 1 commands, which are necessary for basic UNIX system operation and file manipulation. These are normally accessible by all users. (**`bin`** is short for binary).
`/usr/sbin`	Contains the programs for all reference manual section 1m commands. They are system administration commands. You must be superuser to use many of them. These are documented in the reference manual sections 1m.
`/usr/lib`	Contains archive and shared libraries used for applications.
`/usr/share`	Contains vendor independent files (the most important is the manual).
`/usr/share/man`	Contains all files associated with the online manual pages.
`/usr/local/bin`	Usually stores locally developed programs and utilities.
`/usr/contrib/bin`	Usually stores public programs and utilities. You might retrieve these from a bulletin board service or a user group.
`/sbin`	Contains the essential commands used for startup and shutdown.
Dynamic Files	(These are private.) There are seven important directories in this part: **`/home`**, **`/etc`**, **`/stand`**, **`/tmp`**, **`/dev`**, **`/mnt`** and **`/var`**.
`/home`	Normally contains one subdirectory for each user account on the system. Every user on a UNIX system should have his or her own account. Along with the login identification and password, the system administrator will also provide you with your own directory. You have complete control over the contents of your own directory. You are responsible for organizing and managing your work by creating subdirectories and files underneath the directory associated with your account. When you log in to the system, initially you will be located in the directory associated with your account. This directory, therefore, is commonly referred to as the *HOME* directory or *login* directory. From here, you can change your position to any other directory in the hierarchy to which you have access. At a minimum, you will be able to access everything beneath your *HOME* directory; at a maximum, you will be able to move to *any* directory in the UNIX system hierarchy (the default). It is up to your system administrator to restrict users' access to specific directories on the system.
`/etc`	Holds many of the system configuration files. These are documented in the reference manual section 4.

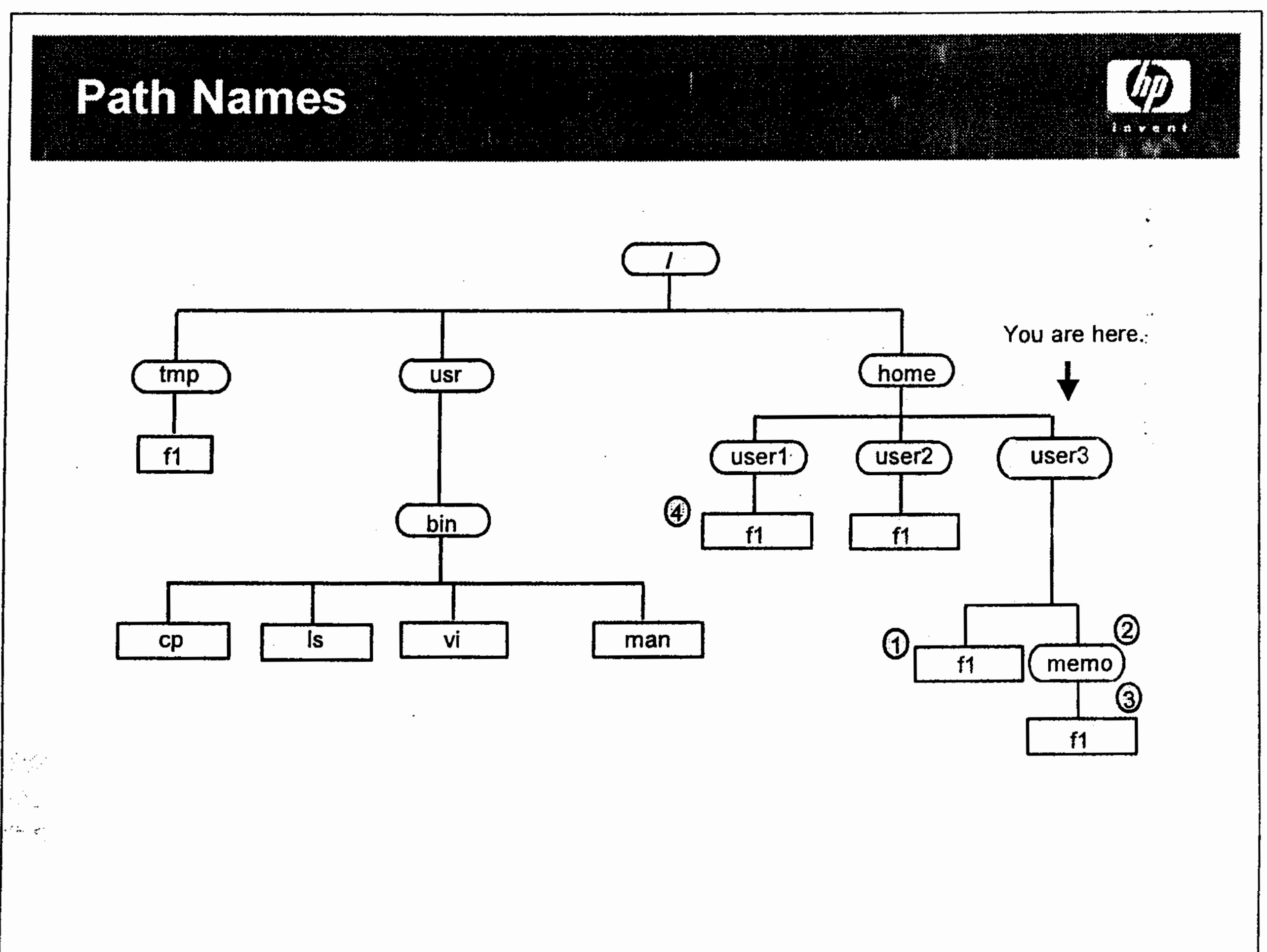
© 2003 Hewlett-Packard Development Company, L.P.

/stand/vmunix	Stores the program that is the UNIX system kernel. This program is loaded into memory when your system is turned on, and controls all of your system operations.
/tmp	Commonly used as a scratch space for operating systems that need to create intermediate or working files. Note: A UNIX system convention defines that files under any directory called **tmp** can be removed at *any time*.
/dev	Contains the files that represent hardware devices that may be connected to your system. Since these files act as a gateway to the device, data will never be directly stored in the device files. They are often referred to as **special files** or **device files**.
/mnt	Used to mount other devices (laserROM for instance).
/var/mail	Contains a "mailbox" for each user who has incoming mail.
/var/news	Contains all of the files representing the current news messages. All their contents can be displayed by entering **news -a**.
/var/tmp	Commonly used as a scratch space for users.

Although the Linux directory hierarchy follows conventions similar to those applied to HP-UX, there are differences. However, most conventional directory names are applied to the Linux directory hierarchy as they are in HP-UX.

© 2003 Hewlett-Packard Development Company, L.P.

4–4. SLIDE: Path Names



Student Notes

Absolute: **Relative to `/home/user3`**

① `/home/user3/f1` ① `f1`

② `/home/user3/memo` ② memo

③ `/home/user3/memo/f1` ③ memo/f1

Relative to `/home/user1`

④ `/home/user1/f1` ④ `f1`

Many UNIX system commands operate on files and/or directories. To inform a command of the location of the requested file or directory, you provide a path name as an argument to the command. A **path name** represents the route through the hierarchy that is traversed to reach the desired file or directory.

```
$ command [options] [pathname pathname ...]
```


© 2003 Hewlett-Packard Development Company, L.P.

To illustrate the concept of path names, we use the analogy of tracing along the branches of the UNIX system tree with a pencil to get from one location to another. The path name will be the list of all directories that the pencil point touches while tracing its way through the hierarchy, concluding with the desired file or directory.

When designating the path name of a file or directory, a forward slash (/) is used to delimit the directory and/or file names.

```
directory/directory/directory

directory/file
```

At all times while you are logged in to a UNIX system you are positioned in some directory in the hierarchy. You are able to change your position to some other directory through UNIX system commands, but you will still always be in some directory. For example, when you log in, initially you will be placed in your *HOME* directory.

File and directory locations can be designated with either an absolute path name or a relative path name.

Absolute Path Name

- gives the complete designation of the location of a file or directory
- always starts at the top of the hierarchy (the root)
- always starts with a /
- is not dependent on your current location in the hierarchy
- is always unique across the entire hierarchy

Absolute Path Name Examples

The following path names designate the location of all files called `f1` in the hierarchy illustrated on the slide. Note that there are many files called `f1`, but they each have a unique absolute path name.

```
/tmp/f1

/home/user1/f1

/home/user2/f1

/home/user3/f1

/home/user3/memo/f1
```

Relative Path Name

- Always starts at your current location in the hierarchy
- Never starts with a /
- Is unique relative to your current location only
- Is often shorter than the absolute path name

© 2003 Hewlett-Packard Development Company, L.P.

Relative Path Name Examples

The following examples are again referencing the files named `f1`, but their relative path designation is dependent on the user's current position in the hierarchy.

Assume current position is **`/home`**:

```
user1/f1

user2/f1

user3/f1

user3/memo/f1
```

Assume current position is **`/home/user3`**:

```
f1

memo/f1
```

Assume current position is **`/home/user3/memo`**:

```
f1
```

Notice that the relative file name, **`f1`**, is not unique. However, the UNIX system knows which one to retrieve, because it knows to retrieve **`/home/user1/f1`** if you are currently located in the directory **`/home/user1`**, or if you are currently located in the directory **`/home/user3/memo`**, it knows to retrieve **`/home/user3/memo/f1`**. Also notice that the relative path name can be much shorter than the absolute path designation. For example, if you are in the directory **`/home/user3/memo`**, you can print **`f1`** with either of the following commands:

Absolute path name	**`lp /home/user3/memo/f1`**
Relative path name	**`lp f1`**

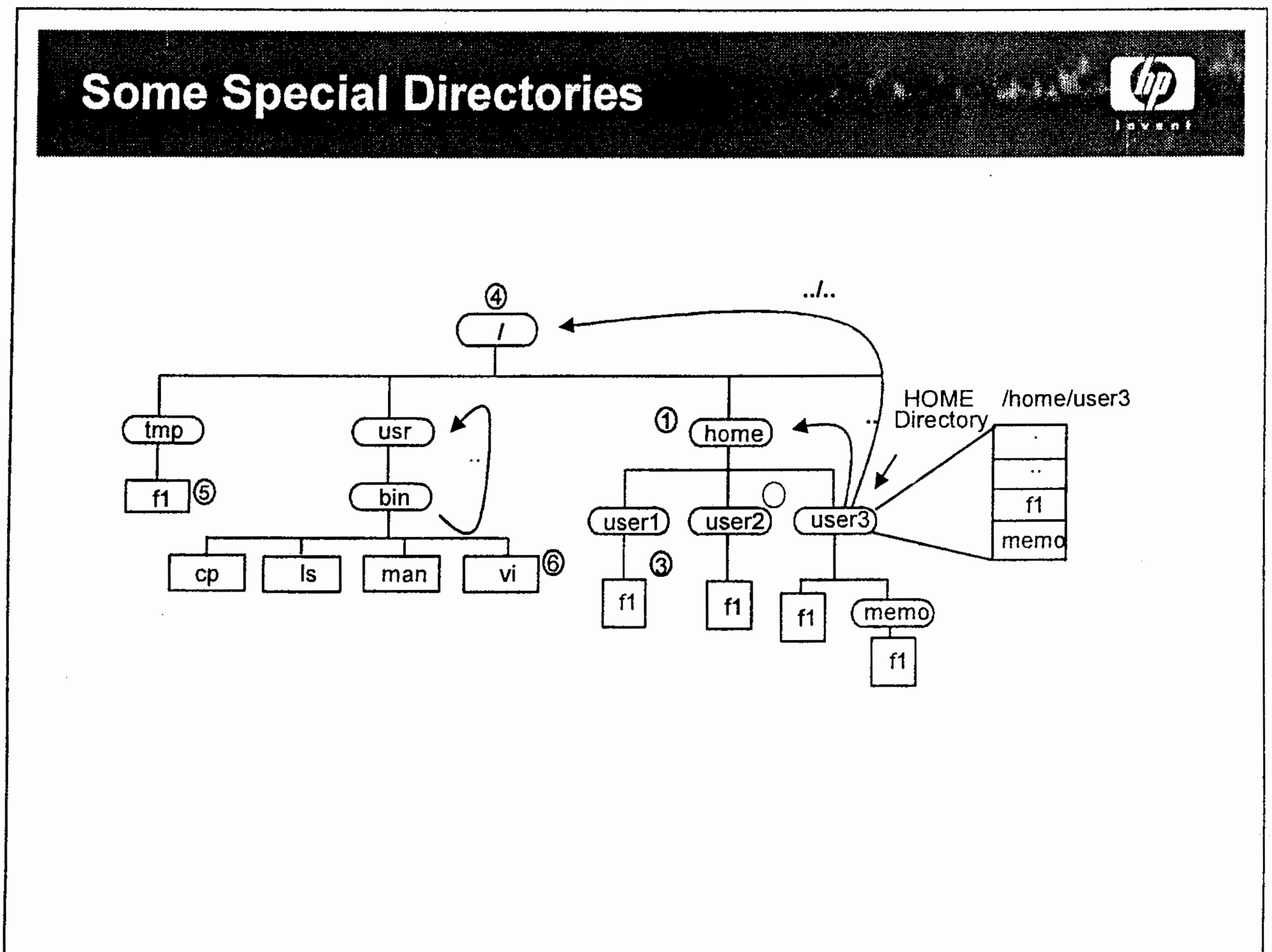
In this case, the relative path name can save you a lot of keystrokes.

NOTE: When accessing files with relative path names, it is important that you know in what directory you are currently located to ensure that you are accessing the correct file, especially if files with the same name exist in more than one directory on the system.

Internally, the UNIX system finds all files or directories by using an absolute path name. This makes sense, because the absolute path name absolutely and uniquely identifies a file or directory (since there is only one root). The UNIX system allows the use of relative path names only as a typing convenience for the user.

© 2003 Hewlett-Packard Development Company, L.P.

4–5. SLIDE: Some Special Directories



Student Notes

Absolute	**Relative to `/home/user3`**
① `/home`	① `..`
② `/home/user2`	② `../user2`
③ `/home/user1/f1`	③ `../user1/f1`
④ `/`	④ `../..`
⑤ `/tmp/f1`	⑤ `../../tmp/f1`
⑥ `/usr/bin/vi`	⑥ `../../usr/bin/vi`

When any directory is created, two entries, called dot (`.`), and dot dot (`..`), are created automatically. These are commonly used when designating relative path names. On the previous slide you may have noticed that the relative path examples could only traverse down through the hierarchy. With `..`, you can traverse up through the hierarchy as well.

© 2003 Hewlett-Packard Development Company, L.P.

Login Directory

When a new user is added to the system, he or she will be assigned a login ID and possibly a password. A directory will be created that the user will own and control. This directory is usually created under the `/home` directory, and has the same name as the user's login ID. The user can then create files and subdirectories under this directory.

When you log into the system, the UNIX system will place you in this directory. This directory is, therefore, referred to as your login directory or your *HOME* directory.

Dot (.)

The entry called `dot` represents your current directory position.

Examples of Dot (.)

If you are currently in the directory `/home/user3`:

`.`	Represents the current directory `/home/user3`
`./f1`	Represents `/home/user3/f1`
`./memo/f1`	Represents `/home/user3/memo/f1`

Dot Dot (..)

The entry called **dot dot** represents the directory immediately above your current directory position, often referred to as the **parent directory**. Every directory can have several files and subdirectories contained within it, but every directory has only one parent directory. Thus, there is no confusion when traversing up the hierarchy.

The root directory (/) is like any other directory and contains entries for both **dot** and **dot dot**. But since the root directory does not have a parent directory, its dot dot entry refers to itself.

Examples of Dot Dot (..)

If you are currently in the directory `/home`:

`..`	represents `/`
`../..`	also represents `/`
`../tmp`	represents `/tmp`
`../tmp/f1`	represents `/tmp/f1`

If you are currently in the directory `/home/user3`:

`..`	represents `/home`
`../..`	represents `/`

© 2003 Hewlett-Packard Development Company, L.P.

`../user2`	represents `/home/user2`
`../user1/f1`	represents `/home/user1/f1`
`../../tmp/f1`	represents `/tmp/f1`

Notice that in the last example, the absolute path is shorter than the relative path in two cases. If the relative path takes you through the root directory, you may just as well use the absolute path instead of the relative path.

© 2003 Hewlett-Packard Development Company, L.P.

4–6. SLIDE: Basic File System Commands

Basic File System Commands

`pwd`	Displays the directory name of your current location in the hierarchy
`ls`	Sees what files and directories are under the current directory
`cd`	Changes your location in the hierarchy to another directory
`find`	Finds files
`mkdir`	Creates a directory
`rmdir`	Removes a directory

Student Notes

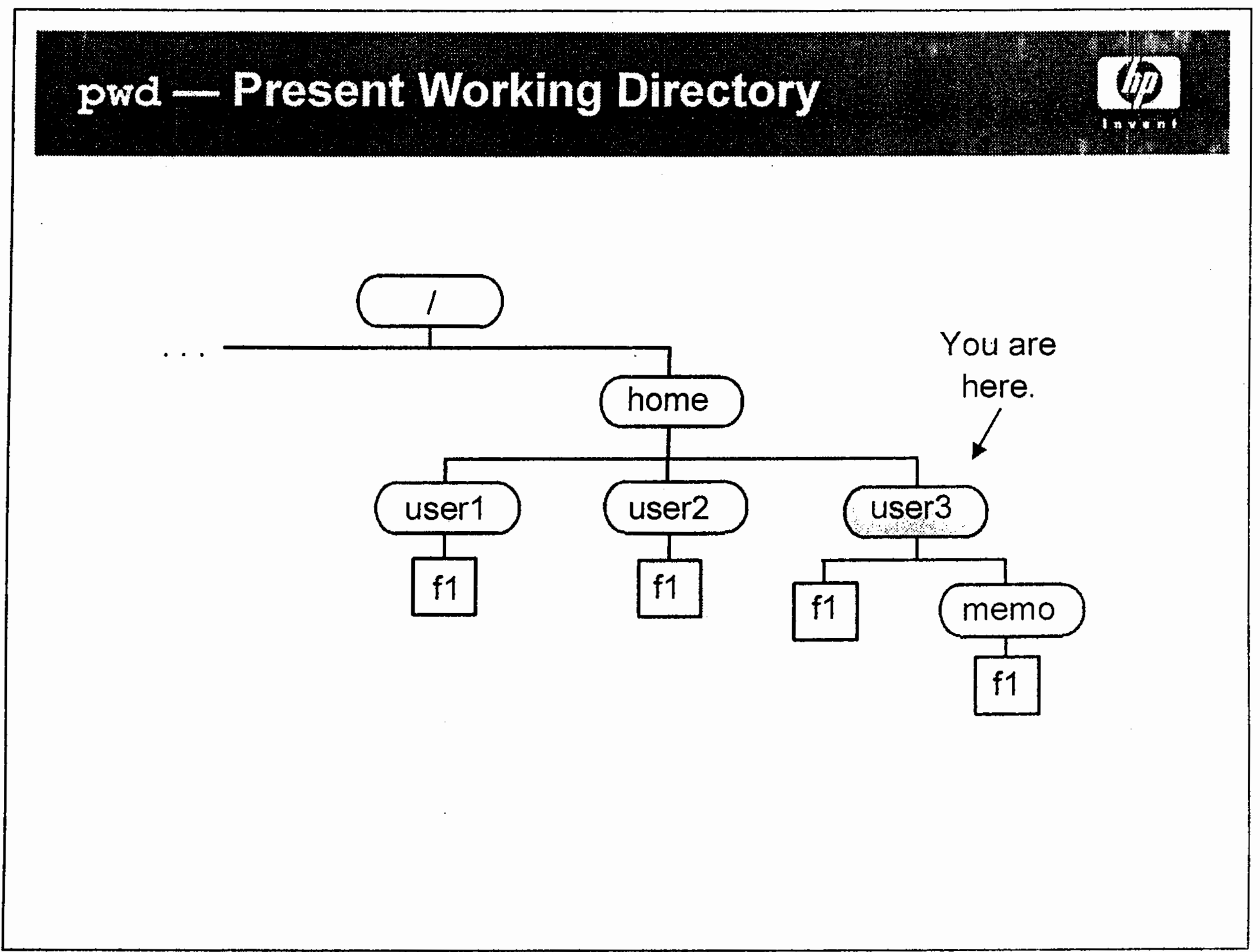
A directory, like a file folder, is a way to organize your files. The remainder of this module introduces basic directory manipulation commands. You will learn to:

- Display the directory name of your current location in the hierarchy.
- See what files and directories are under the current directory.
- Change your location in the hierarchy to another directory.
- Create a directory.
- Remove a directory.

In this module we will not deal with the files within a directory. We will examine directories only.

© 2003 Hewlett-Packard Development Company, L.P.

4–7. SLIDE: pwd — Present Working Directory



Student Notes

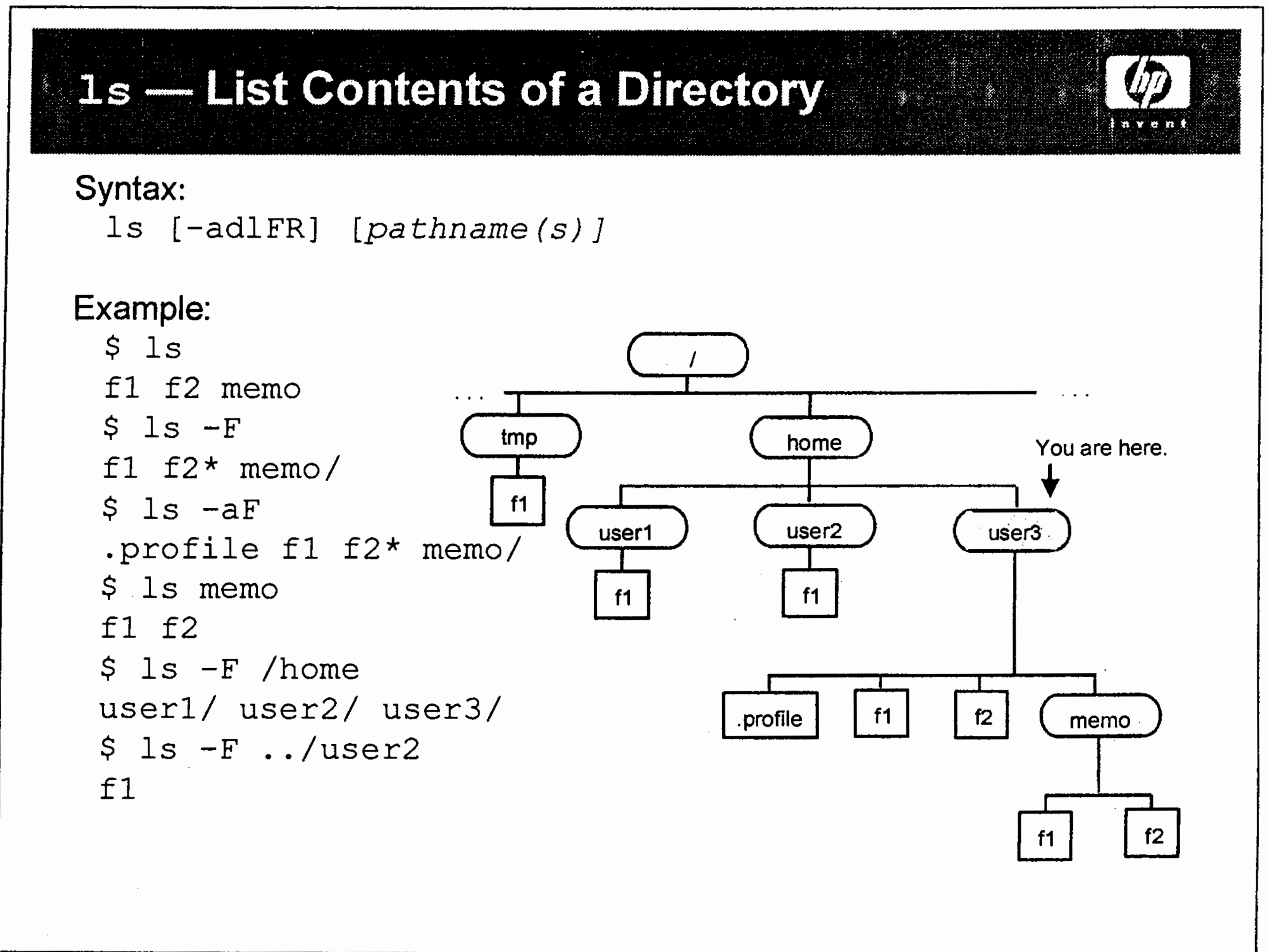
At all times while you are logged in to your UNIX system, you will be positioned in some directory somewhere in the file system hierarchy. The directory you are located in is often referred to as your working directory.

The pwd command reports the absolute path name to your current directory location in a UNIX system's file system. It is a shorthand notation for present working directory.

Since the UNIX system allows you to move very easily through the file system, all users depend on this command to verify their current location in the hierarchy. New users are encouraged to issue this command frequently to display their location as they move through the file system.

© 2003 Hewlett-Packard Development Company, L.P.

4–8. SLIDE: `ls` — List Contents of a Directory



Student Notes

The **`ls`** command is used to list the names of files and directories.

With no arguments, **`ls`** displays the names of the files and directories under the current directory.

`ls` will accept arguments designating a relative or absolute path name of a file or directory. When the path of a file is provided, **`ls`** reports information associated with the designated file. When the path of a directory is provided, **`ls`** displays the contents of the requested directory.

`ls` supports many options. The options cause **`ls`** to provide additional information. Multiple options may be supplied on a single command line to display more complete file or directory information. Some of the more frequently used options are listed on the slide. They are:

`-a` Lists all files, including those whose names start with a dot (`.`). Normally these **dot files** are *hidden*, except when the **`-a`** option is specified. The dot files commonly hold configuration information for your user session or applications.

© 2003 Hewlett-Packard Development Company, L.P.

`-d` Lists characteristics of the directory, instead of the contents of the directory. Often used with `-l` to display the status of a directory.

`-l` Provides a long listing that describes attributes about each file, including type, mode, number of links, owner, group, size (in bytes), the modification date, and the name.

`-F` Appends a slash (/) to each listed file that is a directory and an asterisk (*) to each listed file that is executable.

`-R` Recursively lists files in the given directory and in all subdirectories.

Examples

```
$ pwd
/home/user3

$ ls -F /home                     Absolute path as an argument
user1/ user2/ user3/

$ ls -F ..                        Relative path as an argument
user1/ user2/ user3/

$ ls -F ../user1                  Relative path as an argument
f1

$ ls -l memo                      Relative path of a dir as an argument
-rw-rw-rw- 1 user3 class 27 Jan 24 06:11 f1
-rw-rw-rw- 1 user3 class 37 Jan 23 19:03 f2

$ ls -ld memo                     Display info for directory memo
drwxr-xr-x 2 user3 class 1024 Jan 20 10:23 memo

$ ls -l f1 f2                     Multiple arguments, relative paths of files
-rw-rw-rw- 1 user3 class 27 Jan 24 06:11 f1
-rw-rw-rw- 1 user3 class 37 Jan 23 19:03 f2

$ ls -R                           Recursive listing of subdirectories
memo f1 f2
./memo:
f1 f2

$ ls user2                        user2 does not exist under current directory
user2 not found
```

HP-UX Shorthand Commands

Hewlett-Packard's implementation of the UNIX system provides some shorthand commands for common options used with the **ls** command:

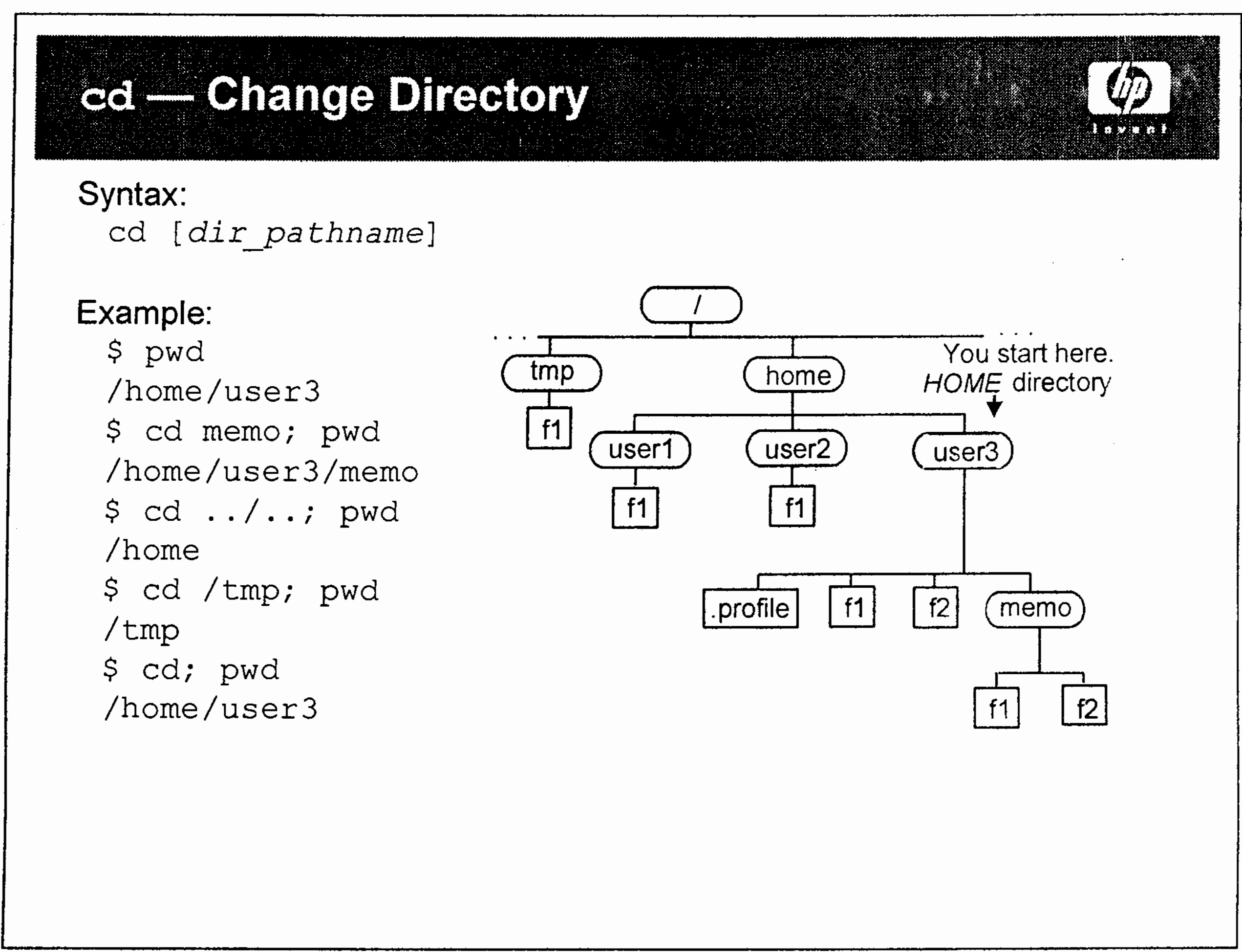
© 2003 Hewlett-Packard Development Company, L.P.

UNIX/Linux System Command	HP-UX Equivalent
`ls -F`	`lsf`
`ls -l`	`ll`
`ls -R`	`lsr`

Linux does not recognize the `lsf`, `ll`, or `lsr` abbreviations that are available in HP-UX. In Linux, you must use the full form of these commands: `ls -F`, `ls -l`, and `ls -R`.

© 2003 Hewlett-Packard Development Company, L.P.

4–9. SLIDE: cd — Change Directory



Student Notes

Think of the tree diagram as a road map showing the location of all of the directories and files on your system. You are always positioned in a directory. The **cd** command allows you to change directory, and move to some other location in the hierarchy.

The syntax is:

```
cd path_name
```

In which *path_name* is the relative or absolute path name of the directory to which you would like to go. When executed with no arguments, the **cd** command will return you to your login or *HOME* directory. If you ever get "lost" in the hierarchy, you can simply execute **cd** and you will be *HOME* again.

NOTE:	When using the **cd** command to move around the hierarchy, be sure to issue the **pwd** command frequently to verify your location in the hierarchy.

© 2003 Hewlett-Packard Development Company, L.P.

POSIX Shell Enhanced cd

The POSIX shell has a memory of your previous directory location. The cd command still changes directories as you would expect, but it has some additional features that will save typing.

The cd command has a memory of your previous directory (stored in the environment variable OLDPWD) and it can be accessed with cd -.

```
$ pwd
/home/user3/tree

$ cd /tmp

$ pwd
/tmp

$ cd -                    Takes you to the previous directory
/home/user3/tree
```


© 2003 Hewlett-Packard Development Company, L.P.

4–10. SLIDE: The `find` Command

The `find` Command

Syntax:

`find path_list expression` Performs an ordered search through the file system. *path_list* is a list of directories to search. *expression* specifies search criteria and actions.

Examples:

```
$ find . -name .profile
./.profile
$
```

Student Notes

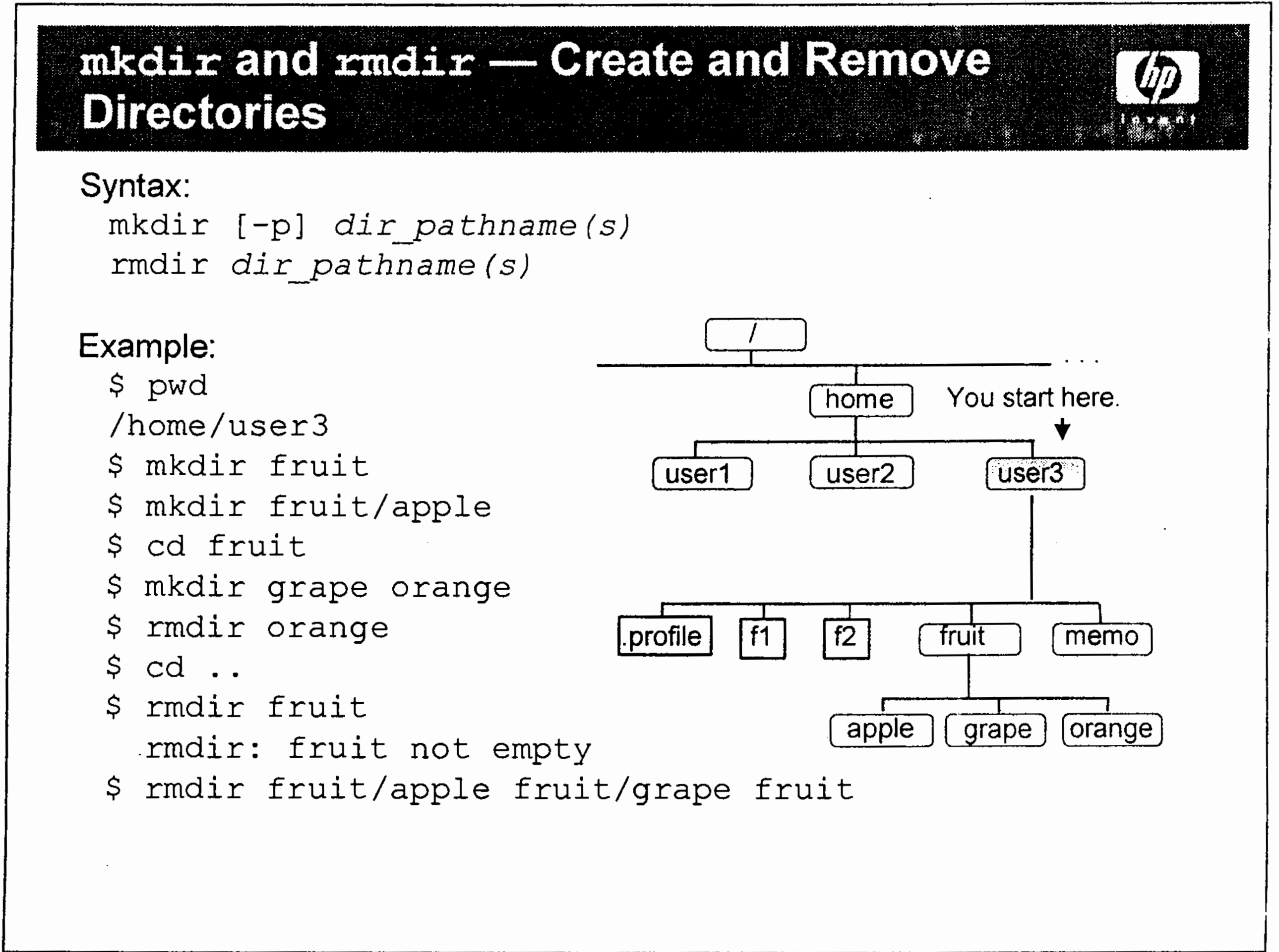
The **`find`** command is the only command that performs an automated search through the file system. It is very slow and uses a lot of the CPU capacity. It should be used sparingly.

The *path_list* is a list of path names, typically from one directory. Often dot (`.`) is specified. The path names are searched recursively for files that satisfy the criteria specified in an **expression**. When **`find`** locates a match, it performs the tasks also specified in the expression. One of the most common tasks is to print the path name to the match.

The expression is made up of keywords and arguments that can specify search criteria and tasks to perform upon finding a match. One of the things that can make `find` complicated is that the keywords used in the expression are all preceded by a hyphen (-), so it looks as if the arguments precede the options.

© 2003 Hewlett-Packard Development Company, L.P.

4–11. SLIDE: mkdir and rmdir — Create and Remove Directories



Student Notes

Create a Directory

The **mkdir** command allows you to make or create a directory. These directories can then be used to help organize our files. When each directory is created, two subdirectories: dot (.) and dot dot (..), representing the current and parent directories, are automatically created. Note that creating directories does not change your location in the hierarchy.

By default, when specifying a relative or absolute path to the directory being created, all intermediate directories must exist. Alternately, you can use the following options:

-p Creates intermediate directories if they do not already exist.

-m *mode* After creating the directory as specified, the file permissions are set to ***mode***.

The following command creates the **fruit** directory, if it does not already exist:

```
$ mkdir -p fruit/apple fruit/grape fruit/orange
```


© 2003 Hewlett-Packard Development Company, L.P.

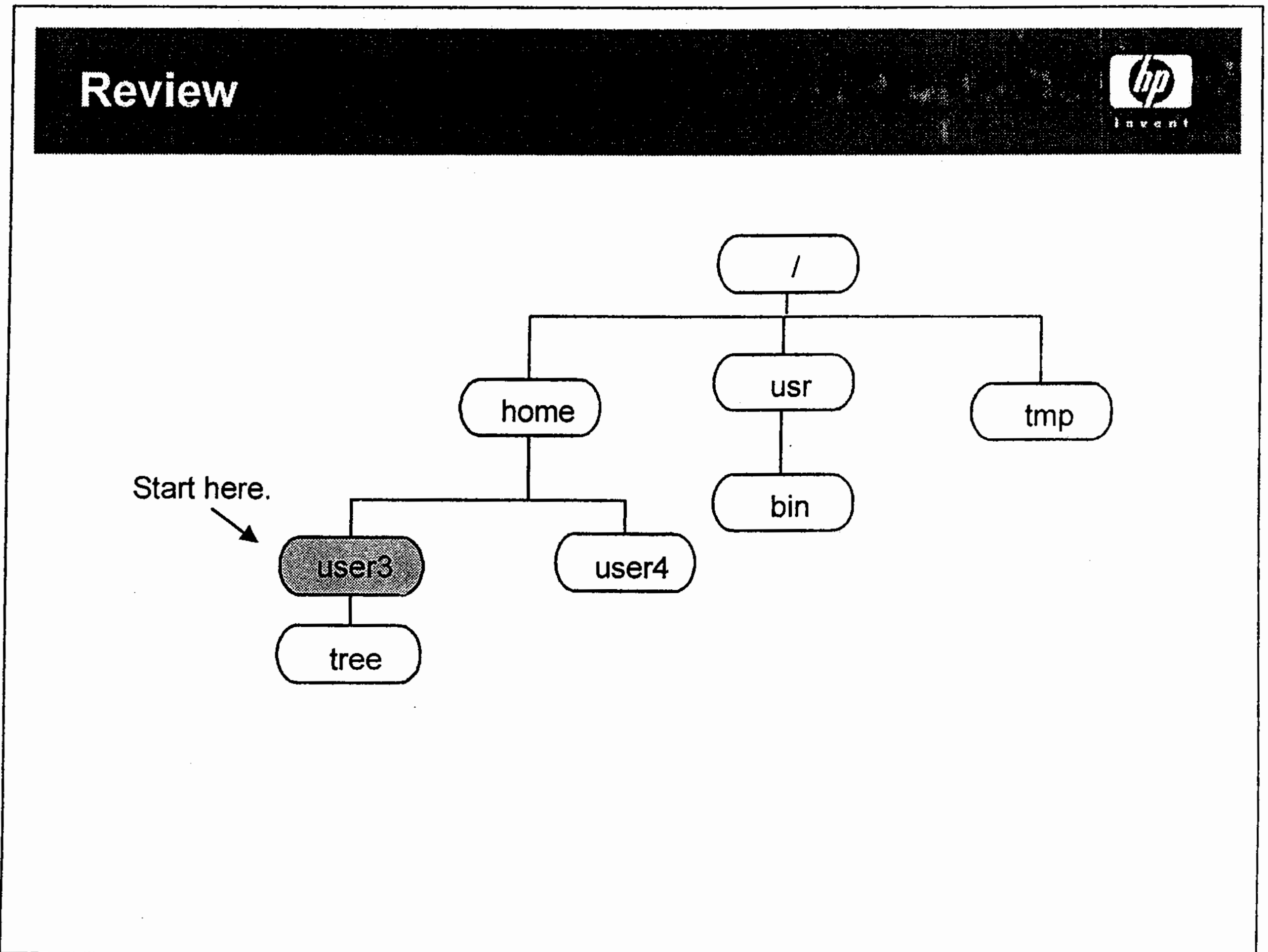
Remove a Directory

The `rmdir` command allows you to remove a directory. To remove a directory, it must be empty (that is, hold no entries except dot and dot dot). You cannot remove a directory that is between your current location and the root directory.

Both commands can take multiple arguments. The arguments to `mkdir` represent the new directory names. The arguments to `rmdir` must be existing directory names. As with any of the commands that take file or directory names as arguments, you can provide absolute or relative path names.

© 2003 Hewlett-Packard Development Company, L.P.

4–12. SLIDE: Review



Student Notes

Work through the examples on the slide to review the use of the **cd** and **pwd** commands and the use of relative and absolute paths.

Using the directory structure on the slide, if you started at the directory **user3**, where would you be after typing each of the following **cd** commands?

```
$ pwd            /home/user3
$ cd ..
$ pwd            ____________________
$ cd usr
$ pwd            ____________________
$ cd /usr
$ pwd            ____________________
$ cd ../tmp
$ pwd            ____________________
$ cd .
$ pwd            ____________________
```


© 2003 Hewlett-Packard Development Company, L.P.

4–13. SLIDE: The File System — Summary

File	A container for data
Directory	A container for files and other directories
Tree	Hierarchical structure of a UNIX system
Path name	Identifies a file's or directory's location in the hierarchy
HOME	Represents the path name of your login directory
`pwd`	Displays your current location in the hierarchy
`cd`	Changes your location in the hierarchy to another directory
`ls`	Lists the contents of a directory
`find`	Finds files specified by options
`mkdir`	Creates directories
`rmdir`	Removes directories

Student Notes

© 2003 Hewlett-Packard Development Company, L.P.

4–14. LAB: The File System

Directions

Complete the following exercises and answer the associated questions.

1. What is the name of your HOME directory?

2. From your HOME directory, find out the entire tree structure rooted at the subdirectory called **`tree`** using the **`ls`** command. Draw a picture of it, marking directories by circling them. Use a separate sheet of paper if you need more space

3. What is the full path name of the file **`labrador`** in the tree drawing from the previous exercise? What is its relative path name from your HOME directory?

4. From your *HOME* directory, change into the **`retriever`** directory. Using a relative path name, change into the **`shepherd`** directory. Again using a relative path name, change into the **`car.models`** directory. Finally, return to your HOME directory. What commands did you use? How did you know if you arrived at each of your destinations?

© 2003 Hewlett-Packard Development Company, L.P.

5. Create a directory in your *HOME* directory called `junk`. Make that directory your current working directory. What commands did you use? What is the full path name of this new directory?

6. From your **HOME** directory, make the following directories with a single command line:

```
junk/dirA/dir1
junk/dirA
junk/dirA/dir2
junk/dirA/dir1/dirc
```

   Did you have any problems? If you encounter any problems, before trying again, remove any directories created as a result of your effort. What single command did you use?

7. From your **HOME** directory, obtain a directory listing of the directory `dirA` under the `junk` directory. Use both relative and absolute path names. What commands did you use?

8. From your **HOME** directory, using only the `rmdir` command, remove all of the subdirectories under the directory `junk`. How is this accomplished using a single `rmdir` command?

9. Return to your **HOME** directory. With one command, display a long listing of the files `cp` and `vi` (from the `/usr/bin` directory). Try to use both absolute and relative path names.

© 2003 Hewlett-Packard Development Company, L.P.

Module 5 — Managing Files

Objectives

Upon completion of this module, you will be able to do the following:

- Use the common UNIX system file manipulation commands.
- Explain the purpose of the line printer spooler system.
- Identify and use the line printer spooler commands used to interact with the system.
- Monitor the status of the line printer spooler system.

© 2003 Hewlett-Packard Development Company, L.P.

5–1. SLIDE: What Is a File?

What Is a File?

A container for data or a link to a device.

- Every file has a name and may hold data that resides on a disk.
- There are several different types of files:
  - Regular files
    - text, data, drawings
    - executable programs
  - Directories
  - Device files

Student Notes

Everything in the UNIX system is a file that includes:

Regular files	Text, mail messages, data, drawings, program source code
Programs	Executable programs such as `ksh`, `who`, `date`, `man`, and `ls`.
Directories	Special files with the name and file system identifier for the files and directories they contain.
Devices	Special files providing the interface to hardware devices such as disks, terminals, printers, and memory.

A **file** is simply a name and the associated data stored on a mass storage device, usually a disk. As far as the UNIX system is concerned, a file is nothing more than a stream of data bytes. There are no predefined records, fields, end-of-record marks, or end-of-file marks. This provides a lot of flexibility for application developers to define their own internal file characteristics.

© 2003 Hewlett-Packard Development Company, L.P.

A **regular file** normally contains ASCII text characters and is typically created using a text editor at a terminal.

A **program file** is a regular file that contains executable instructions. It can include compiled code that cannot be displayed on your terminal (`mail`, `who`, `date`), or it can contain UNIX-system shell commands, commonly referred to as **shell scripts**, which can be displayed to your terminal (`.profile`, `.logout`).

A **directory** is a special file containing the names of the files and directories that it holds. It also stores an **inode number** for every entry, which identifies where file information and data storage addresses can be found in the file system. (Note: It is not a regular text file.)

A **device file** is a special file that provides the interface between the kernel and the actual hardware device. Since these files are for interface purposes, they will never hold any actual data. These files are commonly stored under the `/dev` directory, and will contain a file for each hardware device with which your computer needs to communicate.

© 2003 Hewlett-Packard Development Company, L.P.

5–2. SLIDE: What Can We Do with Files?

`ls`	Look at the characteristics of a file
`cat`	Look at the contents of a file
`more`	Look at the contents of a file, one screenful at a time
`lp`	Print a file
`cp`	Make a copy of a file
`mv`	Change the name of a file or directory
`mv`	Move a file to another directory
`ln`	Create another name for a file
`rm`	Remove a file

Student Notes

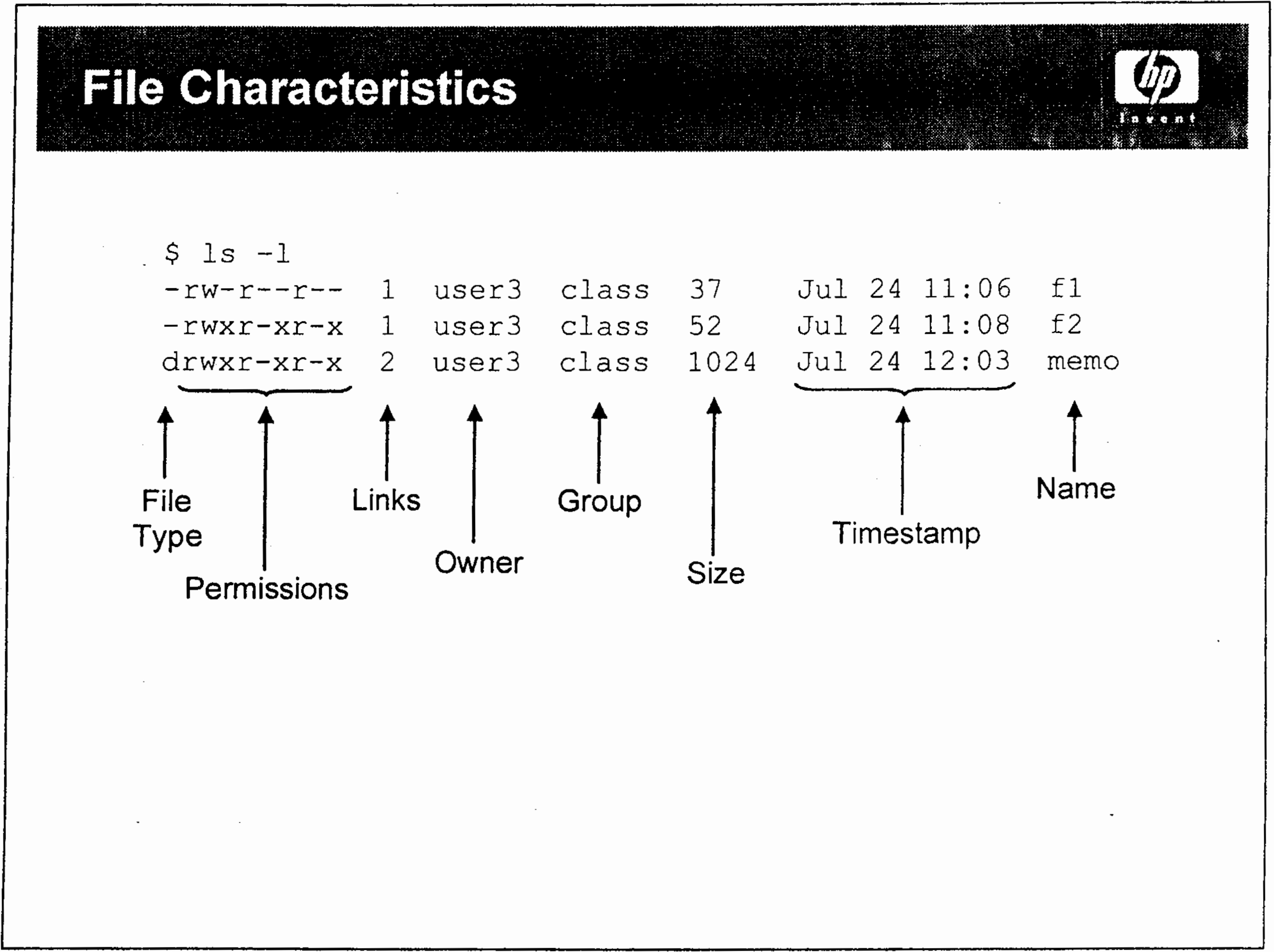
Given that most activity on a UNIX system focuses around files and directories, many commands are available for manipulating files and directories.

You know some introductory directory manipulation commands. In this module we will present additional commands that can be used with files and directories.

You will also need to create files and manipulate their contents. This is commonly done through the use of an editor, such as **`vi`**.

© 2003 Hewlett-Packard Development Company, L.P.

5–3. SLIDE: File Characteristics



Student Notes

A file has several characteristics associated with it. They can be displayed using the `ls -l` command.

Type	Regular file or special file
Permissions or Mode	Access definition for the file
Links	Number of file names associated with a single collection of data
Owner	User identification of file owner
Group	Group identification for file access
Size	Number of bytes file contains
Timestamp	Date file last modified
Name	Maximum of 14 characters (255 characters if long file names are supported)

© 2003 Hewlett-Packard Development Company, L.P.

File Name Specifications

- maximum of 14 characters
- maximum of 255 characters if long file names are supported
- normally contain alpha characters (azAZ), numeric (09), dot (.), dash (-), and underscore(_)

Many of the other characters have a "special" meaning to the shell, such as a *blank space* or the *forward slash*, so you normally cannot include these characters as part of a file name. Other special characters include, *, <, >, \, $, and |. If you try to include these characters in a file name, you often will get unexpected results.

File names that represent two words are often connected with an underscore:

`$ cd a dir`	*Illegal syntax —* `cd` *sees two arguments*
`$ cd a_dir`	*Legal syntax —* `cd` *sees one argument*

In the UNIX system, the dot (.) is a regular character, and, therefore, can appear anywhere (and multiple times) in a file name, making file names `a.bcdefg`, `a.b.c.d` and `a...b` legal.

Dot is somewhat special when it appears as the *first* character of a file name, in which case it designates a *hidden file*. You can display file names containing a leading dot by issuing `ls -a`.

File Types

There are many types of files supported in the UNIX system. The file type is displayed through the first character of the `ls -l` output. The common types include:

-	A regular file
`d`	A directory
`l`	A symbolically linked file
`n`	A network special file
`c`	A character device file (terminals, printers)
`b`	A block device file (disks)
`p`	A named pipe (an interprocess communication channel)

© 2003 Hewlett-Packard Development Company, L.P.

5–4. SLIDE: `cat` — Display the Contents of a File

Student Notes

The **cat** command is used to concatenate and display text files seamlessly. It adds no format to the output of the files, including no delimiter between the end of one file and the beginning of the next. The syntax is

```
cat [file ...]
```

A typical use of the **cat** command is looking at the contents of a single file. For example,

```
cat funfile
```

Writes the content of the file **funfile** to the screen. However, if the file is too big for the terminal's screen, the text will go by too quickly to read. Therefore, we need a more intelligent way to display files to the screen.

When the **cat** command is issued with no arguments, it will wait for input from the keyboard. This works similarly to the **mail** and **write** commands. A [Return], [Ctrl] + [d] must be issued to conclude the input. Once input is concluded, your input text is displayed to the screen.

© 2003 Hewlett-Packard Development Company, L.P.

CAUTION: If the file contains control characters, such as a compiled program, and you `cat` it to your terminal, your terminal may become disabled. Reset your terminal with either of the following methods:

Method 1:

1. Try to log out — press [Return], then issue the `exit` command.
2. Power cycle your terminal. — Turn it off, and then turn it back on.
3. Log back in. — You should be able to log in and continue normally.

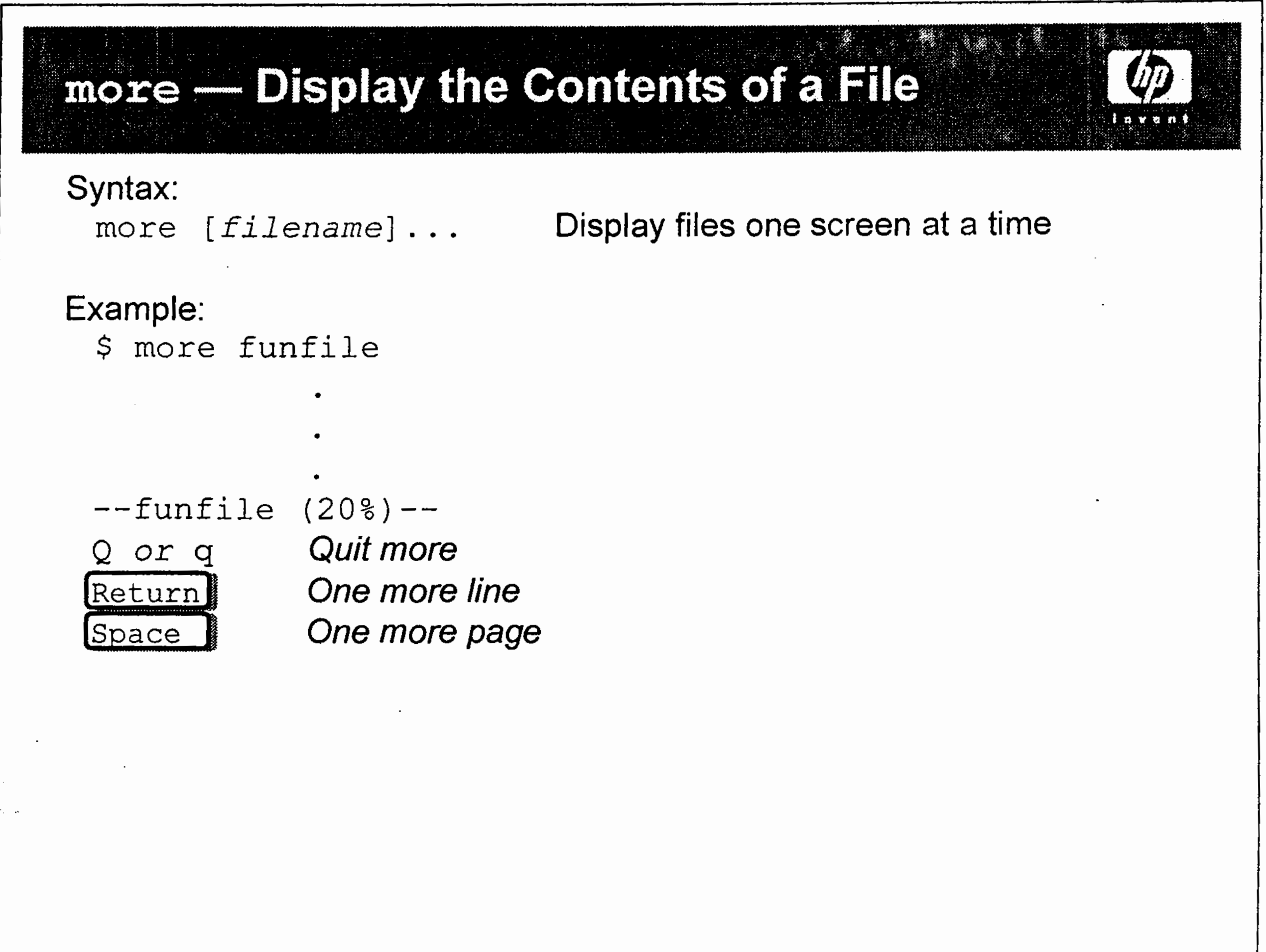
Method 2:

1. Press the [Break] key.
2. Simultaneously press [Shift] + [Ctrl] + [Reset].
3. Press [Return].
4. Issue the command: `tset -e -k`.
5. Issue the command: `tabs`.

Otherwise, your system administrator (or instructor) may have to cancel your terminal session.

© 2003 Hewlett-Packard Development Company, L.P.

5–5. SLIDE: more — Display the Contents of a File



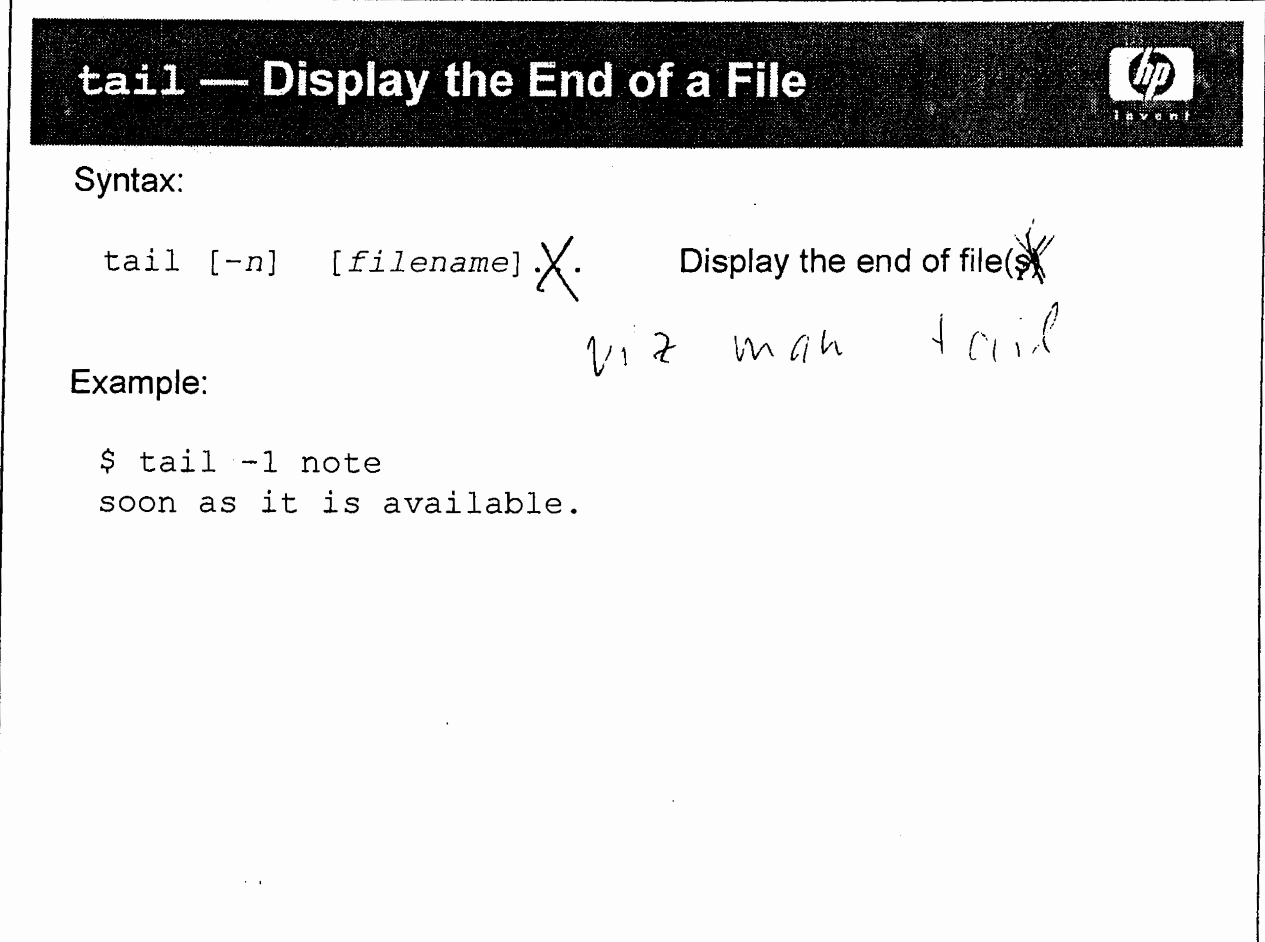
Student Notes

The **more** command prints out the contents of the named files. It will print only one screen of text at a time. To see the next screen of text, press the Space key. To see the next line, press the Return key. To quit from the **more** command, use the q key.

The **more** command supports many other features. Refer to the manual page for an explanation of other available capabilities.

© 2003 Hewlett-Packard Development Company, L.P.

5–6. SLIDE: `tail` — Display the End of a File



Student Notes

The `tail` command is useful for displaying the last ***n*** lines of a file. (Note: *n* defaults to 10 if it is not supplied.) This is especially useful for long log files that are periodically being appended to. With the `tail` command, you can go immediately to the last messages logged instead of scrolling through the entire file with `cat` or `more`.

© 2003 Hewlett-Packard Development Company, L.P.

5–7. SLIDE: The Line Printer Spooler System

The Line Printer Spooler System

- The lp spooler system is a utility that coordinates printer jobs.
- Allows users to:
  - Queue files to printers.
  - Obtain status of printers and print queues.
  - Cancel any print job.

User1 — Output →
User2 — Output →
User3 — Output →
•
•
•
User*n* — Output →
LP Spooler System → Printer

Student Notes

The UNIX operating system provides a utility called the **line printer spooler** (or lp spooler), which is used to configure and control printing on your system. The lp spooler is a mechanism that accepts print requests from all of the users on the system, then configures the printer appropriately and prints the requests one at time. Think of the problems we would have if we did not have a spooler. Every time a user wants to print a file, he or she would have to make sure that no one else is currently printing a file. Two users cannot print to the same printer at the same time.

The lp spooler system has many features that allow for smooth running with minimum administrator intervention. You submit your print requests to the lp spooler system, where they will wait in a queue to be printed. You can check which files are queued and the status of the system. You can also cancel a queued printing request if you decide it should not be printed.

© 2003 Hewlett-Packard Development Company, L.P.

5–8. SLIDE: The `lp` Command

The `lp` Command

- Queues files to be printed.
- Assigns a unique ID number.
- Many options are available for customizing routing and printing.

Syntax: `lp [ -dprinter] [-options]  filename ...`

Example:

```
$ lp report
request id is dp-112 (1 file)
$ lp -n2 memo1 memo2
request id is dp-113 (2 files)
$ lp -dlaser -t"confidential" memo3
request id is laser-114 (1 file)
$
```

Student Notes

The **lp** command allows the user to queue files for printing. A unique job identification number (called a request ID) is given to each request submitted using **lp**.

lp will queue a file to be printed or it will read standard input.

The simplest use of **lp** is to give it a file name as an argument. It will then queue the file to be printed on the default printer.

The **lp** command has a number of options available, which allows you to customize the routing and printing of your jobs.

© 2003 Hewlett-Packard Development Company, L.P.

The syntax of the `lp` command is:

```
lp [-ddest] [-nnumber] [-ooption] [-t title] [-w][file...  ]
```

Some options to `lp` are:

-n*number*	Print ***number*** copies of the request (default is 1).
-d*dest*	***dest*** is the name of the printer on which the request will be printed.
-t*title*	Print ***title*** on the banner page of the printout. The banner page is a header page that identifies the owner of the printout.
-o*option*	Specify printing options specific to your printer, such as font, pitch, density, raw (for graphics dumps), and so on.
-w	Write a message to the user's terminal after the files have been printed.

See the **lp(1)** man page for a complete listing of available options.

The first example on the slide shows the simplest form of **lp**. We are sending the file **report** to the system default printer. **lp** returns the request ID and the number of files submitted to the queue. Here, the file **report** has been sent to printer "dp," and the job is queued with request ID **dp-112**.

In the second example, we are sending **memo1** and **memo2** to be printed, and we want two copies (**-n2**).

In the third example, using the **-d** option, you can specify the printer to which your request will be sent. The output will be titled "confidential."

The Linux system makes use of the BSD print spooler and associated commands. The HP-UX and Linux print-related commands are shown in a comparison table, later in this module.

To print a file, the Linux **lpr** command is used. For example:

```
$ lpr funfile
```

By default, a printer named **lp** is the system default. Alternately, the system administrator can set the name of the default printer on a user-specific basis by configuring that user's login environment controls in the appropriate manner.

If a user wishes to print to an alternate printer, the printer name must be specified, along with the **-P** option.

© 2003 Hewlett-Packard Development Company, L.P.

For example:

```
$ lpr -Pnewprinter funfile
```

NOTE: There must be no space between the **-P** option and the name of the printer.

For more details on the **lpr** command, refer to the appropriate manual pages.

© 2003 Hewlett-Packard Development Company, L.P.

5-9. SLIDE: The `lpstat` Command

The `lpstat` Command

Syntax:

```
lpstat [-t]
```

- `lpstat` reports the requests that you have queued to be printed.
- `lpstat -t` reports the status of the scheduler, default printer name, device, printer status, and all queued print requests.

Student Notes

The **`lpstat`** command reports the status of the various parts of the lp spooler system. **`lpstat`**, when it is used with no options, reports the requests that you currently have queued to be printed.

The **`-t`** option prints all of the status information about all of the printers on the system.

© 2003 Hewlett-Packard Development Company, L.P.

The output of the **`lpstat -t`** command tells us several things:

```
$ lpstat
rw-55          john       4025    Jul 6 14:26:33 1994
$
$ lpstat -t
scheduler is running
system default destination: rw
device for rw: /dev/lp2235
rw accepting requests since Jul 1 10:56:20 1994
printer rw now printing rw-55. enabled since Jul 4 14:32:52 1994
rw-55          john       4025    Jul 6 14:26:33 1994 on rw
rw-56          root        966    Jul 6 14:27:58 1994
$
```

`scheduler is running`	The **scheduler** is the program that sends your print requests to the proper printer. Nothing will print if the scheduler is not running.
`system default destination: rw`	**rw** is the name of the default system printer. If you use **`lp`** without the **`-d`** ***`printer`*** option, your request will be sent to the printer named **rw**. Note that your default system printer will probably have a different name (such as **lp**).
`device for rw: /dev/lp2235`	This tells the spooler where the printer is connected to the computer.
`rw accepting requests`	This means that the spooler will let you queue files to **rw**.
`printer rw now printing rw-55`	Request ID rw-55 currently is being printed.
`enabled`	Requests can be printed on **rw**. If a printer is **disabled** you can submit requests, but they will not be printed until the printer is **enabled** again.

The rest of the lines are the requests to be printed. These fields list the request ID, followed by the user making the request, the size of the request, and the date the request was made.

© 2003 Hewlett-Packard Development Company, L.P.

Linux Systems

The Linux equivalent to **lpstat** is the **lpc** command. The **lpc** command can be executed only by the administrative user (*called* **root**).

To see the contents of the default print spool queue, use the command **lpq**.

```
[user1]$ lpq
Rank    Owner       Job  Files                              Total Size
1st     user1       45   funfile                            517 bytes
```

The listing will show all print jobs in the queue for the user's default printer (or system default, if no default has been set for that user).

To list queued jobs for other printers, the name of the printer must be specified on the command line:

```
[user1]$ lpq matrix1
Rank    Owner       Job  Files                              Total Size
```

If no print jobs exist in the queue for the named printer, only the headings are displayed, as shown in the example above.

© 2003 Hewlett-Packard Development Company, L.P.

5–10. SLIDE: The `cancel` Command

The `cancel` Command

Syntax:

```
cancel id  [ id ... ]
cancel  printer  [  printer ... ]
```

Examples:

- Cancel a job queued by lp.
  `$ cancel dp-115`
- Cancel the current job on a specific printer.
  `$ cancel laser`

Student Notes

The **`cancel`** command is used to remove requests from the print queue. By canceling the current job on the printer, the next request can be printed. You may want to cancel a request if it is extremely long or if someone tried to print a binary file by mistake (such as **`/usr/bin/cat`**). Remember, **`lp`** normally prints text files. Anything else will just confuse the printer and waste piles of paper if you do not specify the appropriate options (such as **`-oraw`** for graphics dumps).

To cancel a request, you must tell the spooler which request to cancel by giving the **`cancel`** command an argument. Arguments to the cancel command can be of two types.

- a request ID (as given by **`lp`** or **`lpstat`**)
- a printer name

Giving the **`cancel`** command a request ID cancels that specific print request. If you give **`cancel`** a printer name, the current job being printed on that printer will stop, and the next request in the queue will start printing.

```
$ lpstat
```


© 2003 Hewlett-Packard Development Company, L.P.

```
rw-113          mike    6275  Jul 6 18:46 1994
rw-114          mike    3349  Jul 6 18:48 1994
rw-115          mike    3258  Jul 6 18:49 1994
$ cancel rw-115
request "rw-115" canceled
$ lpstat
rw-113          mike    6275  Jul 6 18:46 1994
rw-114          mike    3349  Jul 6 18:48 1994
$ cancel rw
request "rw-113" canceled
$ lpstat
rw-114          mike    3349  Jul 6 18:48 1994
```

Any user can execute this command to cancel any request. You can even cancel another user's request; however, mail will be sent to the person whose job was canceled with the name of the user who canceled it. The system administrator can restrict users to canceling only their own requests.

© 2003 Hewlett-Packard Development Company, L.P.

Linux Systems

The Linux equivalent to **cancel** is the **lprm** command. The **lprm** command allows the removal of a print job only by its owner. One user cannot remove another user's print job(s) from the queue. The only exception to this rule is the **root** (*administrative*) user.

The print job number must be specified whenever a print job is to be removed. More than one print job number can be specified on the same command line.

```
[user1]$ lpq
Rank   Owner      Job  Files                          Total Size
1st    user1      45   funfile                        517 bytes
2nd    user1      46   /etc/passwd                    1904 bytes
[user1]$
[user1]$ lprm 45
dfA045ABQzluw dequeued
cfA045ABQzluw dequeued
[user1]$
[user1]$ lpq
Warning: lp is down: printing disabled
Warning: no daemon present
Rank   Owner      Job  Files                          Total Size
1st    user1      46   /etc/passwd                    1904 bytes
```

HP-UX and Linux Printing Commands — Comparison

HP-UX	Linux
lp	lpr
lpstat	lpq
cancel	lprm

Refer to the appropriate Linux manual pages for further details. For example:

```
$ man lpr
...
NAME
     lpr - off line print

SYNOPSIS
     lpr [-Pprinter] [-#num] [-C class] [-J job] [-T title] [-U user]
     [-i [numcols]] [-1234 font] [-wnum] [-cdfghlnmprstv] [name ...
...
```


© 2003 Hewlett-Packard Development Company, L.P.

5-11. SLIDE: cp — Copy Files

cp — Copy Files

Syntax:

`cp [-i] file1 new_file`	Copy a file
`cp [-i] file [file...] dest_dir`	Copy files to a directory
`cp -r [-i] dir [dir...] dest_dir`	Copy directories

Example:

```
$ ls -F
f1  f2* memo/  note  remind
$ cp f1 f1.copy
$ ls -F
f1  f1.copy f2* memo/ note remind
$ cp note remind memo
$ ls -F memo
note  remind
```

Student Notes

The **cp** command is used to make a duplicate copy of one or more files. The following are some considerations when using the **cp** command:

- It requires *at least two arguments* — the source *and* the destination.
- Relative or absolute path names can be used for any of the arguments.
- When copying a single file, the destination can be a path to a file or a directory. If the destination is a file, and the file does not exist, it will be created. If the destination file does exist, its contents will be replaced by the source file. If the destination is a directory, the file will be copied to the directory and retain its original name.
- The `-i` (interactive) option will warn you if the destination file exists, and require you to verify that the file should be copied over.

`$ cp f1 f1.copy`	*Creates a file under current directory called* **`f1.copy`**
`$ cp f1 memo`	*Creates a file under memo called* **`f1`**
`$ cp f1 memo/f1.copy`	*Creates a file under* **memo** *called* **`f1.copy`**

© 2003 Hewlett-Packard Development Company, L.P.

- When copying multiple files, the destination *must* be a directory.

  ```
  $ cp note remind memo
  ```

- A file cannot be copied onto itself.

  ```
  $ cp f1 f1
  cp: f1 and f1 are identical
  ```

- A directory can be copied using the **-r** (recursive) option.

CAUTION: By default, **cp** will copy over existing files — without warning!

```
$ cp f1 note
$ cat f1
This is a sample file to be copied.
$ cat note
This is a sample file to be copied.
```

© 2003 Hewlett-Packard Development Company, L.P.

5–12. SLIDE: mv — Move or Rename Files

mv — Move or Rename Files

Syntax:

```
mv [-i] file new_file                   Rename a file
mv [-i] file [file...] dest_dir         Move files to a directory
mv [-i] dir [dir...] dest_dir           Rename or move directories
```

Example:

```
$ ls -F
f1  f2* memo/ note remind
$ mv f1 file1
$ ls -F
file1  f2* memo/ note remind
$ mv f2 memo/file2
$ ls -F
file1 memo/ note remind
$ ls -F memo
file2*
```

```
$ mv note remind memo
$ ls -F
file1    memo/
$ls -F memo
file2* note remind
$ mv memo letters
$ ls -F
file1    letters/
```

Student Notes

The **mv** command is used to rename a file or move one or more files to another directory. The following are some considerations when using the **mv** command:

- It requires *at least two arguments* — the source *and* the destination.
- Relative or absolute path names can be used for any of the arguments.
- When renaming a single file, the destination can be a path to a file or a directory. If the destination is a file under the current directory, the file will simply be renamed. If the destination is a directory, the source will be moved to the requested directory. The file will be created if it does not exist.
- If the destination file name already exists, its destination's contents will be replaced by the source file. If the destination is a directory, the file will retain its original name and be moved to that directory.

© 2003 Hewlett-Packard Development Company, L.P.

- The `-i` (interactive) option will warn you if the destination file or directory exists, and require you to verify that the file or directory should be overwritten.

`$ mv f1 file1`	*Renames* ***f1*** *to* ***file1*** *under the current directory*
`$ mv file1 memo`	*Moves* ***file1*** *to the* ***memo*** *directory*
`$ mv f2 memo/file2`	*Moves* ***f2*** *to the* ***memo*** *dir and renames it* ***file2***

- When moving multiple files, the destination *must* be a directory.

```
$ mv note remind memo
```

- When the source is a directory, it will be renamed to the destination name.

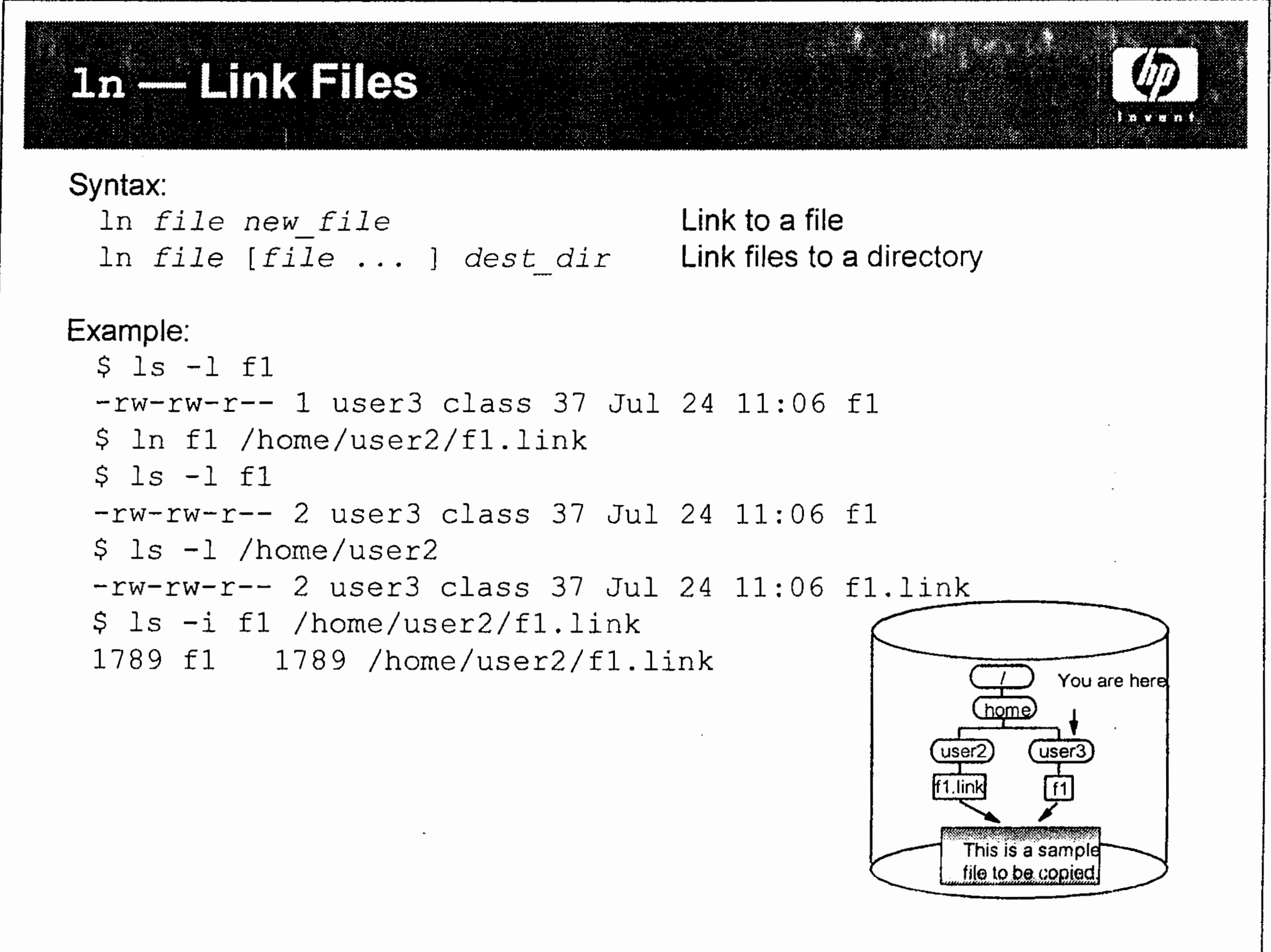
```
$ mv note letter
```

CAUTION: By default, **mv** will move or rename existing files — with no warning!

```
$ mv file1 note
$ cat file1
cat: cannot open file1
$ cat note
This is a sample file to be copied.
```

© 2003 Hewlett-Packard Development Company, L.P.

5–13. SLIDE: ln — Link Files



Student Notes

Links provide a mechanism for multiple file names to reference the same data on the disk. They are useful when many users want to share a file, but prefer to have the file entry under their own directory. If **user3** modifies **f1**, **user2** will see those changes the next time he or she accesses **f1.link**.

CAUTION: The UNIX system does not prohibit more than one user to access and modify a file at the same time. Each user will have a private image to which to make modifications, but the last user to save his or her file to disk will define the version that is stored on the disk. It is up to an application to notify a user that a file is already open for modification, and possibly prohibit additional users from accessing files that are already open.

When many files are linked together, the link count displayed with `ls -l` will be greater than 1. If any of the links are removed, the only effect is to reduce the link count. The file contents are maintained until the link count is reduced to zero, at which time the disk space is released.

© 2003 Hewlett-Packard Development Company, L.P.

Example

```
$ ls -l f1
-rw-rw-r-- 1 user3 class      37   Jul 24 11:06 f1
$ ln f1 /home/user2/f1.link
$ ls -l f1
-rw-rw-r-- 2 user3 class      37   Jul 24 11:06 f1
$ ls -l /home/user2
-rw-rw-r-- 2 user3 class      37   Jul 24 11:06 f1.link
$ ls -i f1 /home/user2/f1.link
1789 /home/user2/f1.link           1789 f1
```


© 2003 Hewlett-Packard Development Company, L.P.

5–14. SLIDE: rm — Remove Files

rm — Remove Files

Syntax:

```
rm [-if] filename [filename...]   Remove files
rm -r[if] dirname [filename...]   Remove directories
```

Examples:

```
$ ls -F
f1 f2 fruit/ memo/
$ rm f1
$ ls -F
f2 fruit/ memo/
$ rm -i f2
f2: ? (y/n)
$ rm fruit
rm: fruit directory
$ rm -r fruit
```

Student Notes

The **rm** command is used to remove files. The files are irretrievable once they are removed. The **rm** command must have at least one argument (a file name) and can accept many. If more than a single file name is given, all of the specified file names will be removed.

The slide shows the most commonly used options.

-f Forces the named files to be removed — no notice will be given to the user, even if an error occurs.

-r Recursively removes the contents of any directories named on the command line.

-i Interrogate or interactive mode, which requires that the user confirm that the removal be completed. You respond with either **y** for yes or **n** for no. Entering a Return is the same as answering no.

© 2003 Hewlett-Packard Development Company, L.P.

CAUTION:	Always use the `-r` option with extreme care. Used incorrectly, this could remove *ALL* of your files. Once a file is removed, it can be restored only from a tape backup. If you must use the `-r` option, use it with the `-i` option. For example, **`rm -ir dirname`**

© 2003 Hewlett-Packard Development Company, L.P.

5–15. SLIDE: File/Directory Manipulation Commands — Summary

File/Directory Manipulation Commands — Summary

`ls -l`	Display file characteristics
`cat`	Concatenate and display contents of files to screen
`more`	Format and display contents of files to screen
`tail`	Display the end of files to screen
`cp`	Copy files or directories
`mv`	Move or rename files or directories
`ln`	Link file names together
`rm`	Remove files or directories
`lp`	Send requests to a line printer
`lpstat`	Print spooler status information
`cancel`	Cancel requests in the line printer queue

Student Notes

© 2003 Hewlett-Packard Development Company, L.P.

5–16. LAB: File and Directory Manipulation

Directions

Complete the following exercises and answer the associated questions.

File Manipulation

1. In your HOME directory, use the `cat` command to display the contents of the file `funfile`. What do you notice? What alternate command provides scrolling control when displaying the contents of a file?

2. Use the `more` command to display the contents of the directory called `tree`. What do you notice? What command do you use to see the contents of a directory?

3. Use the `more` command to display the file `/usr/bin/ls`. What do you notice? Display the contents of `/usr/bin/ls` with the `cat` command. What happens?

4. Go to your HOME directory. Copy the file called `names` to a file called `names.cp`. List the contents of both files to verify that their contents are the same.

© 2003 Hewlett-Packard Development Company, L.P.

5. If the file **names** is modified, will it affect the file **names.cp**? Modify the file **names** by copying the file **funfile** to the file **names**. What happened to the file **names** and the file **names.cp**?

6. How do you restore the file **names**? Issue the command to restore **names**.

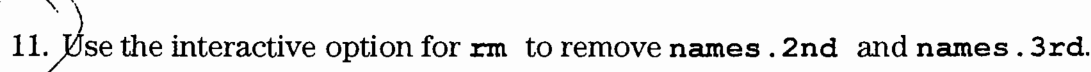
7. Make another copy of the file **names** called **names.new**. Change the name of **names.new** to **names.orig**.

8. How do you create two files (called **names.2nd** and **names.3rd**) that reference the contents of the file **names**?

9. If you modify the contents of **names**, will the contents of **names.2nd** and **names.3rd** be affected? Copy the file **funfile** to the file **names**, and do a long listing of all of your **names** files. Is **names.orig** affected? **names.2nd**? **names.3rd**?

10. Remove the file **names**. What happens to **names.2nd** and **names.3rd**?

© 2003 Hewlett-Packard Development Company, L.P.



11. Use the interactive option for `rm` to remove `names.2nd` and `names.3rd`.

12. Copy the file `names.orig` back to `names`.

Directory Manipulation

1. Make a directory called `fruit` under your `HOME` directory. With one command, move the following files, which are also under your `HOME` directory, to the `fruit` directory:
   lime
   grape
   orange

2. Move the following files, also found under your `HOME` directory, to the `fruit` directory. Their destination names will be as specified below:

Source	Destination
apple	APPLE
peach	Peach

3. Look at the `tree` directory structure in your `HOME` directory. It requires a little organization.

© 2003 Hewlett-Packard Development Company, L.P.

Move the files `collie` and `poodle`, so that they are under the `dog.breeds` directory.
Move the file `probe` under the `sports` directory.
Move the file `taurus` under the directory `sedan`.
Create a new directory under `tree`, called `horses`.
Copy the `mustang` file to the `horses` directory you just created.
Move the file `cherry` to the `fruit` directory you created in the previous exercise.

HINT: You can make these changes from any directory, but what directory do you think you should be in?

4. Move the `fruit` directory from your `HOME` directory to the `tree` directory.

5. Make the `fruit` directory your current working directory. Move the files `banana` and `lemon` to the `fruit` directory. HINT: Remember dot dot (`..`) represents the parent directory and dot (`.`) represents your current directory.

Scavenger Hunt (Optional)

1. Under your `HOME` directory you will find the first clue for the scavenger hunt in the file `scaveng.README`. The hunt will involve files and directories found in the `scavenger` directory underneath the `HOME` directory. Underneath the `scavenger` directory are `north`, `south`, `east`, and `west` subdirectories. Under these are `1_mile`, `2_mile`, `3_mile` subdirectories. Clues are available under each of these directories to the secret code word. For example, if the clue is "go east 3 miles," you go to the `east/3-mile` subdirectory where you will find a file called `README`. This file will give you the next clue. You continue through the clues until you obtain the secret code word. Good luck!

© 2003 Hewlett-Packard Development Company, L.P.

Printing Files - HP-UX

1. List the current status of the printers in the `lp` spooler system and find the name of the default printer.

2. Send the file named `funfile` to the line printer. Make a note of the request ID that is displayed on your terminal.

3. Verify that your requests are queued for printing.

4. How can you tell what files other users are printing? Try it.

5. Use the `cancel` command to remove your requests from the line printer system queue. Confirm that they were canceled.

© 2003 Hewlett-Packard Development Company, L.P.

Printing Files - Linux

1. List the current status of the print queue in the print spooler system.

2. Send the file named **`funfile`** to the line printer.

3. Verify that your requests are queued for printing.

4. How can you tell what files other users are printing? Try it.

5. Use the **`lprm`** command to remove your requests from the line printer system queue. Confirm that they were canceled.

© 2003 Hewlett-Packard Development Company, L.P.

© 2003 Hewlett-Packard Development Company, L.P.

Module 6 — File Permissions and Access

Objectives

Upon completion of this module, you will be able to do the following:

- Describe and change the owner and group attributes of a file.
- Describe and change the permissions on a file.
- Describe and establish default permissions for new files.
- Describe how to change user and group identities.

© 2003 Hewlett-Packard Development Company, L.P.

6–1. SLIDE: File Permissions and Access

File Permissions and Access

Access to files is dependent on a user's identification and the permissions associated with a file. This module will show how to understand the read, write, and execute access to a file

`ls (ll, ls -l)`	Determine what access is granted on a file
`chmod`	Change the file access
`umask`	Change default file access
`chown`	Change the owner of a file
`chgrp`	Change the group of a file
`su`	Switch your user identifier
`newgrp`	Switch your group identifier

Student Notes

Every file on the system is owned by a user. The owner of a file has ultimate control over who has access to it. The owner has the power to allow or deny other users access to files that he or she owns.

© 2003 Hewlett-Packard Development Company, L.P.

6–2. SLIDE: Who Has Access to a File?

Who Has Access to a File?

- The UNIX system incorporates a three-tier structure to define who has access to each file and directory:

user	The owner of the file
group	A group that may have access to the file
other	Everyone else

- The `ls -l` command displays the owner and group who has access to the file.

```
$ ls -l
-rw-r--r-- 1   user3   class      37  Jul 24 11:06 f1
-rwxr-xr-x 1   user3   class      37  Jul 24 11:08 f2
drwxr-xr-x 2   user3   class    1024  Jul 24 12:03 memo
                 |       |
               owner   group
```

Student Notes

The UNIX system provides a three-tier access structure for a file:

user	Represents the owner of the file
group	Represents the group that may have access to the file
other	Represents all other users on the system

Every file will be owned by some user on the system. The owner has complete control over who has what kind of access to the file. The owner can allow or deny access to his or her files to other users on the system. The owner decides what group may have access to his or her files. The owner can also decide to give the file to some other user on the system. But once ownership is transferred, the original owner no longer has any control over the file.

Since files are owned by users and associated with groups, you can use the `id` command to display your identification status and determine what access you have to files that are stored on your system.

© 2003 Hewlett-Packard Development Company, L.P.

The files on the slide are owned by the user `user3`, and members of the group `class` may have access to these files. In addition, `user3` may allow all other users on the system access to these files.

© 2003 Hewlett-Packard Development Company, L.P.

6–3. SLIDE: Types of Access

Types of Access

There are three types of access for each file and directory:

Read

files:	Contents can be examined.
directories:	Contents can be examined

Write

files:	Contents can be changed.
directories:	Contents can be changed.

Execute

files:	File can be used as a command.
directories:	Can become current working directory

Student Notes

There are three types of access available for each file and directory: **read**, **write**, and **execute.**

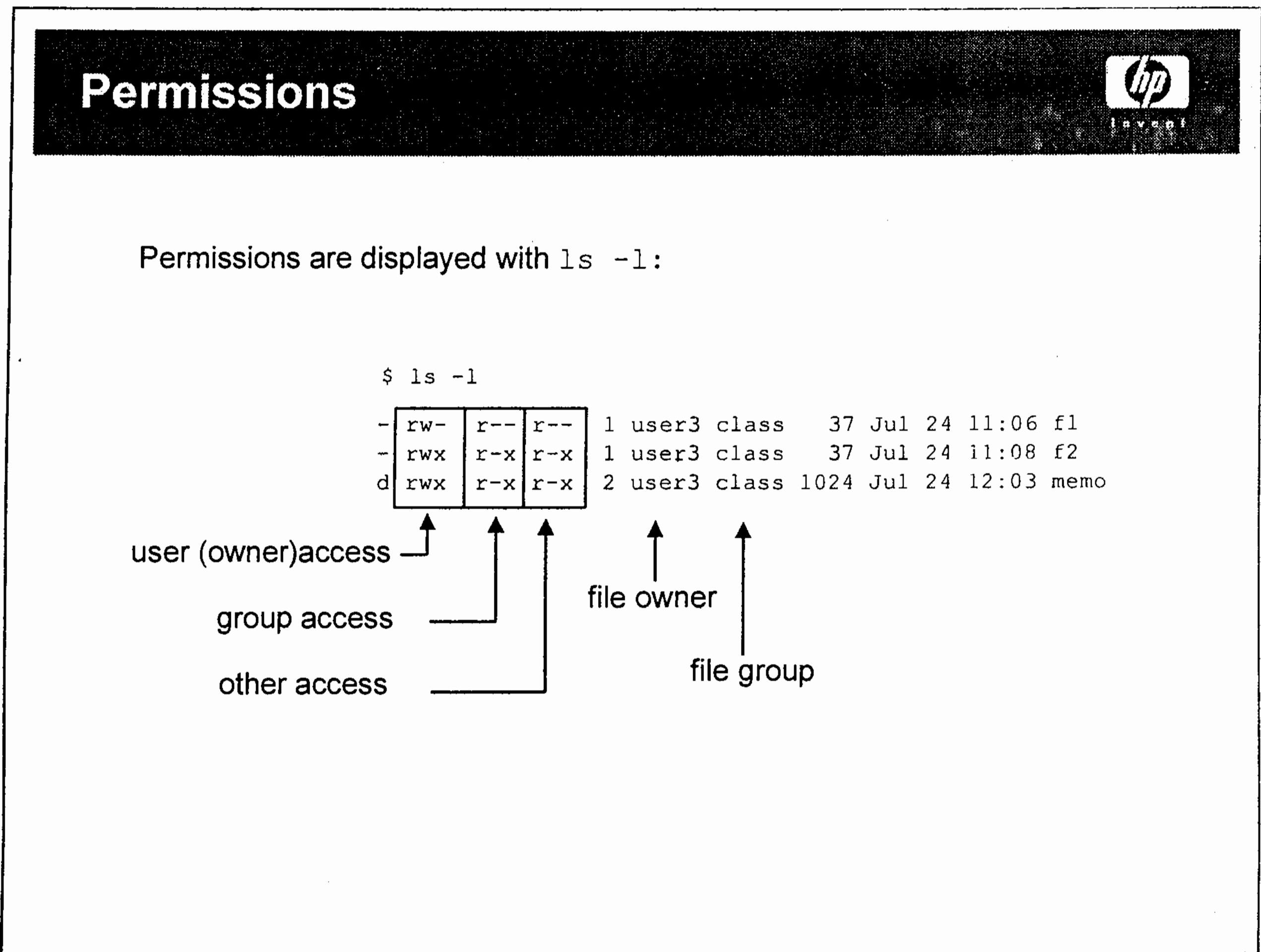
Different UNIX system commands require certain permissions in order to access a program or file. For example, to `cat` a file requires *read* permission, because the `cat` command must be able to read the contents of the file to display it to the screen. Likewise, a directory requires *read* permission to list out its contents with the `ls` command.

Notice that access is dependent on whether you are accessing a file or a directory. For example, *write* access on a file implies that the contents of the file can be changed. Denying *write* access prohibits users from changing the contents of the file. It does not protect the file from being deleted. *Write* access on a directory controls whether the contents of a directory can be modified. If a directory does not have *write* access, its contents cannot be changed, and therefore files cannot be deleted, added, or renamed.

NOTE:	In order to run a file as a program, both *read* and *execute* permissions are required.

© 2003 Hewlett-Packard Development Company, L.P.

6–4. SLIDE: Permissions



Student Notes

Your access to a file is defined by your *user* identification, your *group* identification, and the permissions associated with the file. The permissions to a file are designated in the **mode**. The mode of a file is a nine-character field that defines the permissions for the owner of the file, the group to which the file belongs, and all other users on the system.

© 2003 Hewlett-Packard Development Company, L.P.

Examples

Referring to the files listed on the slide, access is as follows:

File Name	Association	Access Attributes	Authorized Activities
`f1`	`user3` (owner)	read, write	Examine and modify the contents
	members of group `class`	read	Examine the contents
	all others	read	Examine the contents
`f2`	`user3` (owner)	read, write, execute	Examine and modify the contents, run as a program
	members of group `class`	read, execute	Examine the contents, run as a program
	all others	read, execute	Examine the contents, run as a program
`memo`	`user3` (owner)	read, write, execute	Examine and modify contents of directory memo, change to the directory memo
	members of group `class`	read, execute	Examine the contents of directory memo, change to the directory memo
	all others	read, execute	Examine the contents of directory memo, change to the directory memo

© 2003 Hewlett-Packard Development Company, L.P.

6–5. SLIDE: chmod — Change Permissions of a File

chmod — Change Permissions of a File

Syntax:

`chmod mode_list file...` Change permissions of file(s)

```
mode_list   [who[operator]permission] [ ,... ]
```

who	user, group, other or all
operator	+ (add), - (subtract), = (set equal to)
permission	read, write, execute

Example:

Original permissions:

```
mode        user   group  other
rw-r--r--   rw-    r--    r--
```

`$ chmod u+x,g+x,o+x file` or `$ chmod +x file`

Final permissions:

```
mode        user   group  other
rwxr-xr-x   rwx    r-x    r-x
```

Student Notes

The **chmod** command is used to change the permissions of a file or directory. Only the file's owner (or root — the system administrator) can change permissions. Therefore, in the UNIX system, access to a file is generally the responsibility of the owner of the file, as opposed to the system manager.

To protect a file from removal or corruption, the directory the file resides in *and* the file must *not* have write permission. Write permission to a file allows a user to change (or write over) the contents of the file, while write permission to a directory allows a user to remove the file. The **chmod** command supports an alpha method of defining the permissions for a file.

You can specify the permission that you wish to modify:

r	read permission
w	write permission
x	execute permission

© 2003 Hewlett-Packard Development Company, L.P.

And how you would like to modify that permission:

+	add permission
-	subtract permission
=	set permission equal

You can also specify which grouping of permissions you wish to modify:

u	user (owner of the file)
g	group (group the file is associated with)
o	other (all others on the system)
a	all (every user on the system)
none	assigns permission to all fields

NOTE: To disable all of the permissions on a file, issue the following command:

chmod = filename

Examples

```
$ ls -l f1
-rw-r--r-- 1 user3 class 37     Jul 24 11:06 f1
$ chmod g=rw,o= f1
$ ls -l f1
-rw-rw---- 1 user3 class 37     Jul 24 11:06 f1
$ ls -l f2
-rw-rw-rw- 1 user3 class 37     Jul 24 11:08 f2
$ chmod u+x,g=rx,o-rw f2
$ ls -l f2
-rwxr-x--- 1 user3 class 37     Jul 24 11:08 f2
```

You can use the **mesg n** command to disable other users from sending messages to your terminal. Every terminal has a device file, which is responsible for the communication between user and computer. In the example **/dev/tty0p1** should be that device file.

```
$ ls -l /dev/tty0p1
crw--w--w-   1 bin    bin   58 0x000003 Feb 15 11:34 /dev/tty0p3
$mesg n
$ ls -l /dev/tty0p1
crw-------   1 bin    bin   58 0x000003 Feb 15 11:34 /dev/tty0p3
```

Even when you disable messaging, the system administrator can still send messages to your terminal.

The **chmod** command also supports a numeric (octal) representation for assigning file permissions. This representation is obsolete, but it is a commonly used form.

© 2003 Hewlett-Packard Development Company, L.P.

1. To change file permissions, you must convert each group of permissions into the appropriate numeric representation. Access will be defined for the *owner*, the *group*, and *all others*. Remember that each type of access granted carries the following values:
   - read = 4
   - write = 2
   - execute = 1
2. Add together the numerical values associated with the access to be allowed.
3. Gather the three values together. This number will be your argument for the **chmod** command.

For example, if the desired permissions are **rw-** for owner, **r--** for group, and --- for other:

```
user    group   others
rw-     r--     ---
4+2+0   4+0+0   0+0+0
6       4       0
```

convert to numeric values

Thus the **chmod** command is:

chmod 640 *filename*

NOTE: To disable all permissions on a file, issue the following command:
chmod 000 file

© 2003 Hewlett-Packard Development Company, L.P.

6–6. SLIDE: umask — Permission Mask

umask — Permission Mask

Syntax:

`umask [-S] [mode]` User file-creation mode mask

Example:

	user	group	other
default permissions:	rw-	rw-	rw-
set default permissions:	rw-	r--	---

```
$ umask g=r,o=
```

Student Notes

The option **[-S]** prints the current file mode creation mask value using a symbolic format. The **[-S]** option and the symbolic format are not available in the Bourne and C shells.

The option **a-rwx** is the short form of **u-rwx,g-rwx,o-rwx**. The usual default permissions on a newly created file are **rw-rw-rw-**, which means that any user on the system can modify the contents of the file. The default permissions on a newly created directory are **rwxrwxrwx**, which means that any user can change to this directory *and* delete anything from this directory.

To protect the files that you will create during your session, use the **umask** command. This will disable designated default permissions on any *new* file or directory that you create. Write access to *group* and *others* are probably the most important permissions to disable. The mask that you designate is active until you log out. **umask** will have no effect on existing files.

© 2003 Hewlett-Packard Development Company, L.P.

6–7. SLIDE: touch — Update Timestamp on File

touch — Update Timestamp on File

Syntax:

```
touch [-amc] file...    update access and/or modification times of file
```

Examples:

```
$ ll
-rw-r--r--  1  karenk  users  25936 Aug 24 09:53 firstfile
-rw-r--r--  1  karenk  users  10245 Aug 24 09:53 secondfile
$ touch newfile
$ ll
-rw-r--r--  1  karenk  users  25936 Aug 24 09:53 firstfile
-rw-r--r--  1  karenk  users      0 Aug 25 10:02 newfile
-rw-r--r--  1  karenk  users  10245 Aug 24 09:53 secondfile
$ touch secondfile
$ ll
-rw-r--r--  1  karenk  users  25936 Aug 24 09:53 firstfile
-rw-r--r--  1  karenk  users      0 Aug 25 10:02 newfile
-rw-r--r--  1  karenk  users  10245 Aug 25 10:05 secondfile
$
```

Student Notes

The **touch** command allows you to create a new, empty file. If the designated file already exists, **touch** will update the time stamp on the file. It has no effect on the contents of the file.

The **touch** command has the following options:

-a *time*	Change the access time to ***time***.
-m *time*	Change the modify time to ***time***.
-t *time*	Use ***time*** instead of the current time.
-c	If the file does not already exist, do not create it.

© 2003 Hewlett-Packard Development Company, L.P.

Examples

```
$ touch test_file1
$ ls -l test_file1
-rw-rw-rw- 1 user3 class 0 Jul 24 11:08 test_file1
$ umask a-rwx,u=rw,g=r (or umask 137)
$ umask -S (or umask)
u=rw,g=r,o= (or 137)
$ touch test_file2
$ ls -l test_file2
-rw-r----- 1 user3 class 0 Jul 24 11:10 test_file1
```


© 2003 Hewlett-Packard Development Company, L.P.

6–8. SLIDE: chown — Change File Ownership

chown — Change File Ownership

Syntax:

```
chown owner [:group] filename ...
```

Changes owner of a file(s) and, optionally, the group ID

Example:

```
$ id
uid=303 (user3),  gid=300 (class)
$ cp f1 /tmp/user2/f1
$ ls -l /tmp/user2/f1
-rw-r----- 1 user3 class 3967 Jan 24 13:13 f1
$ chown user2 /tmp/user2/f1
$ ls -l /tmp/user2/f1
-rw-r----- 1 user2 class 3967 Jan 24 13:13 f1
```

Only the owner of a file (or root) can change the ownership of the file.

Student Notes

Only the owner of a file has control over the attributes and access to a file. If you would like to give ownership of a file to some other user on the system, use the **chown** command. For example, **user3** might make a copy of his file **f1** for **user2**. **user2** should have complete control of his personal copy, so **user3** transfers ownership of **/tmp/user2/f1** to **user2**. Optionally, **chown** changes the group ID of one or more files to *group*. The owner (group) can be either a decimal user ID (group ID) or a login name found in the **passwd** (group) file.

NOTE: Once the ownership of a file has been changed, only the *new owner* or *root* can modify the ownership and mode.

The *owner* is a user identifier recognized by your system. The file /etc/passwd contains the user IDs for all of your system's users.

© 2003 Hewlett-Packard Development Company, L.P.

Example

Looking at the example on the slide, after **user3** has transferred ownership of **/tmp/user2/f1** to **user2**, **user3** still has read access, as the file allows read access to all users who are a member of **class**.

In the Linux environment, the security policies are applied in a slightly different manner than those in HP-UX.

By default, in the Linux environment, only the administrative user called **root** can execute the **chown** command.

© 2003 Hewlett-Packard Development Company, L.P.

6–9. SLIDE: The chgrp Command

The chgrp Command

Syntax:

```
chgrp newgroup filename ...
```

Changes group access to a file
Only the owner of a file (or root) can change the group of the file.

Example:

```
$ id
uid=303 (user3),  gid=300 (class)
$ ls -l f3
-rw-r----- 1 user3 class 3967 Jan 24 13:13 f3
$ chgrp class2 f3
$ ls -l f3
-rw-r----- 1 user3 class2 3967 Jan 24 13:13 f3
$ chown user2 f3
$ ls -l f3
-rw-r----- 1 user2 class2 3967 Jan 24 13:13 f3
$
```

Student Notes

The *group* field in the long listing identifies what user group has access to this file. This can be modified with the **chgrp** command.

The **new_group** is a group identifier recognized by your system. The file **/etc/group** contains the group IDs for all of your system's users.

The **chgrp** command will not work if the new group specified does not exist. The system administrator controls group existence and membership.

NOTE:	Only the *owner* of a file (or **root**) can change the group identifier associated with a file.

© 2003 Hewlett-Packard Development Company, L.P.

Example

Looking at the example on the slide, after **user3** has transferred group access of the file **f1** to the group **class2**, access has not been affected as user3 still owns the file. After **user3** gives ownership of the file to **user2**, **user3** will not be able to access it at all, because **user3** is currently associated with the group **class**.

In the Linux environment, the **chgrp** command can only be executed by a user who currently belongs to the group that is to assume ownership of the file. The execution of this command may, therefore, differ from the example as shown for HP-UX.

In Linux, one of the security policies is that all users belong to a user-specific workgroup. For example, if a user called **alun** is created, that user will automatically belong to a user-specific workgroup called **alun**. The administrator must add that user to other work groups on the system in order to allow that user access to the appropriate files and directories.

© 2003 Hewlett-Packard Development Company, L.P.

6–10. SLIDE: su — Switch User ID

su — Switch User ID

Syntax:

```
su [user_name]        Change your user ID and group ID designation
```

Example:

```
  $ ls -l /usr/local/bin/class_setup
  -rwxr-x--- 1 class_admin teacher 3967 Jan 24 13:13
class_setup
  $ id
  uid=303 (user3), gid=300 (class)
  $ su class_admin
  Password:
  $ id
  uid=400 (class_admin), gid=300 (class)
  $ /usr/local/bin/class_setup
  $
  log out of su session
  $ Ctrl + D
```

Student Notes

The **su** command allows you to interactively change your user ID and group ID. **su** is an abbreviation for *switch user* or *set user ID*. This allows you to start a subsession as the new user ID, and grants you access to all of the files that the designated user ID owns. Therefore, for security purposes, you are required to enter the account's password to actually switch your user status.

With no arguments, **su** switches you to the user **root** (the system administrator). The **root** account is sometimes known as the *superuser*, since this login has access to anything and everything on the system. For this reason, many people think that the command **su** is an abbreviation for *superuser*. Of course, you must supply the **root** password.

NOTE: To get back to the user you were, do *not use* the **su** command again. Instead, use the **exit** command to exit the new session started for you by the **su** command.

© 2003 Hewlett-Packard Development Company, L.P.

Example

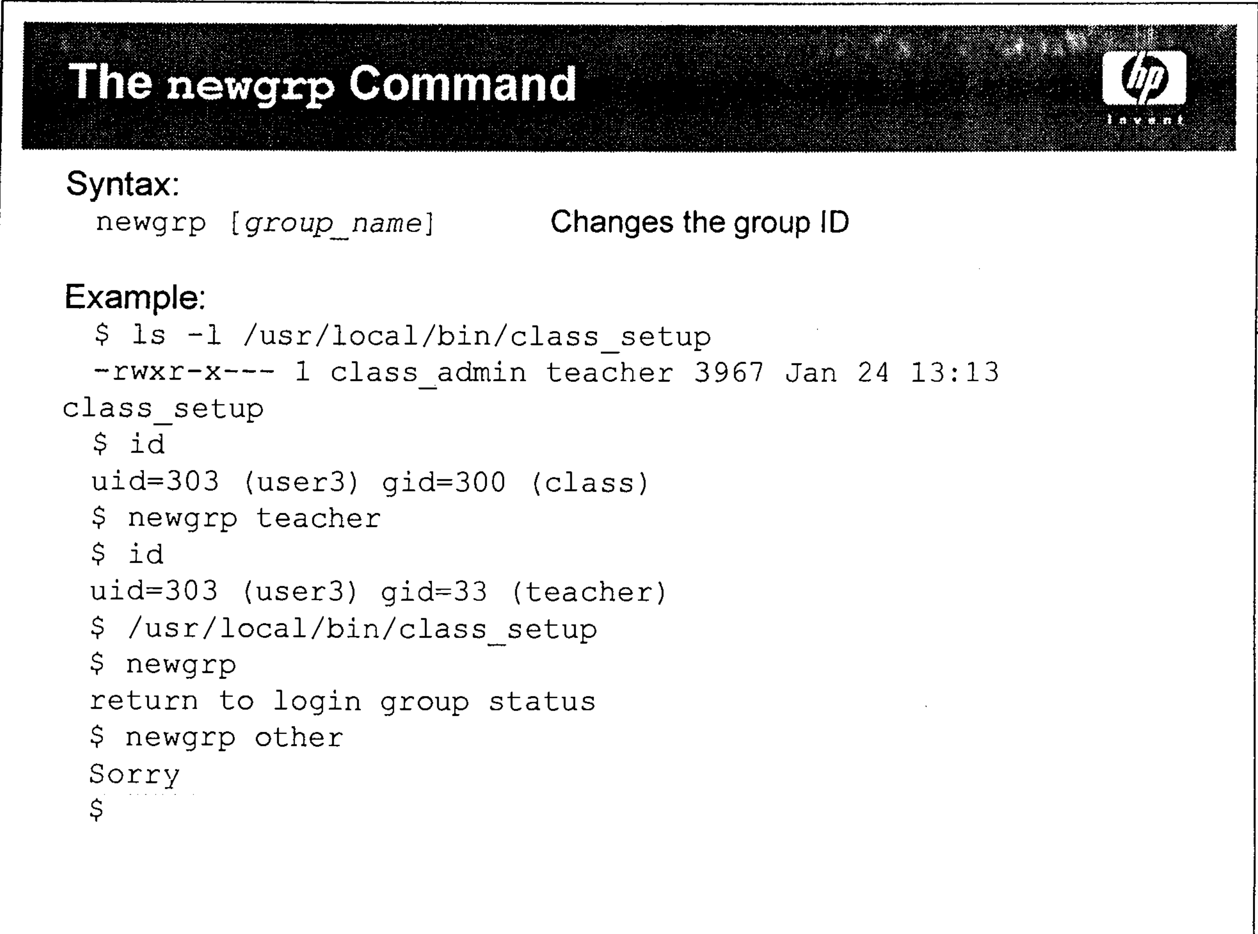
Look at the example on the slide. **user3** does not have access to the program **/usr/local/bin/class_setup**, because **user3** is not a member of the group **teacher**. **user3** can access this program, though, by entering the command **su class_admin**. As **class_admin**, **user3** can also modify the contents of the program **class_setup**. When finished running the program, **user3** resumes the original user status by logging out of the **su** session.

```
su - username
```

There are certain configuration files that set up your session for you. When you issue the command **su username**, your session characteristics remain the same as your original login identification. If you want your session to take on the characteristics associated with the switched user ID, use the dash (-) option with the **su** command: **su - username**.

© 2003 Hewlett-Packard Development Company, L.P.

6–11. SLIDE: The newgrp Command



Student Notes

The **newgrp** command is similar to the **su** command. The **newgrp** command allows you to change your group ID number.

The system administrator defines which groups you have access to change to. By looking at the file **/etc/group**, you can determine which groups you have access to change to. If you are not allowed to become a member of the specified group, you get the message: **Sorry**.

Since the **newgrp** command does not start a new subsession, use the **newgrp** command to return to your original group status.

Example

Look at the example on the slide. **user3** still does not have access to the program **/usr/local/bin/class_setup,** because **user3** is initially a member of the group **class**. **user3** can use the command **newgrp teacher** to the teacher group, because the system administrator has granted **user3** access to this group. Now **user3** can run the program, since the program has execute access allowed for anyone in the **teacher** group, but **user3** cannot modify the contents of the program. Only the user **class_admin** can

© 2003 Hewlett-Packard Development Company, L.P.

write to the **class_setup** program. When he has finished, **user3** will **newgrp** to resume original group status.

Sample /etc/group File

```
teacher::33:class_admin,user3

class::300:user1,user2,user3,user4,user5,user6,class_admin
```


© 2003 Hewlett-Packard Development Company, L.P.

6–12. SLIDE: Access Control Lists

Access Control Lists

Syntax:

`lsacl` *filename*	list the ACL for a file
`chacl ACL` *filename*	change the ACL for a file

Examples:

```
$ lsacl funfile
(user3.%,rw-)(%.class,r--)(%.%,r--) funfile
$ chacl "user2.class=rw,%.%-r" funfile
$ lsacl funfile
(user2.class,rw-)(user3.%,rw-)(%.class,r--)(%.%,---) funfile
$ chacl -d "user2.class" funfile
$ ll funfile
rw-r-----   1  user3  class    3081  May 28  16:12 funfile
$ lsacl funfile
(user3.%,rw-)(%.class,r--)(%.%,---) funfile
```

Note: ACLs are *not* supported with certain versions of JFS filesystems.

setacl
getacl

Student Notes

Access control lists (ACLs) are an enforcement mechanism of discretionary access control (DAC), which allows more selective access specification to files than the traditional UNIX system mechanisms provide. An ACL consists of a user ID and group ID combination, with the associated access permissions allowed for this user/group combination **`(user.group,mode)`**.

Levels of Access Control

There are basically four levels of specificity that can define access to a file:

`(u.g, rwx)`	Specific user, specific group
`(u.%, rwx)`	Specific user, any group
`(%.g, rwx)`	Any user, specific group
`(%.%, rwx)`	Any user, any group

Each file can have up to 13 different sets of permissions provided. If there are multiple, similar entries specifying file access, the more specific access takes precedence over less specific designations.

© 2003 Hewlett-Packard Development Company, L.P.

Example

```
$ lsacl funfile
(user3.%,rw-)(user2.class,rw-)(user2.%,r--)(%.class,r--)(%.%,---) funfile
```

Users will have the following access to the file **funfile**:

User ID	Group ID	Access to File
user3	any	read, write
user2	class	read, write
user2	any group other than class	read
any user other than **user2** and **user3**	class	read
any user other than **user2** and **user3**	any group other than class	none

Changing the Access Control List

The **chacl** command can be used to modify or delete an existing access control list. The **-d** option allows you to delete an existing ACL designation.

Examples

```
$ lsacl funfile
(user3.%,rw-)(%.class,r--)(%.%,r--) funfile
```

The following will add an ACL for **user2** of group **class**, providing read and write access, plus deleting read permission from the open field for all users, all groups (%.%):

```
$ chacl "user2.class=rw,%.% -r" funfile
$ lsacl funfile
(user2.class,rw-)(user3.%,rw-)(%.class,r--)(%.%,---) funfile
```

The above could have been implemented also with:

```
$ chacl "(user2.class,rw)(%.%,-r)" funfile
```

The following will delete the ACL for **user2** of group **class**:

```
$ chacl -d "user2.class" funfile
$ lsacl funfile
(user3.%,rw-)(%.class,r--)(%.%,---) funfile
```

NOTE: Changing permissions with **chmod** removes all ACLs for that file.

NOTE: ACLs are supported only on HFS file systems, which are not the default in HP-UX 11.00.

© 2003 Hewlett-Packard Development Company, L.P.

NOTE: ACLs are not supported by any of the Linux file systems.

© 2003 Hewlett-Packard Development Company, L.P.

6–13. SLIDE: File Permissions and Access — Summary

File Permissions and Access — Summary

Permissions	Define who has what access to a file user, group, others read, write, execute
`chmod`	Change the permissions of a file
`umask`	Define the default permissions for new files
`chown`	Change the owner of a file
`chgrp`	Change the group of a file
`su`	Switch user ID
`newgrp`	Switch group ID

Student Notes

Things to remember about file permissions:

- All directories in the full pathname of a file must have execute permission in order for the file to be accessible.
- To protect a file, take away write permission on that file and on the directory in which the file resides.
- Only the owner of a file (or **`root`**) can change the mode (**`chmod`**), the ownership (**`chown`**), or the group (**`chgrp`**) of a file.

© 2003 Hewlett-Packard Development Company, L.P.

6–14. LAB: File Permissions and Access

Directions

There are four sections of exercises to complete. Run the commands necessary to solve the exercises and answer the associated questions. Time may not allow you to complete *all* of the exercises.

File Permissions

1. Look under your **HOME** directory for a file called `mod5.1`. Who has what kind of access to this file? Can you display the contents of `mod5.1`?

2. Modify the permissions on `mod5.1` so that they are `-w-------`. Can you display the contents of `mod5.1`?

3. Modify the permissions on `mod5.1` so that they are `rw-------`. Can you display the contents of `mod5.1`? Can another user in your group display the contents of your `mod5.1`?

4. How can you modify the permissions on `mod5.1` so that another user in your group *can* read the file?

© 2003 Hewlett-Packard Development Company, L.P.

5. Make a copy of `mod5.1` and call it `mod5.2`. Remove the write permissions from `mod5.2`. Can you delete this file? How do you protect this file from being deleted?

6. Who is the owner of the file `root_file` in your HOME directory? To what group does it belong? Who is allowed to change the ownership or group? What access do you have to this file?

7. If you want to make changes to the file `root_file`, how do you go about it?

8. Run the command **`mesg y`**. Now, type **`tty`** and note the device file associated with your terminal. What are the permissions on this file? Who owns this file? Run the command **`mesg n`**. What are the permissions now? What is the **`mesg`** command effectively doing?

Directory Permissions

1. Under your HOME directory, create a directory called `mod5.dir`. Copy the file `mod5.1` to `mod5.dir`. List the contents of the new directory. What are the permissions on the `mod5.dir`? (Hint: `ls -ld mod5.dir`)

© 2003 Hewlett-Packard Development Company, L.P.

2. Modify the permissions on `mod5.dir` to be `rw-------`. Can you change directory to `mod5.dir`? Can you display the contents of `mod5.dir`? Can you access the contents of the file `mod5.1` under the `mod5.dir`?

3. Modify the permissions on `mod5.dir` to be `-wx------`. Can you display the contents of `mod5.dir`? Can you display the contents of the file `mod5.1` under the `mod5.dir`? Can you change directory to `mod5.dir`?

4. Can other users copy files into your HOME directory? How do you display the permissions for your HOME directory?

5. From your HOME directory, copy the file `mod5.1` to the directory `/usr/bin`. Did you have any problems? What are the permissions of `/usr/bin`?

6. Can you copy the file `/usr/bin/date` to your HOME directory?

© 2003 Hewlett-Packard Development Company, L.P.

Changing Ownership and Group

Linux system users are not permitted to use the `chown` command. Linux system users should not attempt questions 2 through 4.

1. Look in your **HOME** directory for a file that has the same name as your login name. What access do you have to this file? What group does your partner belong to? What access does your partner have to this file?

2. Still working with the file **YOUR_LOGNAME**, change the ownership of this file to your partner. Can you access the file now? Try to make a copy of the file. Can you get ownership back?

3. Make a copy of **mod5.1** and call it **mod5.3**. Remove *all* of the permissions from the file **mod5.3**. Can you change the ownership of this file to your partner?

4. Make a copy of **mod5.1** and call it **mod5.4**. Modify the permissions so that they are **rw-r-----**. Change the group of the file to **class2**. Change the owner of the file to **root**. Can you display the contents of **mod5.4**?

© 2003 Hewlett-Packard Development Company, L.P.

5. Change your group status to `class2`. Can you display the contents of `mod5.4`? Return your group status to your original group ID. (Hint: use the `id` command to see your user and group identifications.)

Permissions for New Files

1. What are the default permissions when you create a new file? Hint: Create a new file by using the editor, copying an existing file, or using the `touch` command. Examine the permissions on the new files. How about a new directory? What is your current file creation mask?

2. How would you modify the default creation permissions to deny write access to others in your group, and others on the system? Test this by creating another new file and another new directory.

© 2003 Hewlett-Packard Development Company, L.P.

Module 7 — Shell Basics

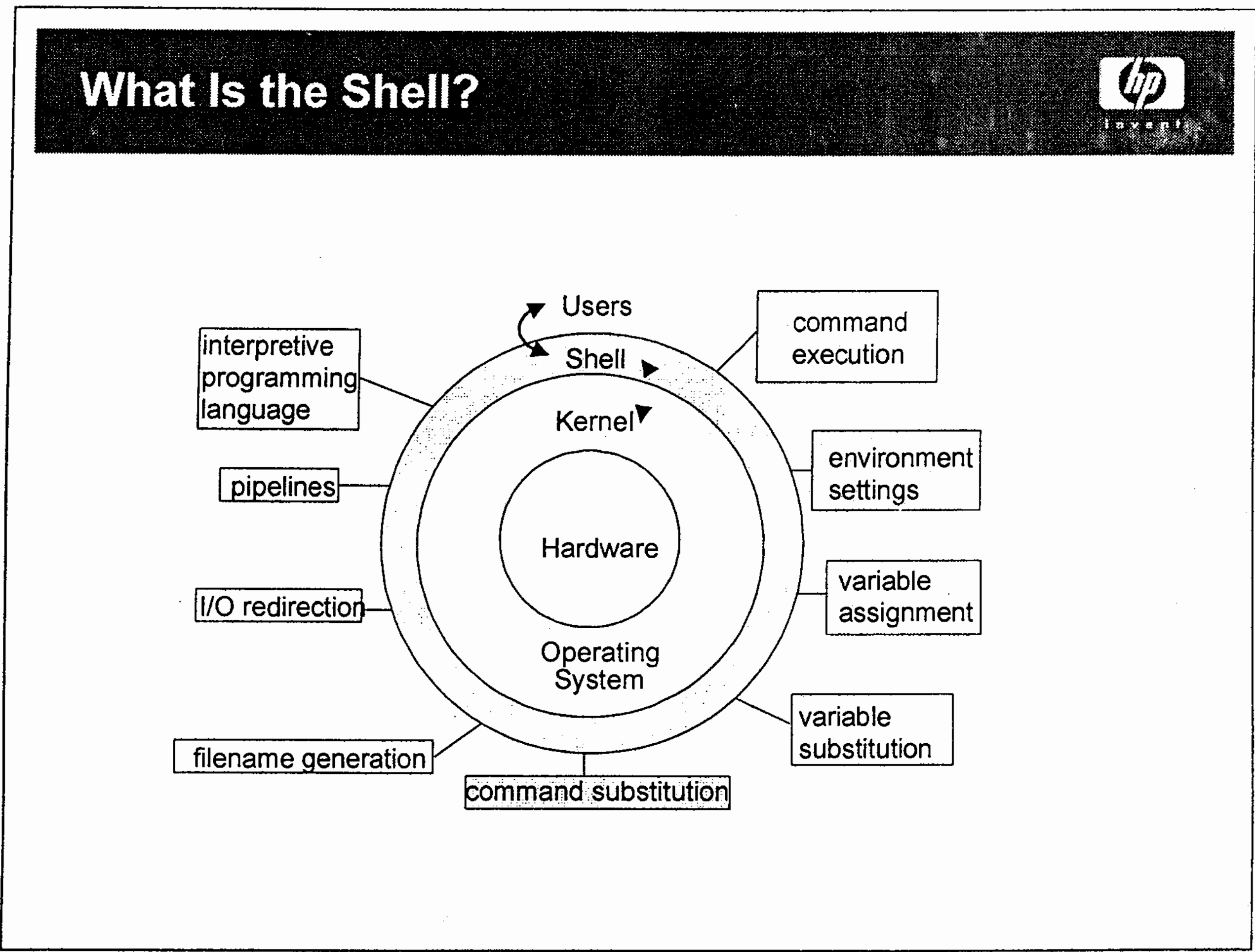
Objectives

Upon completion of this module, you will be able to do the following:

- Describe the job of the shell.
- Describe what happens when someone logs in.
- Describe user environment variables and their functions.
- Set and modify shell variables.
- Understand and change specific environment variables such as *PATH* and *TERM*.
- Customize the user environment to fit a particular application.

© 2003 Hewlett-Packard Development Company, L.P.

7–1. SLIDE: What Is the Shell?



Student Notes

A **shell** is an interactive program that serves as a command line interpreter. It is separate from the operating system. This design provides users with the flexibility of selecting the interface that is most appropriate for their needs. A shell's job is to allow you to type in your command, perform several functions, and pass the interpreted command to the operating system (kernel) for execution.

This module presents interactive features that are provided by the POSIX shell. Interactively, the POSIX shell completes other functions in addition to executing your command. Many of these functions are completed *before* the command is executed.

The following summarizes the shell functionality:

- It searches for a command and executes the associated program.
- It substitutes shell variable values for dereferenced variables. It performs command substitution.
- It completes file names from file name generation characters.

© 2003 Hewlett-Packard Development Company, L.P.

- It handles I/O redirection and pipelines.
- It provides an interpreted programming interface, including tests, branches, and loops.

As you log in to a UNIX system, the shell defines certain characteristics for your terminal session, then issues your prompt. This prompt defaults to a `$` symbol in the case of the POSIX, Bourne, and K shells. The default prompt for the C shell is the percent sign (`%`).

© 2003 Hewlett-Packard Development Company, L.P.

7–2. SLIDE: Commonly Used Shells

Commonly Used Shells

`/usr/bin/sh`	POSIX shell
`/usr/bin/ksh`	Korn shell
`/usr/old/bin/sh`	Bourne shell
`/usr/bin/csh`	C Shell
`/usr/bin/keysh`	A context-sensitive softkey shell
`/usr/bin/rksh`	Restricted Korn shell
`/usr/bin/rsh`	Restricted Bourne shell

Student Notes

The POSIX shell is a POSIX-compliant command programming language and commands interpreter residing in **`/usr/bin/sh`**. It can execute commands read from a terminal or a file. This shell conforms to the current POSIX standards in effect at the time the HP-UX system release was introduced, and is similar to the Korn shell in many ways. It contains a history mechanism, supports job control, and provides various other useful features.

The Korn shell is a command programming language and commands interpreter residing in **`/usr/bin/ksh`**. It can execute commands read from a terminal or a file. Like the POSIX shell, it contains a history mechanism, supports job control, and provides various other useful features. David Korn of AT&T Bell Labs developed the Korn shell.

The Bourne shell is a command programming language and command interpreter, residing in **`/usr/old/bin/sh`**. It can execute commands read from a terminal or a file. This shell lacks many features contained in the POSIX and Korn shells. Stephen R. Bourne developed the Bourne shell. It was the original shell available on the AT&T releases of UNIX.

The C shell is a command language interpreter that incorporates a command history buffer, C-language-like syntax, and job control facilities. William Joy of the University of California at Berkeley developed it.

© 2003 Hewlett-Packard Development Company, L.P.

The `rsh` and `rksh` are restricted versions of the Bourne shell and Korn shells, respectively. A restricted shell sets up a login name and execution environment whose capabilities are more controlled (restricted) than normal user shells. A restricted shell acts very much like a standard shell with several exceptions. A user using a restricted shell cannot:

- change directory
- reset value of *PATH* environment variable
- use the / character in a path name
- redirect output.

The keyshell is an extension of the standard Korn shell. It uses hierarchical softkey menus and context-sensitive help to aid users in building command lines. `keysh` was developed by HP and AT&T.

Table 1 Comparison of Shell Features

Features	Description	POSIX	Bourne	Korn	C
Command history	A feature allowing commands to be stored in a buffer, then modified and reused.	Yes	No	Yes	Yes
Line editing	The ability to modify the current or previous command lines with a text editor.	Yes	No	Yes	No
File name completion	The ability to automatically finish typing file names in command lines.	Yes	No	Yes	Yes
Alias command	A feature allowing users to rename commands, automatically include command options, or abbreviate long command lines	Yes	No	Yes	Yes
Restricted shells	A security feature providing a controlled environment with limited capabilities.	Yes	Yes	Yes	No
Job control	Tools for tracking and accessing processes that run in the background.	Yes	No	Yes	Yes

© 2003 Hewlett-Packard Development Company, L.P.

In the Linux system, there is also a choice of shells:

`/bin/ash`	The **A** shell. Similar to the Bourne shell.
`/bin/bash`	The GNU Bourne-Again shell – used mostly by administrators
`/bin/ksh`	The public domain version of the Korn shell
`/bin/csh`	The **C** shell
`/bin/sh`	A link to the bash shell
`/bin/tcsh`	An amended version of the **C** shell
`/bin/zsh`	The **Z** shell

© 2003 Hewlett-Packard Development Company, L.P.

7–3. SLIDE: POSIX Shell Features

Student Notes

One of the shells provided with UNIX is the **POSIX** shell. This shell has many of the same features as the Korn shell, but that the Bourne shell does not. Even if you do not use all of the advanced features, you will probably find the POSIX shell a very convenient user interface. Here are just a few of the features of the POSIX shell:

- Command history mechanism
- Command line recall and editing
- Job control
- File name completion
- Command aliasing
- Enhanced `cd` capabilities
- Advanced programming capabilities

© 2003 Hewlett-Packard Development Company, L.P.

7–4. SLIDE: Aliasing

Aliasing

Syntax:

```
alias [name[=string]]
```

Examples:

```
$ alias dir=ls
$ alias mroe=more
$ alias mstat=/home/tricia/projects/micron/status
$ alias laser="lp -dlaser"
$ laser fileX
request id is laser-234 (1 file)
$ alias              Displays aliases currently defined
$ alias mroe         Displays value of alias mroe
mroe=more
```

Student Notes

An **alias** is a new name for a command. Aliasing is a method by which you abbreviate long command lines, create new commands, or cause standard commands to perform differently by replacing the original command with a new, shorter command, called an alias. The alias can be a letter or short word. For example, many people use the `ps -ef` command quite often. Wouldn't it be much easier to type `psf` instead? You create aliases with the `alias` command.

```
$ alias name=string
```

Where *name* is the name you are using for the alias, and *string* is the command or character string that the alias represents. If the string contains spaces, enclose the entire string in quotes. The alias is convenient to limit typing, interpret common typing errors, or generate new commands.

An alias looks just like any other command when it is entered. It is transparent to the user if he or she is executing a real UNIX system command or an alias that references a UNIX system command.

© 2003 Hewlett-Packard Development Company, L.P.

The shell will expand the alias *prior* to command execution, and then execute the resulting command line. When entered interactively, the alias is available until you log out.

Some users find this feature so flexible that they make their UNIX system interface recognize commands they usually enter through another operating environment (for example, **`alias dir=ls`** or **`alias copy='cp -i'`**).

Aliases are also often used as shorthand for full path names.

With no arguments, the **`alias`** command reports all aliases currently defined.

To list the value of a particular alias, use **`alias`** ***name***.

Aliases can be turned off with the **`unalias`** command. The syntax is

```
unalias name
```

Examples

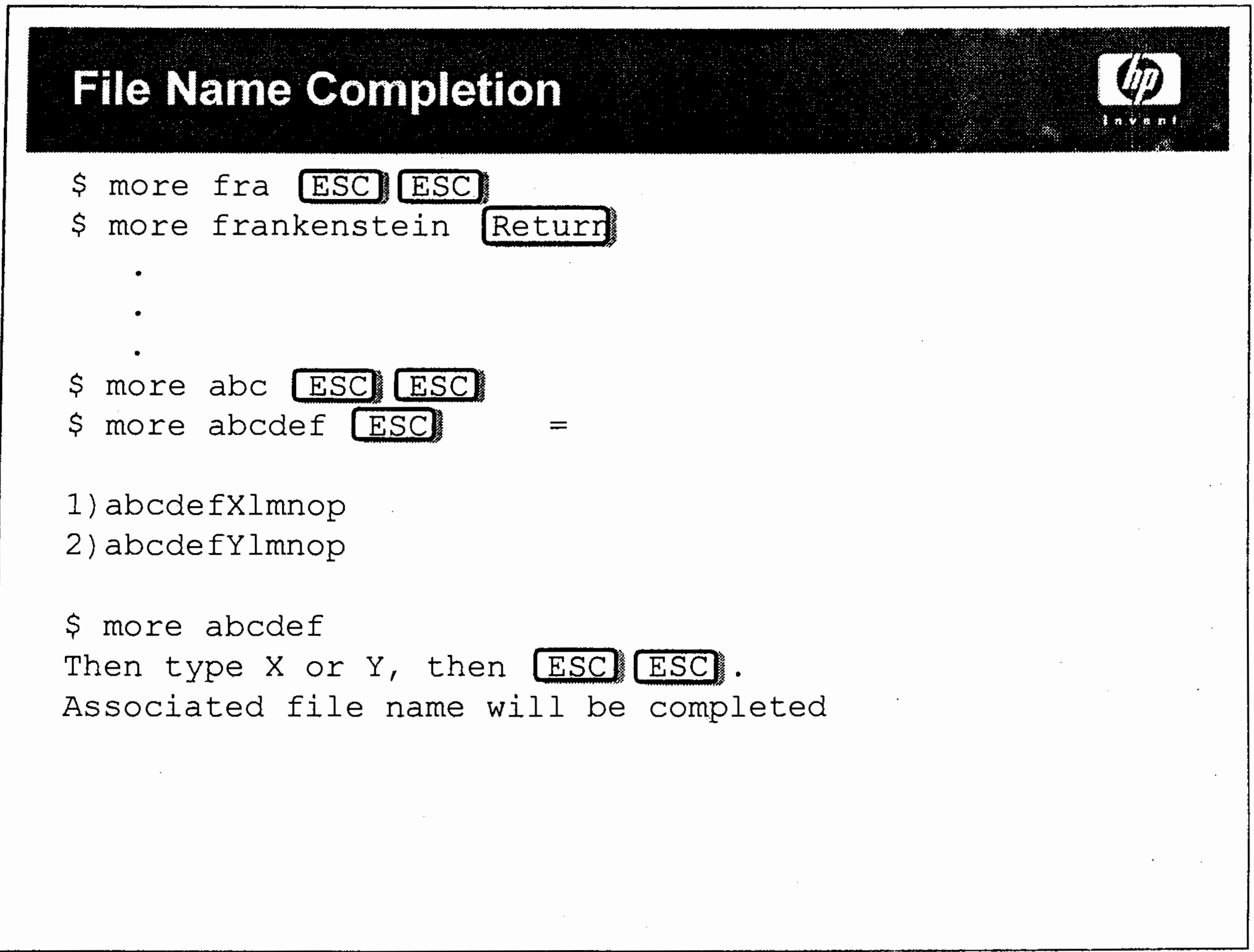
Several aliases can also be entered on a single command line as shown below:

```
$ alias go='cd '
$ alias there=/home/user3/tree/ford/sports
$ go there
$ pwd
/home/user3/tree/ford/sports
```

In order to reference more than one alias on a line, you must leave a space as the last character in the alias definition; otherwise, the shell will not recognize the next word as an alias.

© 2003 Hewlett-Packard Development Company, L.P.

7-5. SLIDE: File Name Completion



Student Notes

File name completion is convenient when you want to access a file that has a long file name. Provide enough characters that uniquely identify the file name, then press Esc Esc, and the POSIX shell will fill in the remainder of the file name. If the string is not unique, the POSIX shell cannot resolve the file name and you will have to provide some assistance. Your terminal will beep when it runs into a file name conflict.

The shell will complete the file name as far as it can without a conflict. You can then list the possible choices at this time by typing Esc =. After the POSIX shell has displayed the available options, you can use editor commands to add subsequent characters that will uniquely identify the desired file, and then enter Esc Esc to conclude the file name.

File name completion can be used anywhere in the path of a file name. For example,

```
$ cd tr Esc Esc do Esc Esc r Esc Esc
```

Causes the following command line to be displayed:

```
$ cd tree/dog.breeds/retriever
```


© 2003 Hewlett-Packard Development Company, L.P.

Filename Completion in Linux

Filename completion in the Bash shell is identical to that in the POSIX shell. However, the Linux Korn shell uses a slightly different set of filename completion controls. A comparative list is provided below:

Description	HP-UX POSIX	Linux bash	Linux Korn
Does the `vi` command line editing option have to be turned on?	No	No	Yes `$ set -o vi`
Complete the filename when only one match exists	Esc Esc	Esc Esc	Esc \
Display possible matching file names when more than one matching name exists	Esc =	Esc =	Esc =

© 2003 Hewlett-Packard Development Company, L.P.

7–6. SLIDE: Command History

Command History

- The shell keeps a history file of commands that you enter.
- The history command displays the last 16 commands.
- You can recall, edit, and re-enter commands previously entered.

Syntax:

`history [-n| a z]` Display the command history.

Example:

```
$ history -2           List the last two commands
15 cd
16 more .profile
$ history 3 5          List command numbers 3 through 5
3 date
4 pwd
5 ls
```

Student Notes

The POSIX shell keeps a history file that stores the commands you enter and allows you to re-enter them. The history file is maintained across login sessions.

The **history** command displays the last 16 commands you have entered. Each line is preceded with a command number. You can refer to that command number when re-entering the command.

You can display more or less than the last 16 commands you entered by typing

```
history -n
```

Where *n* is the number of commands to display.

You can display a range of command numbers by typing

```
history a z
```

Where *a z* is a command number or range of commands.

© 2003 Hewlett-Packard Development Company, L.P.

The *HISTSIZE* variable defines how many previous commands you are able to access (the default *HISTSIZE* is 128 lines). The *HISTFILE* variable specifies a text file that is created to store commands that you have entered (the default *HISTFILE* is `.sh_history`).

Once command history has been displayed, you can recall, edit, or re-enter any of the commands.

© 2003 Hewlett-Packard Development Company, L.P.

7–7. SLIDE: Re-entering Commands

Re-entering Commands

- You type `r c` to re-enter command number `c`.

Example:

```
 $ history 3 5          List command numbers 3 through 5
 3 date
 4 pwd
 5 ls
 $
 $ r 4                  Run command number 4
 pwd
 /home/kelley
```

Student Notes

You can run any command from the command history by simply typing:

r *c*

Where *c* is the command number. You can also enter the first letter of a command, and execute the most recent command that begins with that letter. For example,

```
$ history
1 date
2 cat file1
3 ls -l
$ r d
Mon Jul 4 10:03:13 1994
```


© 2003 Hewlett-Packard Development Company, L.P.

7–8. SLIDE: Recalling Commands

Recalling Commands

- Uses the history mechanism.
- Must have the *EDITOR* environment variable set.
  ```
  EDITOR=vi
  export  EDITOR
  ```
  - At `$`, press `Esc` and use normal `vi` commands to scroll through previous commands.
    - `k` scrolls backward through the command history.
    - `j` scrolls forward through the command history.
    - *n*`G` takes you to command number *n*.
  - Press `Return` to execute the command.

Student Notes

The most widely used editor embedded in the UNIX shell is **vi**. Basic functions of this editor will be used to illustrate command line editing. Detailed use of **vi** is covered elsewhere.

The shell history feature allows you to recall your previous commands so that you can re-execute them without retyping the line. This mechanism also allows you to edit previous command lines using **vi**. These features can save you a great deal of typing. If you are not a great typist, they will also save you a lot of time and aggravation.

In order to use **vi** commands to access the POSIX shell history mechanism, you need the EDITOR variable set in your environment. Execute the **env** command to see this listing:

```
$ env
.
.
EDITOR=/usr/bin/vi .
```


© 2003 Hewlett-Packard Development Company, L.P.

If this parameter is not set, execute these commands to set it:

```
$ EDITOR=/usr/bin/vi
$ export EDITOR
```

This tells the POSIX shell that you want to use the **vi** editor to recall and edit your previous commands. Put these commands in your **.profile** file if you want to make sure *EDITOR* is set every time you log in. If you do not set the *EDITOR* variable, it defaults to **/usr/bin/ed**.

To recall a previous command, simply press [Esc]. You will not see anything happen on your screen yet. Pressing [Esc] puts you in POSIX shell's **vi** mode. At this point, you have many of the normal **vi** commands available to you. For example, pressing [k] moves you back one command in your command stack. If you continue to press [k], you will see your previous commands appear on your command line one at a time. Similarly, if you press the [j] key, you will scroll through your commands in the opposite direction. When you see the command you want on the command line, press [Return].

You can also use the **history** command to see your last 16 commands. It will list the number of the command with the command line. To execute a particular command, type [Esc] ***n*** **G**, where *n* is the command line number from the **history** listing, and G is a command in **vi** that moves you to a specific line.

© 2003 Hewlett-Packard Development Company, L.P.

7–9. SLIDE: Command Line Editing

Command Line Editing

- Provides the ability to modify text entered on current or previous command lines.
- Press [Esc] to enter *command mode*.
- Recall desired command by either
  - Pressing [K] until it appears
  - Typing the *command number*, then G

Student Notes

There are times you would like to recall a command and reuse it, but it needs some minor changes first. By pressing [Esc] and then [k], you will recall the last command. If you know the command number, you can type **command number**, then **G**, to bring up the desired command. For example, assume the **history** command reported the following input:

```
120     env
121     ls
122     cd
123     cd /tmp
124     pwd
125     history
```

If you typed [Esc] [k] and then **122G**, the following line would be recalled:

```
cd
```

An alternate way to locate commands in the command stack is to press [Esc] [k], then type ***/ pattern***. For example, after entering the command stack with [Esc] [k], type **/cd** to locate the last **cd** command. If you type another **/** you will recall the next to last **cd**

© 2003 Hewlett-Packard Development Company, L.P.

command, and so on. Once you have searched for a pattern, typing **n** will also search for the next occurrence.

At this point, you can press Return to execute the command or use the editing commands discussed on the next slide. If you decided not to execute the command, typing Ctrl+c cancels the command.

© 2003 Hewlett-Packard Development Company, L.P.

7–10. SLIDE: Command Line Editing (Continued)

Command Line Editing (Continued)

- To position the cursor
  - Use `l`, or [space] key to move right
  - Use `h`, or [backspace] to move left
- *Do Not Use the Arrow Keys*!
- To modify text
  - Use `x` to delete a character
  - Use `i` or `a` to insert or append characters
  - Press [Esc] to stop adding character
- Press [Return] to execute the modified command

Student Notes

How many times have you been typing a long command line when you found out that you made a mistake at the very beginning of the line? It happens all the time, and all you can do is backspace and retype everything after the mistake.

The POSIX shell lets you correct your mistakes and change parts of a command line before you execute it. Again, this is done with **vi** editing commands.

To change a command line, press [Esc] to enter **vi** editing mode. This works on command lines that you are typing *and* on the lines that you recalled using [Esc] and [k].

Once you are in editing mode, use **vi** commands. For example, **x** deletes a character, **h** and **l** move you left and right across the line, **cw** changes a word, **dw** deletes a word, and so on.

The command stack and line edit features are accessed using **vi** commands. The advantage this design provides is that once you are familiar with the **vi** commands, you have the tools necessary to use the command stack. You *do not* have to learn *another* interface and set of commands! Use the following **vi** commands to edit the command line:

© 2003 Hewlett-Packard Development Company, L.P.

h, Backspace, **l**, Space, **w**, **b**, $	Move the cursor.
x, **dw**, **p**	Delete and paste text.
r, **R**, **cw**	Change text.
a, **I**	Enter *input mode* to add new text.

To have access to the command stack through **vi** commands, set the variable **EDITOR=/usr/bin/vi.** (Other editor options include **gmacs** and **emacs**.)

Consider each command line as a mini-**vi** session. You are in *input mode* at the beginning of each command line. To access previously entered commands, issue the **vi** command that scrolls the cursor up. Before you can issue a **vi** command though, you must toggle to the *command mode* by pressing the Esc key.

Now you can enter the **vi** command to scroll up, which is k. As you continue to enter k s, you step back through your previous commands. When the command is displayed that you want to run, press Return and your command will be executed. This command is then appended to your command stack.

A major benefit of the POSIX shell is that it allows you to enter the current command line, as well as previous commands. It is not necessary to backspace to the point where a change is needed or to start over.

This feature is especially useful when entering long command lines that contain simple typing mistakes, or modifying arguments. Before this feature, you would have had to re-enter the complete line, or Backspace and retype the line.

With the POSIX shell line-editing feature, you can display a previously entered line, and make changes to the line with **vi** commands before executing it. The changes can be as simple as a single character or as extensive as the entire argument list of the command line.

Example

```
$ cp /usr/lib/X11/app-defaults
Usage: cp f1 f2
       cp [-r] f1 ...  fn d1
```

The above was supposed to be **cd**, not **cp**. The POSIX shell lets you fix the line without retyping it. Just press Esc and then k, and the command line will come back. Type **l** to move to the **p** in **cp** and use the **r** command to replace the **p** with a **d**. Your command line will now look like this:

```
$ cd /usr/lib/X11/app-defaults
```

Now, press Return and the **cd** command will execute.

If you had problems editing the line and want to try again, press Break to cancel editing. You will get your regular shell prompt back so you can try again.

Do not use the arrow keys when you are editing command lines in the POSIX shell. In addition to the h and l keys, you can use Backspace and the Space bar.

© 2003 Hewlett-Packard Development Company, L.P.

Transposing characters is another common typing error. Suppose you entered the following line, with the **r** and **o** transposed in **ford**:

```
$ cd $HOME/tree/car.models/frod/sports
cd: directory not found
```

Use the following steps to make the repair, and re-execute the line:

[Esc]

[k] Re-enter as many times as necessary to display the line.

[w] Re-enter until the cursor is under the **f** in **frod**.

[l] Cursor should be under the **r** in **frod**.

[x] [p] Delete the **r** and paste after the **o**.

[Return] Execute the line.

© 2003 Hewlett-Packard Development Company, L.P.

7–11. SLIDE: The User Environment

The User Environment

Your environment describes your session to the programs you run.

Syntax:

```
env
```

Example:

```
$ env
 HOME=/home/gerry
 PWD=/home/gerry/develop/basics
 EDITOR=vi
 TERM=70092
  ...
PATH=/usr/bin:/usr/contrib/bin:/usr/local/bin:\
 /home/gerry/bin
```

Student Notes

Your environment describes many things about your session to the programs that you run. It describes your session to the system. Your environment contains information concerning the following:

- The path name to your home directory
- Where to send your electronic mail
- The time zone in which you are working
- Whom you logged in as
- Where your shell will search for commands
- Your terminal type and size
- Other things your applications may need

For example, the commands **`vi`** and **`more`** need to know what kind of terminal you are using so they can format the output correctly.

© 2003 Hewlett-Packard Development Company, L.P.

An analogy to your user environment is your office environment. In the office, characteristics such as lighting, noise, and temperature are the same for all workers. The factors in your office that are unique to you make up your specific environment. These factors include what tasks you are performing, the physical layout of your desk, and how you relate to other people in the office. Your work environment is unique to you just as your user environment is unique.

Many applications require you to customize your environment in some way. You do this by modifying your `.profile` file.

When you log in, you can check your environment by running the `env` command. It will display every characteristic that is set in your environment.

In the `env` listing, the words to the left of the = are the names of the different environment variables that you have set. Everything to the right of the = is the value associated with each variable. See `env(1)` for more details.

Each one of these environment variables is set for a reason. Here are a few common environment variables and their meanings:

TERM, *COLUMNS*, and *LINES*	Describe the terminal you are using
HOME	Path name to your home directory
PATH	List of places to find commands
LOGNAME	User name you used to log in
ENV and *HISTFILE*	Special POSIX shell variables
DISPLAY	Special X Window variable

Some of these variables are set for you by the system, while others are set in `/etc/profile` or `.profile`.

© 2003 Hewlett-Packard Development Company, L.P.

7–12. SLIDE: Setting Shell Variables

Setting Shell Variables

- A shell variable is a name that represents a value.
- The value associated with the name can be modified.
- Some shell variables are defined during the login process.
- A user can define new shell variables.

Syntax:

```
name=value
```

Example:

```
$ PATH=/usr/bin/X11:/usr/bin
```

Student Notes

A **shell variable** is similar to a variable in algebra. It is a name that represents a value. Variable assignment allows a value to be associated with a variable name. The value can then be accessed through the variable name. If the value is modified, the new value can still be accessed through the same variable name. The syntax for assigning a value to a shell variable is ***name=value***.

This can be typed in at the terminal after a shell prompt or as a line in a shell program. *Notice that there is no white space either before or after the equal sign (=).* This ensures that the shell will not try to interpret the assignment as a command invocation.

It is important to distinguish between the **name** of a shell variable and the **value** of a shell variable. The variable value is set by performing an assignment statement, such as

```
TERM=70092
```

This tells the shell to remember the name *TERM*, and when the value of the variable *TERM* is requested, respond with 70092.

© 2003 Hewlett-Packard Development Company, L.P.

Variable Naming Restrictions

Variable names must start with an alpha character (a–z and A–Z) and can contain alpha, numeric, or underscore characters. There is no restriction on the number of characters that a variable name can contain.

© 2003 Hewlett-Packard Development Company, L.P.

7–13. SLIDE: Two Important Variables

Two Important Variables

- The *PATH* variable
  - A list of directories where the shell will search for the commands you type
- The *TERM* variable
  - Describes your terminal type and screen size to the programs you run

```
$ env
  ...
PATH=/usr/bin:/usr/contrib/bin:/usr/local/bin
$ TERM=70092
$
$ tset
Erase is Backspace
Kill is Ctrl-U
$
```

Student Notes

PATH

The *PATH* variable is a list of directories that the shell will search through to find commands. It gives us the ability to type just a command name instead of the full path name to that command (for example, **`vi`** instead of **`/usr/bin/vi`**). This is an example of the default

```
PATH variable: PATH=/usr/bin:/usr/contrib/bin:/usr/local/bin
```

This means that when you type a command, the shell will search for that command in **`/usr/bin`**, then **`/usr/contrib/bin`**, and so on until it either finds the command or it runs out of directories to look in. If the command you are trying to run cannot be found in any of the *PATH* directories, you will get the "**`command: not found`**" error message on your screen.

TERM

TERM is the environment variable that describes the type of terminal you have. For many commands to run correctly, they need to know what kind of terminal you are using. For example, the **`ls`** command needs to know how many columns there are on the screen, **`more`**

© 2003 Hewlett-Packard Development Company, L.P.

needs to know how many lines there are, and `vi` needs to know both how many columns and how many lines there are, plus much more information about your terminal type, in order to work properly. The terminal type is set using the terminal's model number (such as 2392, 70092, and so on).

The default method for setting up the terminal variable is by prompting the user for the proper terminal type in the following fashion:

```
TERM=(hp)
```

At this prompt, you can either press [Return] to set the terminal type to **hp**, or you can type the name of the terminal you are using. Terminal type hp is a standard 80 column by 24 line Hewlett-Packard terminal.

Your administrator may have set up your system and it may never ask you about your terminal type. In this case, check the *TERM* variable using the **env** command. If you are using a workstation with only one display, the *TERM* variable is probably set correctly and will not need to be changed.

If your terminal is acting strangely when you are using commands such as **more** and **vi**, check the *TERM* variable. If it is set correctly, execute the **tset** command, which resets the terminal characteristics using the terminal type found in the *TERM* variable.

© 2003 Hewlett-Packard Development Company, L.P.

7–14. TEXT PAGE: Common Variable Assignments

Common Variable Assignments

Variable names in **BOLD** denote variables you *would* customize.

`EDITOR=/usr/bin/vi`	Use vi commands for line editing.
`ENV=$HOME/.shrc`	Execute `$HOME/.shrc` at shell startup.
`FCEDIT=/usr/bin/vi`	Start `vi` edit session on previous command lines.
`HOME=/home/user3`	Designates your login directory.
~ (tilde)	POSIX shell equivalent for your HOME directory.
`HISTFILE=$HOME/.sh_history`	Defines file that stores all interactive commands entered.
`LOGNAME=user3`	Designates your login identifier or user name.
`MAIL=/var/mail/user3`	Designates your system mailbox.
`OLDPWD=/tmp`	Designates previous directory location.
`PATH=/usr/bin:$HOME/bin`	Designates directories to search for commands.
`PS1=`	Designates your primary prompt. `PS1='[!] $ '` Displays command line number with prompt. `PS1='$PWD $ '` Displays present working directory with prompt (NOTE: must be enclosed in single quotes('), not double quotes (")). `PS1='[!]$PWD $ '` Displays command line number and present working directory with prompt.
`PWD=/home/user3/tree`	Designates your present working directory.
`SHELL=/usr/bin/sh`	Designates your command interpreter program.

© 2003 Hewlett-Packard Development Company, L.P.

`TERM=2392a`	Designates the terminal type of your terminal. Use the command: `eval 'tset -s -Q -h'` During startup, it will read the file `/etc/ttytype` to map your terminal port with the appropriate terminal type. This is useful if you have different models of terminals attached to your system.
`TMOUT=300`	If no command or Return is entered in this number of seconds, the shell will terminate or time out.
`TZ=EST5EDT`	Defines the time zone the system should use to display the appropriate time.

The *TERM* Variable

The *TERM* variable must be properly defined so that the UNIX system knows the characteristics of your terminal. Many commands need to know what kind of terminal you are using, so that they can properly display their output. For example, `more` and `vi` must know how many lines and columns are on your display for proper screen control.

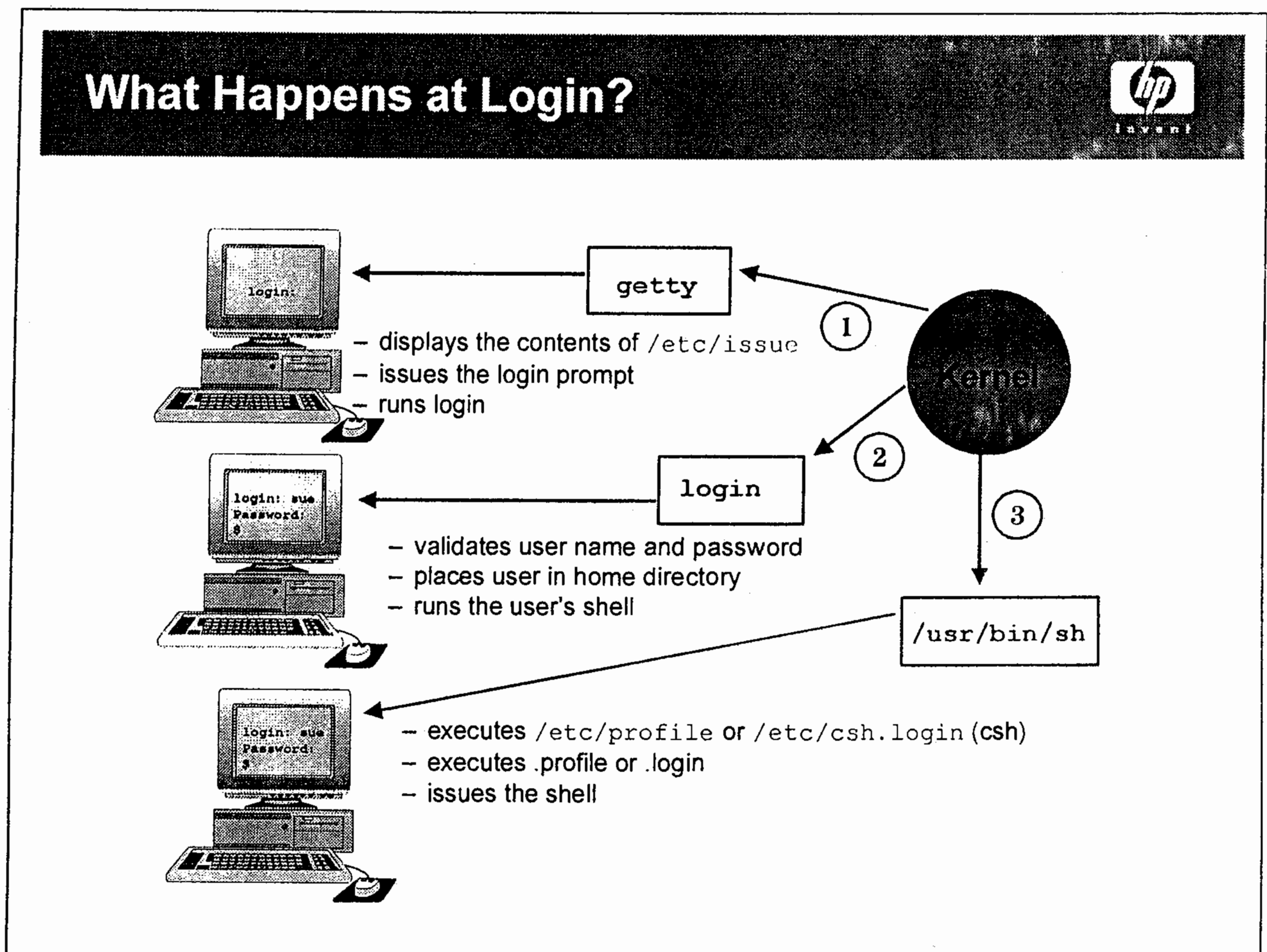
The *TERM* variable can be explicitly defined with a variable assignment, or assigned through the `tset` command, which depends on the terminal device you are connected to and the corresponding value in the file `/etc/ttytype`.

The following table summarizes some of the different terminal models and their associated *TERM* value. If your terminal model is not below, refer to the subdirectories under `/usr/lib/terminfo`.

Terminal Model	TERM value
HP 2392a	2392a
HP 70092	70092
HP 70094	70094
vt 100	vt100
Wyse 50	wy50
Medium resolution graphics display (512 x 600 pixels)	300l or hp300l
High resolution graphics display (1024 x 768 pixels)	300h or hp300h
HP 98550 display station (1280 x 1024 pixels)	98550, hp98550, 98550a, or hp98550a
HP 98720 or HP 98721 SRX (1280 x 1024 pixels)	98720, hp98720, 98720a, hp98720a, 98721, hp98721, 98721a, or hp98721a
HP 98730 or HP 98731 Turbo SRX (1280 x 1025 pixels)	98730, hp98730, 98730a, hp98730a, 98731, hp98731, 98731a, or hp98731a

© 2003 Hewlett-Packard Development Company, L.P.

7–15. SLIDE: What Happens at Login?



Student Notes

When you sit down to do work on the system, you see the login prompt on the screen. When you type your user name, the system reads your name and prompts you for a password. After you enter your password, the system checks your user name and password in the system password file (`/etc/passwd`). If the user name and password you entered are valid, the system places you in your home directory and starts the shell for you. We have seen this happen each time we logged in. The question is: What really happens when the shell is started?

1. `getty`
   a. Displays the contents of `/etc/issue`
   b. Issues the login prompt
   c. Runs login
2. `login`
   a. Validates user name and password
   b. Places user in home directory
   c. Runs the user's shell

© 2003 Hewlett-Packard Development Company, L.P.

3. **shell**
   a. Executes **/etc/profile** (POSIX, Bourne, and Korn shells) or **/etc/csh.login** (C shell)
   b. Executes **.profile** or **.login** in the user's home directory
   c. Executes **.kshrc** in the user's home directory (POSIX and Korn shells) if the user has created this file and if he has declared the ENV variable set to **.kshrc** in the **.profile** file
   d. Issues the shell prompt

Once the shell starts running, it reads commands from a system command file called **/etc/profile**. Whenever someone logs in and starts a shell, this file is read. In your home directory, there is also a file called **.profile**. After **/etc/profile** is read, the shell reads your own **.profile** . These two shell programs are used to customize a user's environment.

/etc/profile sets up the basic environment used by everyone on the system, and **.profile** further tailors that environment to your specific needs. Since everyone uses **/etc/profile**, the system administrator takes care of it. It is your responsibility, however, to maintain you own **.profile** to set up your user environment.

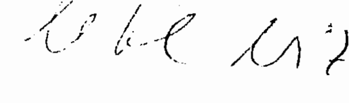
When these two programs are finished, the shell issues the first shell prompt.

A Note about CDE

If you are logging in with CDE, login profile scripts **/etc/profile**, **$HOME/.profile**, and **$HOME/.login** are normally not used by CDE. You may, however, force **$HOME/.profile** (for **sh** or **ksh** users) or **$HOME/.login** (for **csh** users) to be run by setting the following environment variable in **.dtprofile**:

```
export DTSOURCEPROFILE=true
```

Otherwise, only **.dtprofile** will be executed at login. **.dtprofile** contains commented lines of setup variables that you need to set the CDE environment.



A Note about Linux Login

The Linux login controls, while similar to HP-UX, can vary slightly. The login control files will be dependent on the user's login shell type and the graphical environment into which they login by default. Because much greater variation is possible, the controls cannot be covered extensively in these notes. However, where possible, Linux-specific information has been included.

© 2003 Hewlett-Packard Development Company, L.P.

7–16. SLIDE: The Shell Startup Files

The Shell Startup Files

If the Shell is ...	The Local Login Script is ...
Korn (`/usr/bin/ksh`)	`.profile` `.Kshrc`
Bourne (`/usr/old/bin/sh`)	`.profile`
POSIX (`/usr/bin/sh`)	`.profile` `.shrc`
Restricted (`/usr/bin/rsh, /usr/bin/rksh`)	`.profile`
C (`/usr/bin/csh`)	`.login` `.cshrc`

Student Notes

Some environment variables are required to configure your session (for example: *PATH*, *EDITOR*). As you may have seen, when these variables are defined interactively, they must be redefined *every* time you log in. To help you customize your session, the files **`.profile`**, **`.kshrc`**, **`.shrc`**, and **`.cshrc`** are available. These simple shell scripts define environment variables, define aliases, and execute programs upon login.

`.profile`

Any user who wants to customize the default environment provided by the system administrator could create or modify **`.profile`**. This file commonly defines or customizes environment variables, sets up the user's terminal, and executes programs, such as **`date`**, during session login. A user's application can also be initiated from **`.profile`** by **`exec applicationname.`** In this way, the user never has access to a shell prompt, and when the application is exited, the user is logged out.

`/etc/profile`

The file **`/etc/profile`** is a system-wide startup file that is executed by *all* users who are running under the Bourne, Korn, or POSIX shell. The system administrator may customize this to provide all users with a consistent user environment necessary to run their

© 2003 Hewlett-Packard Development Company, L.P.

applications. Regular users generally do not have write access to this file, so they are not allowed to modify its contents. Users will customize their environment through their personal copies of `.profile` or with the `.kshrc` or `.shrc` files.

`.kshrc` and `.shrc`

The Korn and POSIX shells have optional configuration files called, respectively, `.kshrc` and `.shrc`. These configuration files are used to configure your user environment, much like `.profile`. Unlike `.profile` however, `.kshrc` and `.shrc` are read every time you start a new shell, not just when you log in. This allows you to set up your aliases or even your prompt every time you start a shell. In an environment like X 11 Windows, you can have several shells running at once. You can use the `.kshrc` or `.shrc` file so that every one of those shells looks the same.

The file names `.kshrc` and `.shrc` are not required file names. When you invoke the shell, it looks for the file referenced by the *ENV* variable. This file is often named `.kshrc` or `.shrc`, but it may be named anything you wish.

To use your `.kshrc` or `.shrc` file, you must put a new environment variable in your `.profile` (and `.vueprofile` if you are using HP VUE). This is the *ENV* variable. Add these lines to your `.profile`:

```
ENV=~/.kshrc
export ENV
```

or

```
ENV=~/.shrc
export ENV
```

This tells the shell that you want to use the `.kshrc` (or `.shrc`) file in your home directory. Now just add all of your alias commands to `.kshrc` or `.shrc`.

If you are in an environment where you are using the Bourne and the POSIX shells, you may want to store POSIX shell specific variable assignments in this file, as it is *never* read by the Bourne shell.

`.cshrc` and `.login`

When you log in to the system with the C shell as your login shell, the shell searches your home directory for a file named `.login` . If found, the commands in the file are executed before you get your first shell prompt. This is exactly the same as the `.profile` file for the POSIX, Bourne, and Korn shells. If found, the commands in the file `.cshrc` are also executed before you get your first C shell prompt.

`.bash_profile`

When logging into a Linux system, the `.bash_profile` file is read instead of the `.profile` file. However, the `.profile` file can also be read by inserting an appropriate command into the `.bash_profile` file.

© 2003 Hewlett-Packard Development Company, L.P.

7–17. SLIDE: Shell Intrinsics versus UNIX Commands

Shell Intrinsics versus UNIX Commands

- Shell intrinsics are built into the shell.

  Examples:
  `cd`
  `ls`
  `pwd`
  `echo`

- UNIX commands live in `/usr/bin`.

  Examples:
  `more`
  `file`

- Some intrinsics are also available as separate commands.
  The system locates UNIX commands by using the PATH variable.

Student Notes

Some commands that you type at the keyboard are files in directories such as `/usr/bin`. These commands are UNIX commands. But many commands, such as `cd`, `pwd`, and `echo`, are actually built into the shell itself. These commands do not exist as files in the UNIX file system, but are like subroutines of the shell program. These commands are intrinsic shell commands.

Since UNIX commands can exist in several directories, the shell must know where to search for them. The *PATH* variable in your shell defines the directories to search and the order in which they are searched.

UNIX commands can have the same name as shell intrinsics; however, to access these commands, the user must use the command's absolute *PATH* name to inform the shell to use *it* rather than the intrinsic of the same name.

© 2003 Hewlett-Packard Development Company, L.P.

7–18. SLIDE: Looking for Commands — whereis

Looking for Commands — whereis

Syntax:

```
$ whereis [-b|-m|-s] command
```

Searches a list of directories for a command

Examples:

```
$ whereis if
if :
$
$ whereis ls
ls : /sbin/ls /usr/bin/ls /usr/share/man/man1.Z/ls.1
$
$  whereis cd
cd : /usr/bin/cd /usr/share/man/man1.Z/cd.1
$
$ whereis holdyourhorses
holdyourhorses :
$
```

Student Notes

UNIX stores its commands in four main directories: **/sbin** , **/usr/bin**, **/usr/local/bin**, and **/usr/contrib/bin**. The **whereis** command searches these as well as other directories to determine where a particular command lives. Many users also have a personal bin directory under their login directory. **whereis** does not search this directory. Sometimes you lose track of a command and its manual page. UNIX, through the **whereis** command, provides a way to locate commands and their manual pages.

The **whereis** command accepts a single argument that is the name of a command. It returns the location of the executable code and the manual page for the command.

© 2003 Hewlett-Packard Development Company, L.P.

The whereis command searches the following directories:

/usr/src/*	/usr/sbin	/sbin
/usr/bin	/usr/lbin	/usr/ccs/bin
/usr/share/man/*	/usr/local/man/*	/usr/local/bin
/usr/local/games	/usr/local/include	/usr/local/lib
/usr/contrib/man/*	/usr/contrib/bin	/usr/contrib/games
/usr/contrib/include	/usr/contrib/lib	/usr/share/man/$LANG/*
/usr/local/man/$LANG/*	/usr/contrib/man/$LANG/*	

If you want to change the directories the whereis command searches, use the flags -b, -m, or -s to limit the search to binary, manual pages, or source code, respectively.

© 2003 Hewlett-Packard Development Company, L.P.

7–19. TEXT PAGE: Sample .profile

Sample .profile

```
# Set up the command search paths:
PATH=.:/bin:/usr/bin ; export PATH

# Define the prompt:
PS1="$ " ; export PS1

# Set up the terminal:
# The -h option in the following tset command tells the shell to
# find the appropriate terminal type to assign to TERM from the
# file /etc/ttytype
eval `tset -s -Q -h`

# You could also hardcode your terminal type with:
#TERM=2392a

# Map control characters
# The intr "^C" maps Ctrl-c instead of DEL for program interrupt
stty erase "^H" kill "^U" intr "^C" eof "^D" susp "^S"
stty brkint hupcl ixon ixoff

# Uncomment the following line if you want to change default permissions
#umask 022

# Set up POSIX shell variables

# Inform the POSIX shell to reference the $HOME/.kshrc file
# Aliases are usually defined here
ENV=$HOME/.kshrc
export ENV

# The following variables are used to set up the command stack
# and the history feature
EDITOR=/usr/bin/vi; export EDITOR
HISTSIZE=50; export HISTSIZE
HISTFILE=$HOME/.sh_history; export HISTFILE
FCEDIT=/usr/bin/vi; export FCEDIT

# Run the script .logout to clean out the history file
# created by the POSIX shell command stack
trap "$HOME/.logout" 0

# The following lines can be updated for your application and uncommented
# if you want your application to start automatically when logging in
#exec /usr/bin/myapplicationname
```


© 2003 Hewlett-Packard Development Company, L.P.

7–20. TEXT PAGE: Sample .kshrc and .logout

Sample .kshrc

```
# Customize the prompt:
# The ! will display the command number in the prompt
#PS1='[!] $ '

# The $PWD will display the present working directory in the prompt
#PS1='[!] $PWD $ '

# The hostname will display the system name in the prompt
#PS1="[`hostname`] $ "

# Define some aliases
alias ls="ls -aCF"
alias history="fc -l"
alias h="fc -l"
alias r="fc -e - "
alias mroe=more

# Set up the shell environment
set -o markdirs      # All directory names resulting from filename
                     # generation will have a trailing / appended
set -o monitor       # Jobs will send messages to screen when complete
set -u               # Treat unset parameters as an error when substituting
```

Sample .logout

As you execute commands, they are appended to your designated history file (**$HISTFILE**). The POSIX shell does not provide an automatic mechanism to clean up this file. Therefore, you may want to execute the following when you log out. This moves the current history file to an *old* history file. The next time you log in, a new history file is generated. Therefore, this file does not grow unreasonably large. Note that in order to use this file with the POSIX shell, you must also have the appropriate trap set in the **.profile** file.

```
# Change to login directory
cd

# Save the current history file
mv $HOME/.sh_history $HOME/.sh_hist.old

# Send messages to the user
clear
echo `whoami` logged out at `date`
echo
```


© 2003 Hewlett-Packard Development Company, L.P.

7–21. LAB: Exercises

Directions

Complete the following exercises and answer the associated questions.

Linux Systems

In the exercise text, the key-sequences used in the HP-UX version of filename completion are used in the following questions. Please refer to the comparison chart provided earlier, for the appropriate filename completion key sequence for the shell you are using.

1. Create an alias called **h** that executes the **history** command.

2. Check the commands in the **.shrc** file in your home directory. Add your **h** alias to the list.

3. On the command line, set up an alias called **go** to change your working directory to **tree** and do an **ls -F**. Now type in the string **go** on the command line. What happens? Type **pwd** and see where you are. Now change back to your home directory. (Hint: Multiple commands can be entered on one line if separated with a semicolon.)

4. Log out and then log back in to test your aliases. Why did you have to log out?

© 2003 Hewlett-Packard Development Company, L.P.

5. Make sure you are in your home directory. What happens when you type `more f` [Esc] [Esc]? Using this command line, how can you make it display `funfile`?

6. From your **HOME** directory, copy the file `frankenstein` to the directory `tree/car.models/ford/sports`. Use file name completion to enter `frankenstein` and any other directory or file name in the directory path.

7. Type this incorrect command without pressing [Return]:

   ```
   cd /user/spol/ko/interface
   ```

   Using command line editing, correct the line to read:

   ```
   cd /usr/spool/lp/interface
   ```

   (Do *not* retype the command).

8. Execute the command `ls -F`.

   Recall this command line and change the `ls -F` to `ls -l`, using whatever `vi` editing commands are necessary. Re-execute the command.

9. Using the command stack, recall the previous copy command, and change `frankenstein` to `funfile`.

© 2003 Hewlett-Packard Development Company, L.P.

10. Recall the previous copy command, and modify it so that you display the contents of the sports directory.

11. Recall the previous list command, and modify it so that you *change directory* to the **sedan** directory (HINT: the path will be **tree/car.models/ford/sedan**). Use the **pwd** command to confirm your directory change.

12. Change back to your **HOME** directory; then use the **history** command or your **h** alias to recall your command stack; then use the **r** command to re-execute the command to return you to the **sedan** directory. Also, use the **r** command to display your present working directory.

© 2003 Hewlett-Packard Development Company, L.P.

© 2003 Hewlett-Packard Development Company, L.P.

Module 8 — Shell Advanced Features

Objectives

Upon completion of this module, you will be able to do the following:

- Use shell substitution capabilities, including variable, command, and tilde substitution.
- Set and modify shell variables.
- Transfer local variables to the environment.
- Make variables available to subprocesses.
- Explain how a process is created.

© 2003 Hewlett-Packard Development Company, L.P.

8–1. SLIDE: Shell Substitution Capabilities

Shell Substitution Capabilities

There are three types of substitution in the shell:

- Variable substitution
- Command substitution
- Tilde substitution

Student Notes

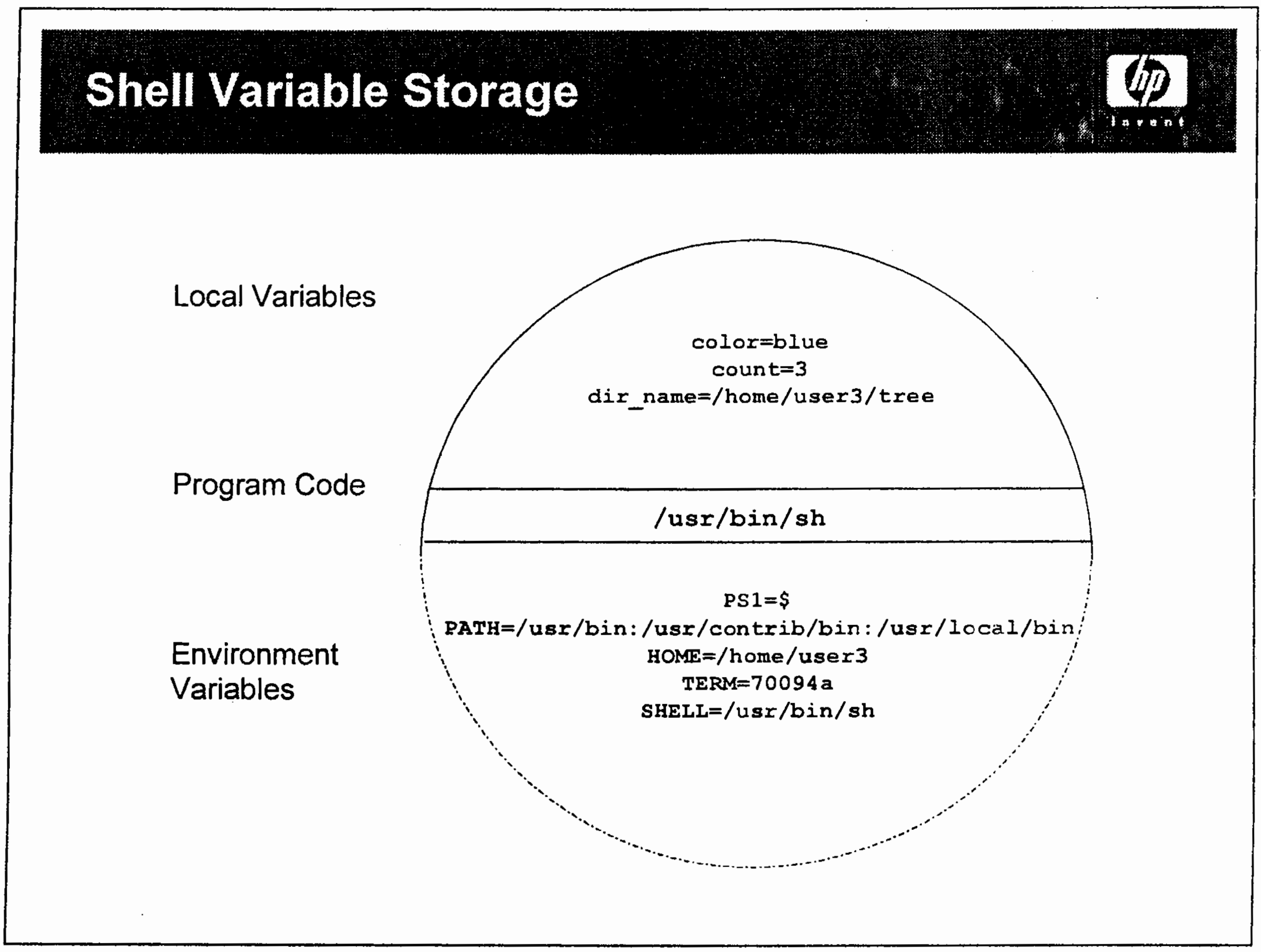
There are three types of substitution in the shell:

- Variable substitution
- Command substitution
- Tilde substitution

Substitution methods are used to speed up command-line typing and execution.

© 2003 Hewlett-Packard Development Company, L.P.

8–2. SLIDE: Shell Variable Storage



Student Notes

Built into the shell are two areas of memory for use with shell variables: the **local data area** and the **environment**. Memory is allocated from the local data area when a *new* variable is defined. The variables in this area are private to the current shell, and are often referred to as *local variables*. Any subsequent subprocesses will not have access to these local variables. However, subprocesses can access variables that are moved into the environment.

There are several special shell variables that are defined for you through your login process. Many of these variables are stored in the environment; some, such as PS1 and PS2, are usually stored in the local data area. You can change the values of these variables to customize the characteristics of your terminal session.

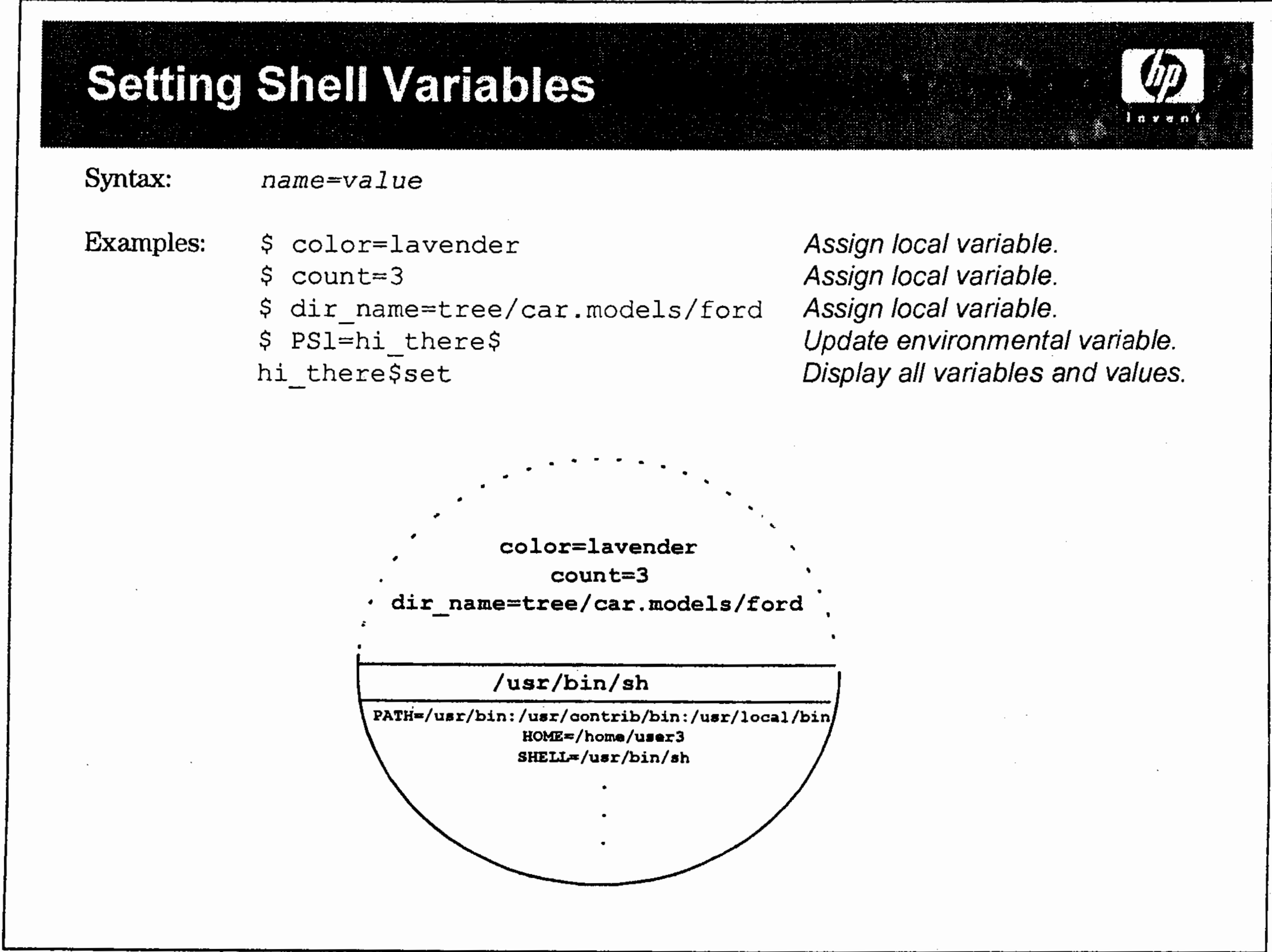
© 2003 Hewlett-Packard Development Company, L.P.

The **env** command can be used to display *all* of the variables that are currently held in the environment, for example:

```
$ env
MANPATH=/usr/share/man:/usr/contrib/man:/usr/local/man
PATH=/usr/bin:/usr/ccs/bin:/usr/contrib/bin:/usr/local/bin
LOGNAME=user3
ERASE=^H
SHELL=/usr/bin/sh
HOME=/home/user3
TERM=hpterm
PWD=/home/user3
TZ=PST8PDT
EDITOR=/usr/bin/vi
```


© 2003 Hewlett-Packard Development Company, L.P.

8–3. SLIDE: Setting Shell Variables

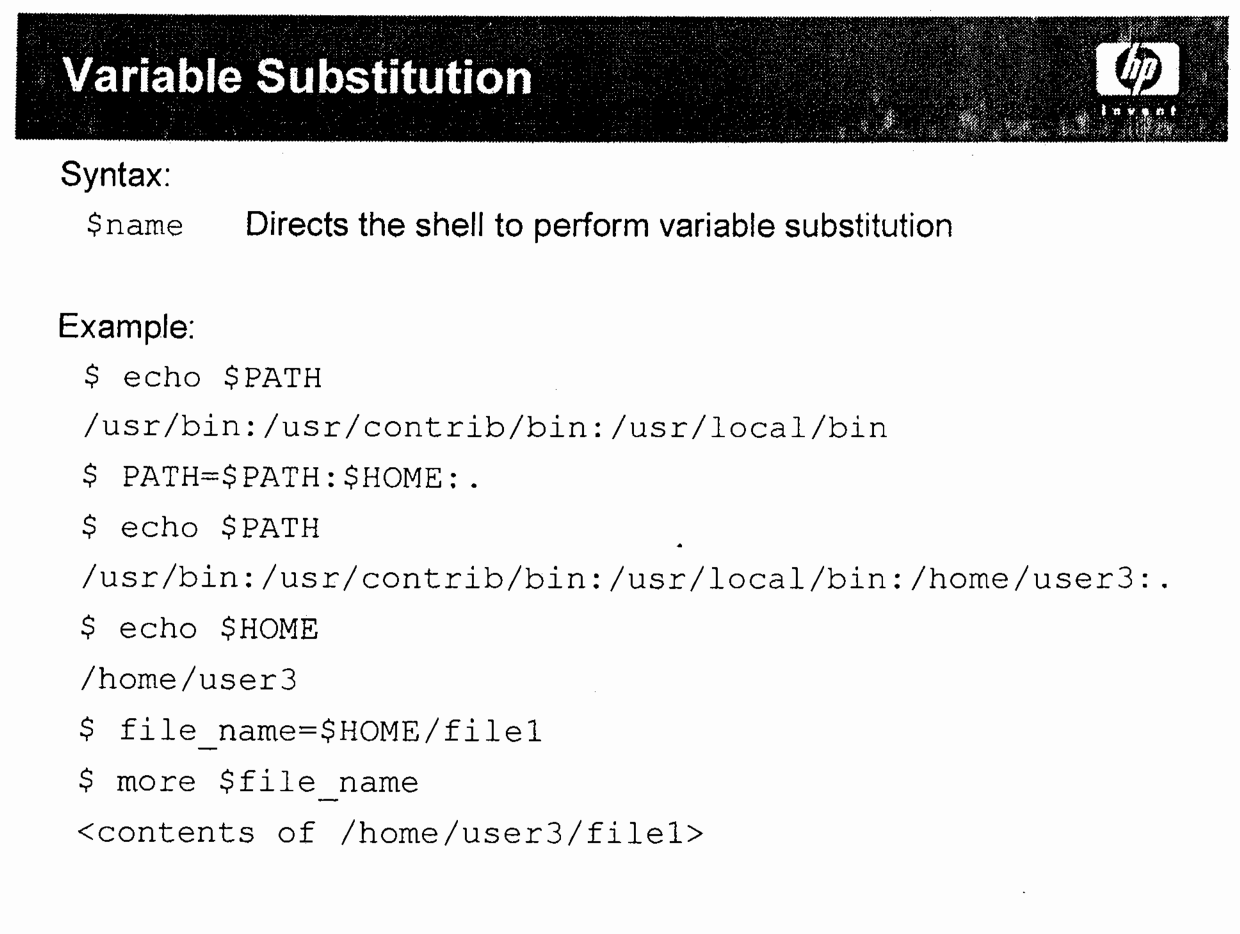


Student Notes

When a user creates a new variable, such as *color*, it will be stored in the local data area. When assigning a new value to an existing environment variable such as *PATH*, the new value replaces the old value in the environment.

© 2003 Hewlett-Packard Development Company, L.P.

8–4. SLIDE: Variable Substitution



Student Notes

Each defined variable has an associated value. When a variable name is immediately preceded by a dollar sign ($), the shell replaces the parameter with the value of the variable. This procedure is known as **variable substitution.** It is one of the tasks the shell performs *before* executing the command entered on the command line. After the shell has made all of the variable substitutions on the command line, it will execute the command. Therefore, variables can also represent commands, command arguments, or a complete command line, providing a convenient mechanism to rename frequently issued long path names or long command strings.

Examples

This slide demonstrates some uses of shell variables. Notice that variable substitution can appear anywhere in the command line, and multiple variables can be referenced in one command line. As seen on the slide, an existing value of a variable can even be used to update the current value of the variable.

```
$ echo $PATH
/usr/bin:/usr/contrib/bin:/usr/local/bin
$ PATH=$PATH:$HOME:.
```


© 2003 Hewlett-Packard Development Company, L.P.

```
$ echo $PATH
/usr/bin:/usr/contrib/bin:/usr/local/bin:/home/user3:.
$ echo $HOME
/home/user3
$ file_name=$HOME/file1                    file_name=/home/user3/file1
$ more $file_name                          more /home/user3/file1
<contents of /home/user3/file1>
```

NOTE: The **echo** ***$name*** command provides an effective method to display the current value of a variable.

The Use of {}

Assume you have a variable called *file* and a variable called *file1*. They can be assigned with the following statements:

```
$ file=this
$ file1=that
$ echo $fileand$file1                     looks for variables fileand, file1
sh: fileand: parameter not set
$ echo ${file}and$file1                   looks for variables file, file1
thisandthat
```

The curly braces can be used to delimit the variable name from the surrounding text.

© 2003 Hewlett-Packard Development Company, L.P.

8–5. SLIDE: Variable Substitution (Continued)

Variable Substitution (Continued)

```
$ dir_name=tree/car.models/ford
$ echo $dir_name
tree/car.models/ford
$ ls -F $dir_name
sedan/   sports/
$ my_ls="ls -aFC"
$ $my_ls
./                file.1            tree/
../               file.2
$ $my_ls $dir_name
./        ../      sedan/   sports/
$ cd /tmp
$ dir_name=/home/user2/tree/dog.breeds/retriever
$ $my_ls $dir_name
./         ../        golden     labrador   mixed
```

Student Notes

The use of an *absolute path name* for the value of a variable that references a file or directory allows you to be anywhere in the file hierarchy and still access the desired file or directory.

© 2003 Hewlett-Packard Development Company, L.P.

Consider the examples on the slide:

```
$ dir_name=tree/car.models/ford
$ echo $dir_name                    echo tree/car.models/ford
tree/car.models/ford
$ ls -F $dir_name                   ls -F tree/car.models/ford
sedan/ sports/
$ my_ls="ls -aFC"                   use quotes so shell ignores space
$ $my_ls                            ls -aFC
./   file.1   tree/
../   file.2
$my_ls $dir_name                    ls -aFC tree/car.models/ford
./   ../   sedan/   sports/
$ cd /tmp
$ dir_name=/home/user2/tree/dog.breeds/retriever
$ $my_ls $dir_name                  ls -aFC /home/user2/tree/dog.breeds/retriever
./   ../ golden labrador mixed
```


© 2003 Hewlett-Packard Development Company, L.P.

8–6. SLIDE: Command Substitution

Student Notes

Command substitution is used to replace a command with its output within the same command line. The standard syntax for command substitution, and the one encouraged by POSIX, is **$(*command*)**.

Command substitution allows you to capture the output of a command and use it as an argument to another command or assign it to a variable. As in variable substitution, the command substitution is performed before the leading command on the command line. When the command output contains carriage return/line feeds, they will be replaced with blank spaces.

Command substitution is invoked by enclosing the command in parentheses preceded by a dollar sign, similar to variable substitution.

Any valid shell script can be put in command substitution. The shell scans the line and executes any command it sees after the opening parenthesis until a matching, closing parenthesis is found.

An alternate form of command substitution uses grave quotes surrounding the command, as in **`command`**.

© 2003 Hewlett-Packard Development Company, L.P.

It is equivalent to **$(*command*)**, and is the only form recognized by the Bourne shell. The \`*command*\` form should be used in scripts that may be run by the POSIX, Korn, and Bourne shells.

Examples

Command substitution is very commonly used to assign the output of a command to a variable for later reference or manipulation. Normally the **pwd** command sends its output to your screen. When you execute the assignment:

```
$ curdir=$(pwd) OR $ curdir=`pwd`
```

The output of the **pwd** command is assigned to the variable **curdir**.

Consider this example:

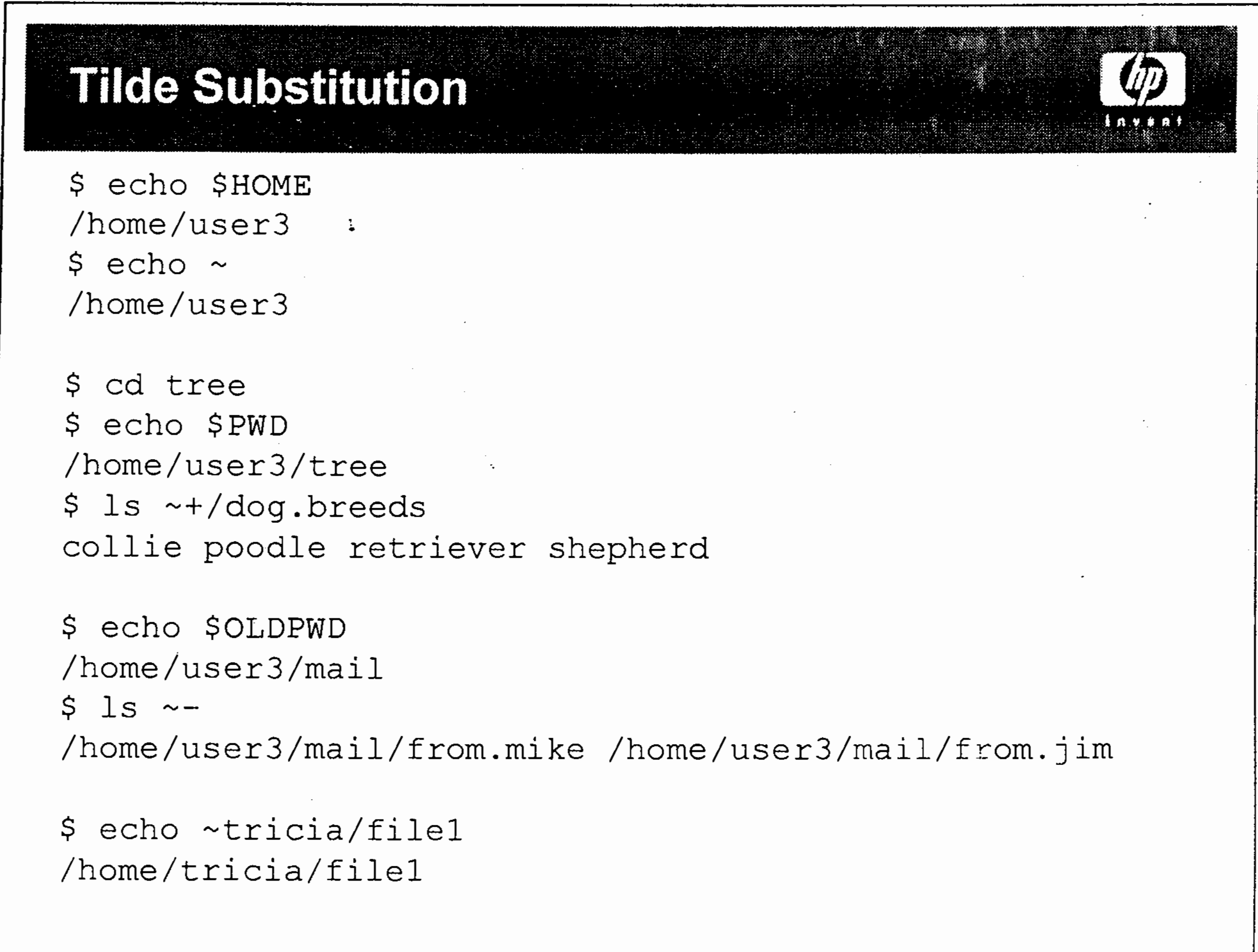
```
$ echo date
date
$ banner date
#####       #     #######   ######
#    #     # #       #      #
#     #   #   #      #      ######
#    #   # ### #     #      #
#####   #       #    #      ######$
echo $(date)                          executes: echo Thu Jul 11 16:40:32 EDT 1994
Thu Jul 11 16:40:32 EDT 1994
$ banner $(date)                      executes: banner Thu Jul 11 16:40:32 EDT 1994

#######  #    #  #    #          #  #    #  #      ##    ##
   #     #    #  #    #          #  #    #  #       #     #
   #     ######  #    #          #  #    #  #       #     #
   #     #    #  #    #     #    #  #    #  #       #     #
   #     #    #  ######      ####    ####   #####  ###   ###
```

Normally, the **date** command sends its output to your screen. When the command **banner date** is executed, the string *date* is bannered. In the second example, when *date* is used with command substitution, the shell first executes the **date** command, and replaces the **date** argument with the output of the **date** command. Therefore, it will display the ten first characters of **banner Thu Jul 11 16:40:32 EDT 1994**.

NOTE: The Linux version of the banner command is not intended to produce output on-screen, as shown in the examples above. Therefore, the **banner** command is not recommended for use by Linux users.

© 2003 Hewlett-Packard Development Company, L.P.

8–7. SLIDE: Tilde Substitution



Student Notes

If a word begins with a tilde (~), tilde expansion is performed on that word. Note that tilde expansion is provided only for tildes at the beginning of a word, that is, **`/~home/user3`** has no tilde expansion performed on it. Tilde expansion is performed according to the following rules:

- A tilde by itself or in front of a / is replaced by the path name set in the *HOME* variable.
- A tilde followed by a + is replaced with the value of the *PWD* variable. *PWD* is set by **`cd`** to the new, current, working directory.
- A tilde followed by a minus (-) is replaced with the value of the *OLDPWD* variable. *OLDPWD* is set by **`cd`** to the previous working directory.
- If a tilde is followed by several characters and then a /, the shell checks to see if the characters match a user's name on the system. If they do, then the **`~characters`** sequence is replaced by that user's login path.

© 2003 Hewlett-Packard Development Company, L.P.

Tildes can be put in aliases:

```
$ pwd
/home/user3
$ alias cdn='cd ~/bin'
$ cdn
$ pwd
/home/user3/bin
```

NOTE: The default Linux permission settings may prevent users from accessing other users' **$HOME** directories.

© 2003 Hewlett-Packard Development Company, L.P.

8–8. SLIDE: Displaying Variable Values

Displaying Variable Values

```
$ echo $HOME
/home/user3

$ env
HOME=/home/user3
PATH=/usr/bin:/usr/contrib/bin:/usr/local/bin
SHELL=/usr/bin/sh

$ set
HOME=/home/user3
PATH=/usr/bin:/usr/contrib/bin:/usr/local/bin
SHELL=/usr/bin/sh
color=lavender
count=3
dir_name=/home/user3/tree

$ unset dir_name
```

Student Notes

Variable substitution, ***$variable***, can be used to display the value of an individual variable, regardless of whether it is in the local data area or the environment.

The **env** command can be used to display *all* of the variables that are currently held in the environment.

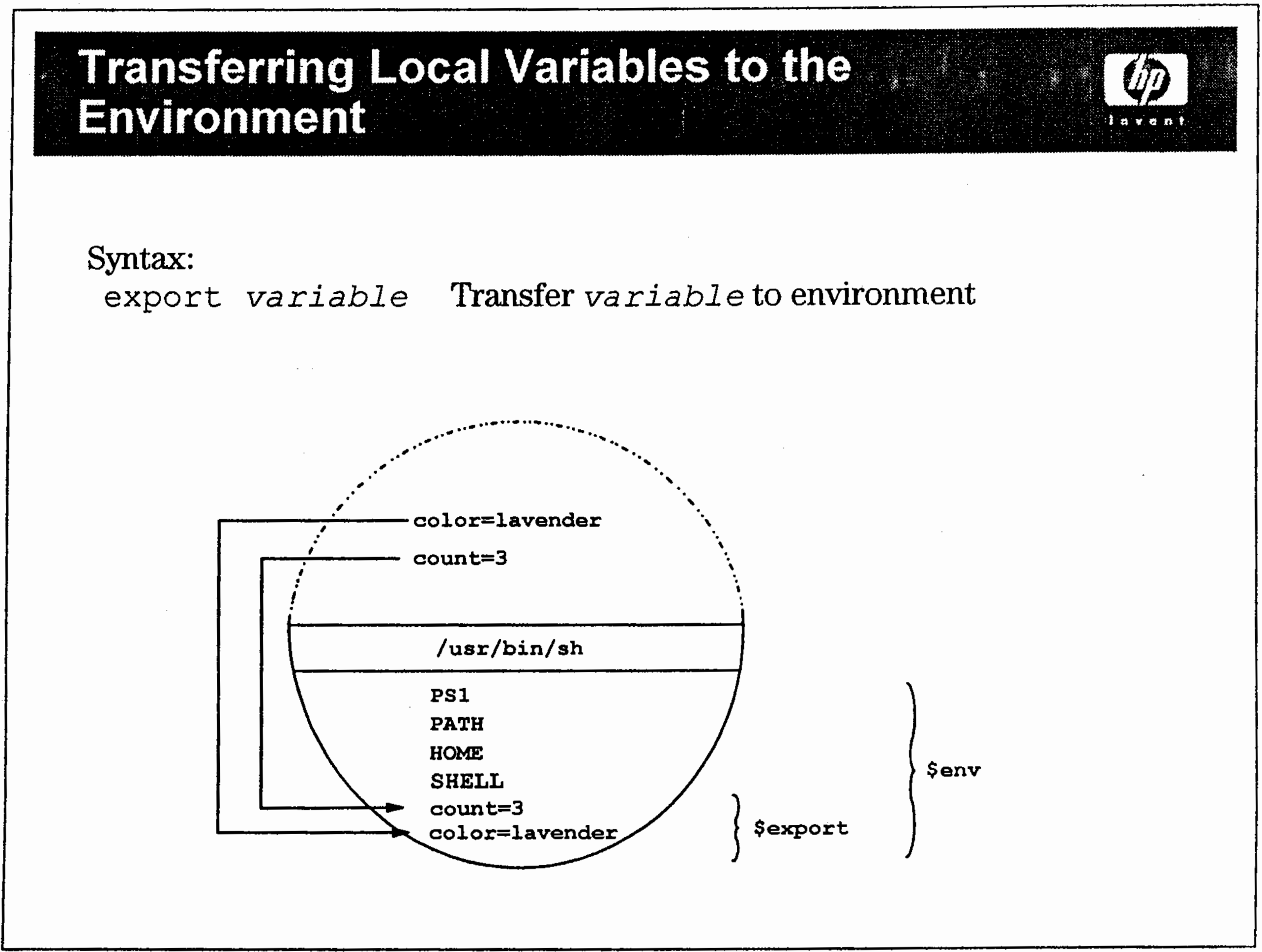
The **set** command will display *all* of the currently defined variables, local and environment, and their values.

The **unset** command can be used to remove the current value of the specified variable. The value is effectively assigned to NULL.

Both **set** and **unset** are shell built-in commands. **env** is the UNIX command **/usr/bin/env**.

© 2003 Hewlett-Packard Development Company, L.P.

8–9. SLIDE: Transferring Local Variables to the Environment



Student Notes

The diagram on the slide illustrates transferring the variables *color* and *count* into the environment by executing the following commands:

```
$ color=lavender
$ export color
$ export count=3
$ export
export PATH=/usr/bin:/usr/ccs/bin:/usr/contrib/bin:/usr/local/bin
export color=lavender
export count=3
```

In order for a variable to be available to other processes, it must exist in the environment. When a variable is defined, it is stored in the local data space and must be **exported** to the environment.

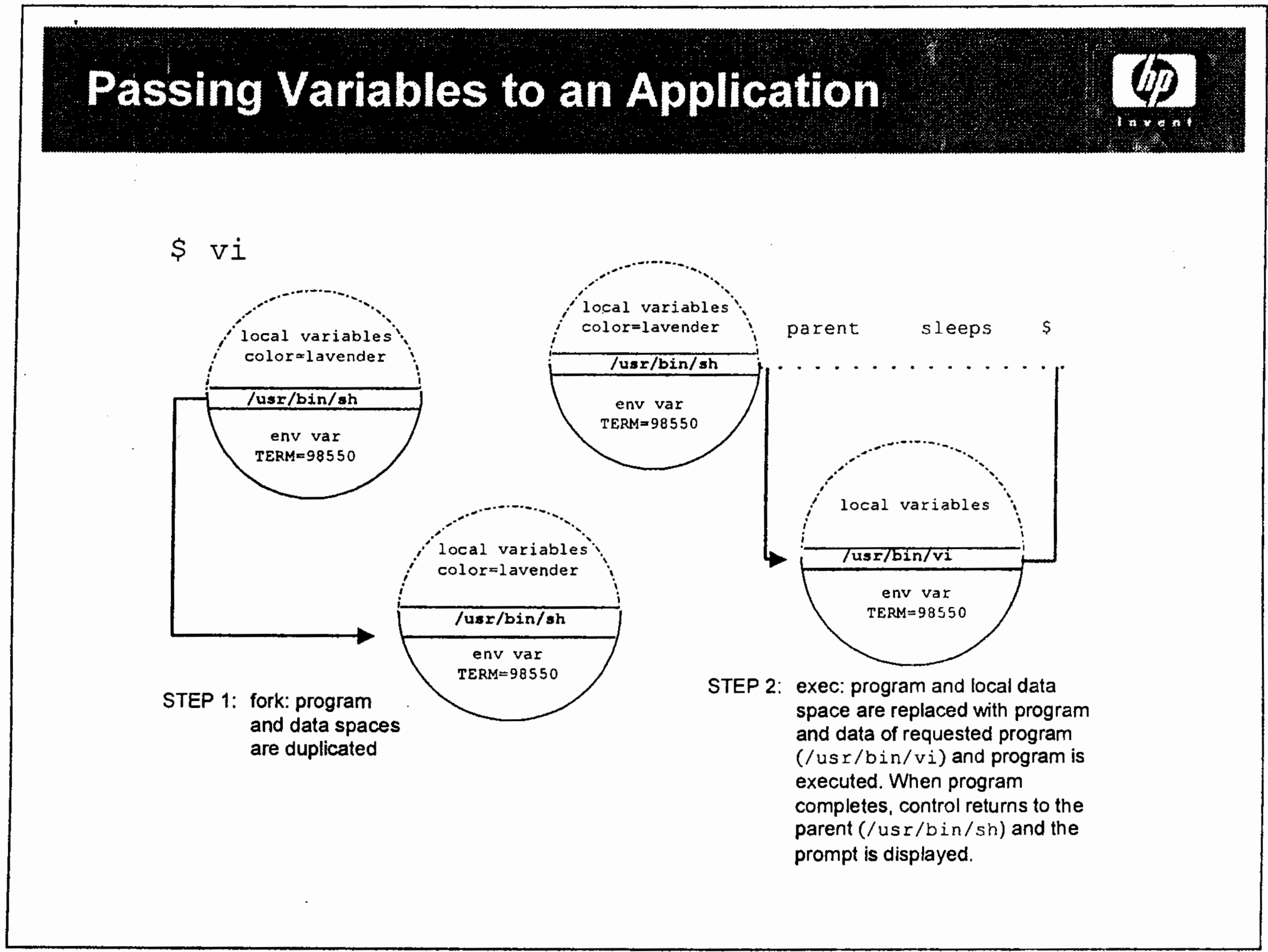
The **`export`** ***`variable`*** command will transfer the specified variable from the local data space to the environment data space. **`export`** ***`variable=value`*** will assign (possibly update) the value of a variable, and place it in the environment. With no arguments, the

© 2003 Hewlett-Packard Development Company, L.P.

`export` command is similar to the `env` command in that it will display the names and values of all exported variables. Note that `export` is a shell built-in command.

© 2003 Hewlett-Packard Development Company, L.P.

8–10. SLIDE: Passing Variables to an Application



Student Notes

Every application or command on the system will have an associated program file stored on the disk. Many of the standard UNIX system commands are found under the directory **`/usr/bin`**. When a command is requested to run, the associated program file must be located, the code loaded into memory and then executed. The running program is known as a UNIX system **process**.

When you log in to your UNIX system, the shell program will be loaded, and a shell process executed. When you enter the name of an application (or command) to run at the shell prompt, a **child** process is created and executed through:

1. A **`fork`**, which duplicates your shell process, including the program code, the environment data space, and the local data space.

2. An **`exec`**, which replaces the code and local data space of the child process with the code and local data space of the requested application.

3. The **`exec`** will conclude by executing the requested application process.

While the child process is executing, the shell (the **parent**) will sleep, waiting for the child to finish. Once the child finishes execution, it terminates, releases the memory associated with

© 2003 Hewlett-Packard Development Company, L.P.

its process, and wakes up the parent who is now ready to accept another command request. You know the child process has concluded when the shell prompt returns.

Local versus Environment Variables

Anytime a new variable is defined, it will be stored in the local data area associated with the process. If a child process requires access to this variable, the variable must be transferred into the environment using **`export`**. Once a variable is in the environment, it will be made available to *all* subsequent child processes, because the environment is propagated to each child process.

On the slide, before the **`vi`** command is issued, the **`color`** variable is in the shell's local data area, and the **TERM** variable is in the environment. When the **`vi`** command is issued, the shell performs a fork and exec; the local data area of the child process is overwritten by the child's program code, but the environment is passed, intact, to the child process. Therefore the child process **`vi`** does *not* have access to the **`color`** variable, but it *does* have access to the **TERM** variable. The vi editor needs to know the type of terminal the user is using to properly format its editing screen. It gets this information by reading the value in the **TERM** variable, which is available in its environment.

Therefore we see that one way of passing data to (child) processes is through the environment.

© 2003 Hewlett-Packard Development Company, L.P.

8–11. SLIDE: Monitoring Processes

Monitoring Processes

```
$ ps -f
   UID  PID  PPID    C    STIME   TTY      TIME COMMAND
 user3 4702     1    1  08:46:40 ttyp4     0:00 -sh
 user3 4895  4702   18  09:55:10 ttyp4     0:00 ps -f

$ ksh
$ ps -f
   UID  PID  PPID    C    STIME   TTY      TIME COMMAND
 user3 4702     1    0  08:46:40 ttyp4     0:00 -sh
 user3 4896  4702    1  09:57:20 ttyp4     0:00 ksh
 user3 4898  4896   18  09:57:26 ttyp4     0:00 ps -f

$ exec ps -f
   UID  PID  PPID    C    STIME   TTY      TIME COMMAND
 user3 4702     1    0  08:46:40 ttyp4     0:00 -sh
 user3 4896  4702   18  09:57:26 ttyp4     0:00 ps -f
$
```

Student Notes

Every process that is initiated on the system is assigned a unique identification number, known as a process ID (**PID**). The **ps** command displays information about processes currently running (or sleeping) on your system, including the PID of each process and the PID of each process' parent (**PPID**). Through the PID and PPID numbers, you can trace the lineage of any process that is running on your system. The **ps** command will also report who owns each process, which terminal each process is executing through, and additional useful information.

The **ps** command is commonly invoked with no options, which gives a short report about processes associated only with your terminal session, as follows:

```
$ ps
PID TTY     TIME COMMAND
4702 ttyp4   0:00 sh
4894 ttyp4   0:00 ps
```

As you can see above, the command reveals that only the shell, **sh**, and the **ps** command are running. Observe the PID numbers of the two processes. When invoked with the **-f** option, as seen on the slide, the **ps** command produces a *full* listing, which includes the PPID

© 2003 Hewlett-Packard Development Company, L.P.

numbers, plus additional information. We can see that the `ps -f` command runs as a child of the shell `sh` because its PPID number is the same as the PID number of the shell.

Remember that a shell is a program just like any other UNIX command. If we issue the `ksh` command at our current POSIX shell prompt, a `fork` and `exec` will take place, and a Korn shell child process will be created and will start executing. When we then execute another `ps -f`, we see that, as expected, `ksh` runs as a child of the original shell, `sh`, and the new `ps` command runs as a child of the Korn shell.

The `exec` command is available as a shell built-in command. If instead of running `ps -f` in the usual way, we instead `exec ps -f`, the program code for `ps` will overwrite the program code for the current process (`ksh`). This is evident because the PID of the `ps -f` is the same number as `ksh` used to be. When `ps -f` terminates, we will find ourselves back at our original POSIX shell prompt.

© 2003 Hewlett-Packard Development Company, L.P.

8–12. SLIDE: Child Processes and the Environment

Child Processes and the Environment

```
 export color=lavender
$ ksh          (create child shell process)
$ ps -f
   UID     PID  PPID  C  STIME    TTY      TIME COMMAND
 user3    4702     1  0 08:46:40 ttyp4    0:00 -sh
 user3    4896  4702  1 09:57:20 ttyp4    0:00 ksh
 user3    4898  4896 18 09:57:26 ttyp4    0:00 ps -f
$ echo $color
lavender
$ color=red
$ echo $color
red
$ exit          (exit child shell)
$ ps -f         (back in parent shell)
   UID     PID  PPID  C  STIME    TTY      TIME COMMAND
 user3    4702     1  0 08:46:40 ttyp4    0:00 -sh
 user3    4895  4702  1 09:58:20 ttyp4    0:00 ps -f
$ echo $color
lavender
```

Student Notes

The slide illustrates that child processes cannot alter their parent process' environment.

```
$ ps -f
   UID    FSID            PID   PPID C STIME    TTY   TIME COMMAND
  user3 default_system 4702      1 0 08:46:40 ttyp4 0:00 -sh
  user3 default_system 4895   4702 1 09:58:20 ttyp4 0:00 ps -f
```

If an initial **ps -f** command were executed, it would reveal that only our login shell, **sh** (and **ps**, of course) is running. As seen on the slide, we will assign the value of *lavender* to the variable *color* and export it into the environment. Next we will execute a child process. The **ksh** command is invoked, creating a child Korn shell process. The **ps -f** command that follows provides confirmation. Of course the parent shell's environment has been passed to the child Korn shell, and we observe that the variable *color* has the value *lavender*. We will then change the value of the variable *color* by assigning a value of *red*. The **echo** command confirms that the value of the variable *color* has changed in the child shell's environment. When we exit the child shell and return to the parent shell, we see that the parent's environment has *not* been altered by the child process, and the variable *color* has retained the value *lavender*.

© 2003 Hewlett-Packard Development Company, L.P.

8–13. LAB: The Shell Environment

Directions

Complete the following exercises and answer the associated questions.

1. Using command substitution, assign today's date to the variable *today*.

2. What is an easy way to list the contents of another user's home directory?

?

3. Set a shell variable named *MYNAME* equal to your first name. How do you see the contents of that variable?

4. Now start a child shell by typing **sh**. Look at the contents of *MYNAME* now. What happened? Exit the child shell (use Ctrl+ d or **exit**). Does the parent still know about the variable *MYNAME*?

5. Enter the command in the parent shell to enable the child to see the contents of *MYNAME*. How can you see all variables that the child shell will inherit?

© 2003 Hewlett-Packard Development Company, L.P.

6. Start another child shell. Look at the variable *MYNAME*. Now set the variable *MYNAME* equal to your partner's name. Is *MYNAME* now a local or environment variable? List the environment variables. What is *MYNAME* set to?

7. Now remove the variable *MYNAME* from the child shell. Does *MYNAME* exist either locally or within the environment of the child shell? Why or why not?

8. Kill the child shell and return to your LOGIN shell. Does **MYNAME** still exist? Why or why not? What commands did you use to verify this?

9. Modify your shell prompt so that it displays: **`good_day$`**. What happens to your prompt when you log out and log back in?

10. Modify your shell prompt so that it displays your user identification name. For example, if you are logged in as **`user3`**, the prompt will display **`user3$`**. (Hint: Is there an environment variable that stores your login identifier?)

© 2003 Hewlett-Packard Development Company, L.P.

11. Set a variable *dir* equal to **`/usr/bin/ls`**. How can you use the value of this variable to execute the **`ls`** command? Will the variable *dir* accept directory or file name argument?

© 2003 Hewlett-Packard Development Company, L.P.

Module 9 — File Name Generation

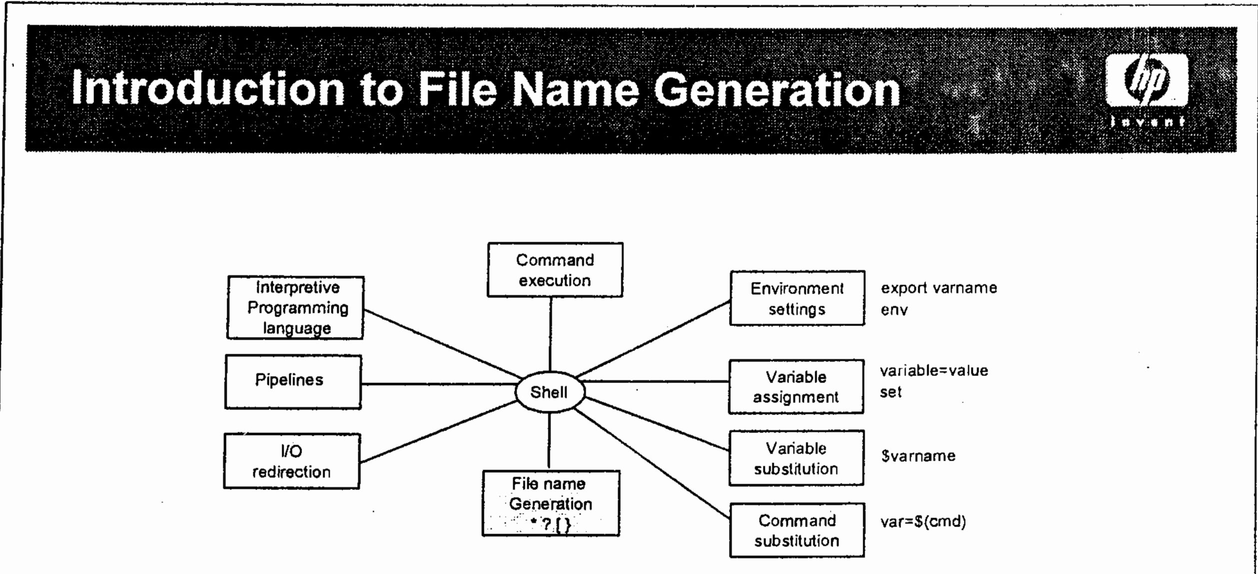
Objectives

Upon completion of this module, you will be able to do the following:

- Use file name generation characters to generate file names on the command line.
- Save typing by using file name generating characters.
- Name files so that file name generating characters will be more useful.

© 2003 Hewlett-Packard Development Company, L.P.

9–1. SLIDE: Introduction to File Name Generation



- File name generating characters are interpreted by the shell.
- The shell will generate file names that satisfy the requested pattern.
- File name generation is done before the command is executed.
- The command will operate on the generated file names.

Student Notes

The shell provides a timesaving feature for typing file names. The feature is called **file name generation**, or **file name expansion**. You can find file names that match a pattern, for example, all file names that end in `.c` or all file names that begin with `draw`. You enter special characters that can stand for one or more characters in a file name. The shell will expand the requested file name pattern into the corresponding file names *before* the command is executed. Therefore, the file name generating characters can save you a lot of typing.

The file name-generating feature is useful because most applications will define naming conventions for their files. Once you know what the naming conventions are, you can use file name expansion to access just the files whose names contain the desired pattern. For example, source code for C programs conventionally ends in `.c`, and word processors may use `.doc` as an extension for document files.

© 2003 Hewlett-Packard Development Company, L.P.

9–2. SLIDE: File Name Generating Characters

File Name Generating Characters

?	Matches any single character except a leading dot
[ ]	Defines a class of characters
–	Defines an inclusive range
!	Negates the defined class
*	Matches zero or more characters except a leading dot

Student Notes

The special characters that are interpreted by the shell for file name generation are:

?	Matches any single character (except a leading dot).
[ ]	Defines a class of characters from which one will be matched (unless it is a leading dot). Within this class, a hyphen (-) can be used between two ASCII characters to mean *all* characters in that range, inclusive, and an exclamation point (!) can be used as the first character to negate the defined class.
*	Matches zero or more characters (except a leading dot).

We will see each of these characters in detail.

© 2003 Hewlett-Packard Development Company, L.P.

9–3. SLIDE: File Name Generation and Dot Files

File Name Generation and Dot Files

- File name generating characters will *never* generate a file name that has a *leading* dot.
- The leading dot in dot files must be explicitly provided.

Student Notes

Dot files are files whose names begin with the dot (`.`) character, such as `.profile`, `.kshrc` and `.exrc`. These files are normally hidden; you must use the `ls -a` command to display these file names.

The dot files are hidden from the file name generating characters as well. Therefore, the file name generating characters will never generate a file name that begins with a leading dot. If you would like to display the file names that begin with a dot, you will need to explicitly provide the leading dot as part of the file name pattern that you are trying to match.

© 2003 Hewlett-Packard Development Company, L.P.

9–4. SLIDE: File Name Generation — ?

File Name Generation — ?

? matches any single character.

```
$ ls -a
. .. .zz abc abcd abcdef abcz bbabb cyz zzayy

$ echo ???   Executes:  echo abc cyz
$ echo abc?  Executes:  echo abcd abcz
$ echo ??a?? Executes:  echo bbabb zzayy
$ echo .??   Executes:  echo .zz
$ echo ?     Executes:  echo ?
```

Student Notes

A question mark matches any single character, but it will not match a leading dot.

File name generation is accomplished by the shell before commands are invoked. Thus, in the example, the shell generates file names that match the patterns specified. All resulting file names are passed as arguments to the echo command. If there is no match, then the pattern itself is passed as the argument.

NOTE:	The file name generating feature is more commonly used with file manipulation commands such as `ls`, `more`, and `cp`. The `echo` command is useful for confirming how the shell will expand the requested pattern, especially when using destructive commands such as `rm`. *Remember: once a file is removed, it is gone.*

© 2003 Hewlett-Packard Development Company, L.P.

9–5. SLIDE: File Name Generation — [ ]

File Name Generation — [ ]

[ ] defines a class of characters from which one will be matched.

```
$ ls -a
. .. .zz 1G 2G 7G 15G Ant Cat Dog abc abcdef ba cyz

$ echo [abc]??        Executes:  echo abc cyz
$ echo [1-9][A-Z]     Executes:  echo 1G 2G 7G
$ echo [!A-Z]??       Executes:  echo 15G abc cyz
```

Student Notes

Brackets are used to specify a **character class**. A character class matches any single character from the enclosed list.

An exclamation point (`!`) as the first item inside the bracket negates the character class; that is, the class stands for the class of all characters *not* listed inside the brackets.

If a hyphen (`-`) is placed between two characters within brackets, the character class will be all characters in the ASCII sequence — see **`ascii(5)`** — from the first character to the last one inclusive. Thus the classes **`[!123456789]`** and **`[!1-9]`** both stand for any character except the digits 1 through 9.

A leading dot (`.`) cannot be matched with a character class.

© 2003 Hewlett-Packard Development Company, L.P.

9–6. SLIDE: File Name Generation — *

File Name Generation — *

* matches zero or more characters except a leading dot (.).

```
$ ls -a
. .. .profile ab.dat abcd.dat abcde abcde.data
$ echo *
Executes: echo ab.dat abcd.dat abcde abcde.data
$ echo .*
Executes: echo . .. .profile
$ echo *.dat
Executes: echo ab.dat abcd.dat
$ echo *e
Executes: echo abcde
```

Student Notes

An asterisk (*) matches any string of zero or more characters.

As usual in file name generation, an asterisk (*) will not match a leading dot.

© 2003 Hewlett-Packard Development Company, L.P.

9–7. SLIDE: File Name Generation — Review

File Name Generation — Review

```
$ ls -a
  .             Abc      e35f
  ..            Abcd     efg
  .test1        abc      fe3f
  .test2        abcdemf  fe3fg
```

Given the above directory, list all file names that

- contain *only* five characters
- contain *at least* five characters
- begin with an "a" or an "A"
- have *at least* four characters and begin with an "a" or an "A"
- end with the sequence "e", a single number, and an "f"
- begin with a dot
- begin with a dot, except .
- begin with a dot, except . and ..

Student Notes

The slide shows a directory listing. Determine the file name generation designations that will display the requested file name patterns.

The file names can be found under the **`filegen`** directory under your HOME directory.

© 2003 Hewlett-Packard Development Company, L.P.

9–8. LAB: File Name Generation

Directions

Complete the following exercises and answer the associated questions.

1. Change to your HOME directory, then type the command `ls *` and explain the output.

2. If the command `echo ???XX` produces the output `???XX`, what does it mean?

3. From your HOME directory, what command would you issue to do the following?
   a. Display all file names that end in `.c`.
   b. Display just the `.c` files associated with `mod`.
   c. Display all file names that contain `file`.
   d. Display all file names that end in `.c`, `.f` or `.p`.

4. Create a directory called `c_source`. Move all of your `.c` files to the `c_source` directory using the file name generating characters.

5. Create a directory called `dir_1` under your HOME directory. What happens when you issue the command: `cd dir*`?

© 2003 Hewlett-Packard Development Company, L.P.

6. Go back to your HOME directory and create directories called `dir_2`, `dir_3` and `dir_4`. Now try `cd dir*` again and explain what happens.

7. Using the `touch` command (syntax: `touch` ***`filename`***), create files so that the following will be true:

   The pattern `?XX` will match exactly ONE file name.
   The pattern `?.XX` will match exactly TWO file names.
   The pattern `*XX` will match exactly THREE file names.
   The pattern `XX.??` will match exactly ONE file name.
   The pattern `XX.*` will match exactly TWO file names.

   Use the `echo` command to check your results.

8. Use a single `rm` command to remove all of the files created in the previous exercise. (Hint: you might want to use the `rm -i` command.)

© 2003 Hewlett-Packard Development Company, L.P.

Module 10 — Quoting

Objectives

Upon completion of this module, you will be able to do the following:

- Use the quoting mechanisms to override the meaning of special characters on the command line.

© 2003 Hewlett-Packard Development Company, L.P.

10–1. SLIDE: Introduction to Quoting

Introduction to Quoting

- Many characters have "special" meaning to the shell:
  - white space
  - carriage return
  - $
  - #
  - *
  - < >
- Quoting removes (escapes) the special meaning of the special characters.

Student Notes

There are many characters in the UNIX system that have special meaning for the shell. For example, white space is the delimiter between commands and arguments. The carriage return signals the shell to execute the entered line, the **$** character is used to display the value associated with a variable name.

There are situations in which you do not want the shell to interpret the special meaning associated with these characters. You require just the literal character. Therefore, the UNIX system must provide a mechanism to escape or remove the special meaning of a designated character. This mechanism is known as **quoting**.

© 2003 Hewlett-Packard Development Company, L.P.

10–2. SLIDE: Quoting Characters

Quoting Characters

Backslash	\
Single Quotes	'
Double Quotes	"

Student Notes

The *backslash* (\) removes the special meaning of the special character immediately following the backslash.

Single quotes (') will also disable the special meaning of special characters. *All* special characters enclosed by the single quotes are escaped. The single quote cannot be escaped because it is required to close the quoted string.

NOTE: Single quotes (') are not the same as the grave quote (grave accent) (`).

Double quotes (") are less comprehensive. *Most* special characters enclosed by double quotes are escaped. The exceptions are the **$** symbol (when used for variable and command substitution), the backslash (\) and the double quote (") which is required to close the quoted string. You can use the backslash inside double quotes to escape the meaning of **$** or ".

© 2003 Hewlett-Packard Development Company, L.P.

10–3. SLIDE: Quoting — \

Quoting — \

Syntax:

\ Removes the special meaning of the next character

Example:

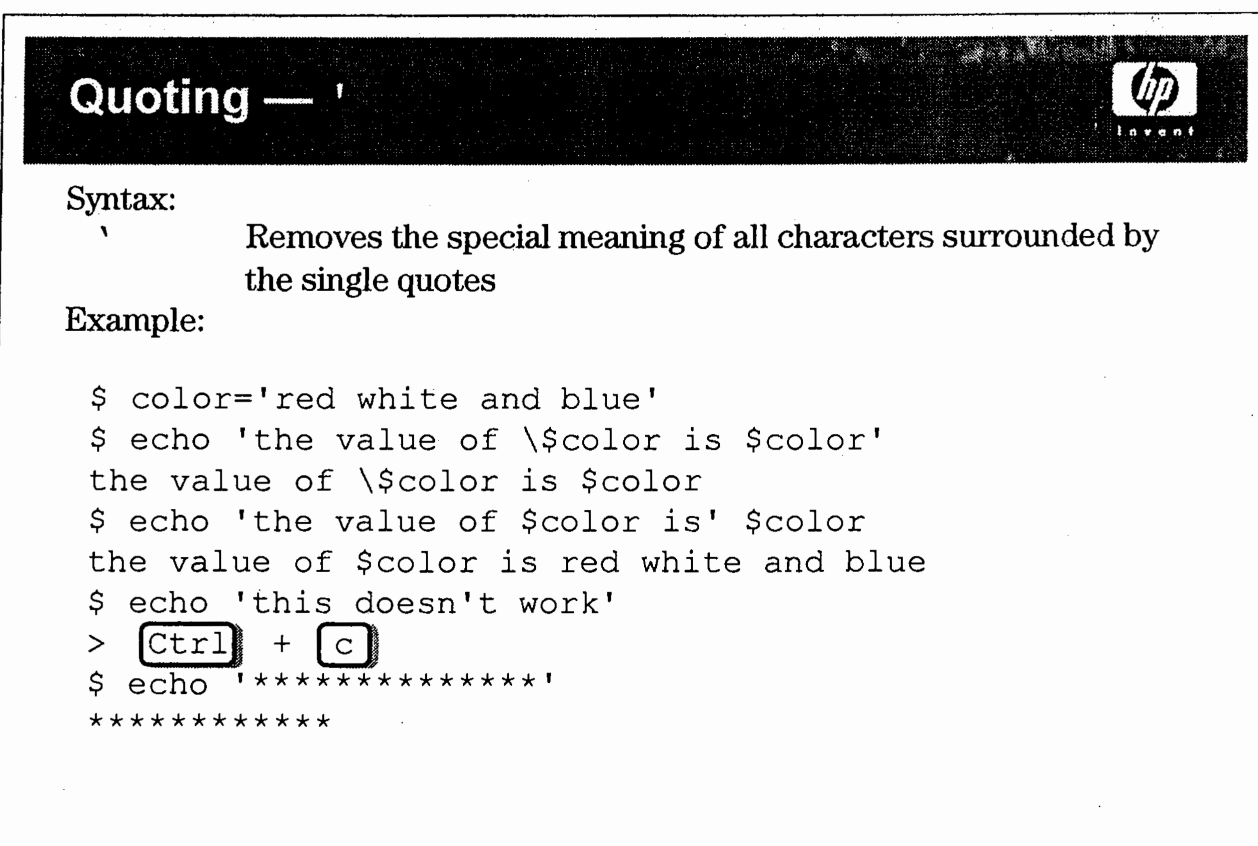
```
$ echo the \\ escapes the next character
the \ escapes the next character
$ color=red\ white\ and\ blue
$ echo the value of \$color is $color
the value of $color is red white and blue
$ echo one two \
> three four
one two three four
```

Student Notes

The backslash always removes the special meaning of the next character. There are no exceptions.

© 2003 Hewlett-Packard Development Company, L.P.

10–4. SLIDE: Quoting — '



Student Notes

Single quotes remove the special meaning of *all of* the special characters enclosed by the single quotes.

© 2003 Hewlett-Packard Development Company, L.P.

10–5. SLIDE: Quoting — "

Quoting — "

Syntax:

" Removes the special meaning of all characters surrounded by the double quotes except \, $, {*variable name*}, $(*command*), and "

Examples:

```
$ color="red white and blue"
$ echo "the value of \$color is $color"
the value of $color is red white and blue
$ cur_dir="$LOGNAME - your current directory is $(pwd)"
$ echo $cur_dir
user3 - your current directory is /home/user3/tree
$ echo "they're all here, \\, ', \" "
they're all here, \, ', "
```

Student Notes

Double quotes are not as comprehensive as the single quotes. *Most* of the special characters are escaped. The exceptions allow you to perform variable substitution, **$variable**, and command substitution, **$(cmd)** .

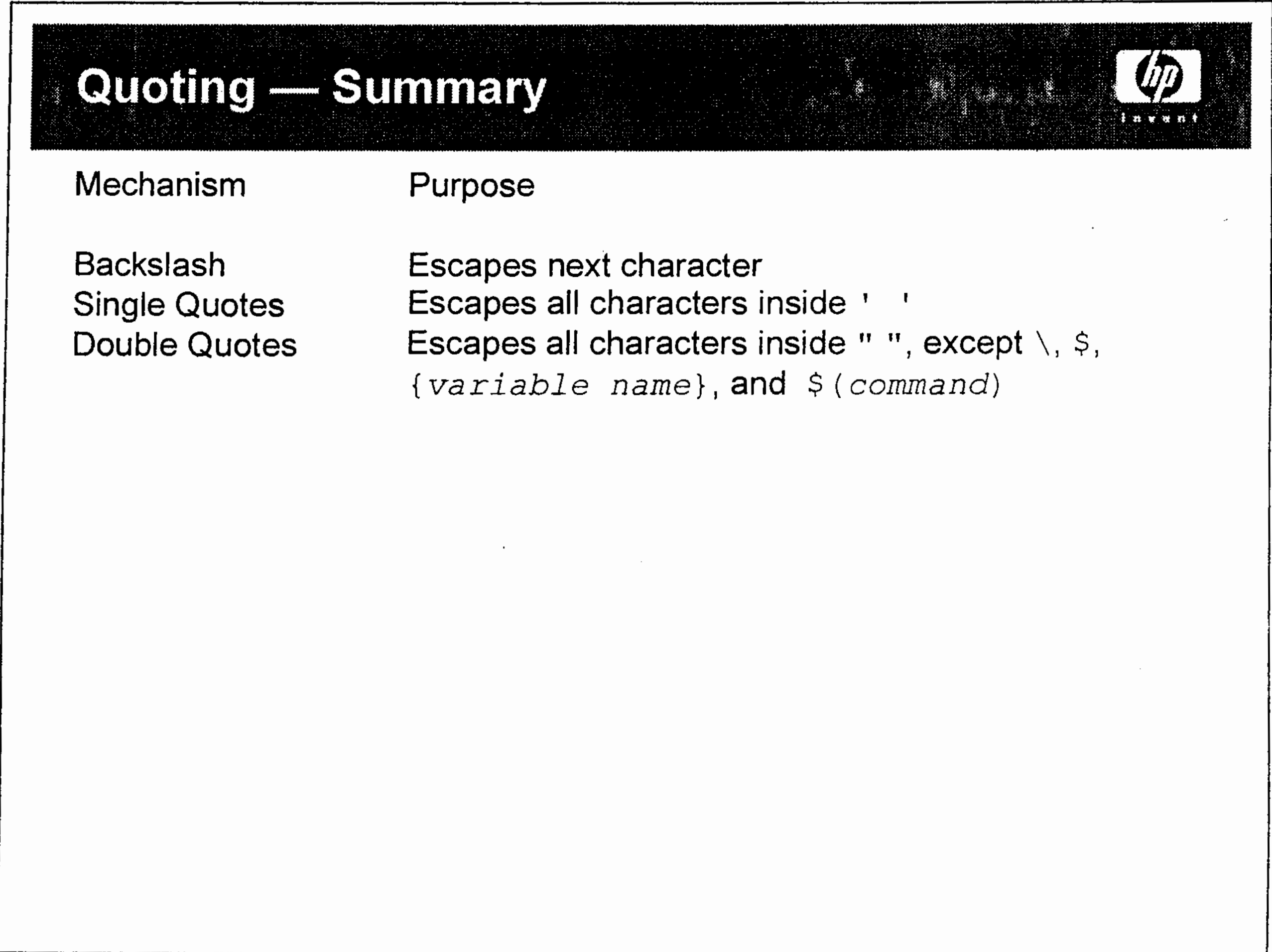
NOTE:	The Bourne shell uses grave quotes to perform command substitution, as in **pwd**, which produces the same result as the POSIX shell **$(pwd)** (the grave quote form is valid in the POSIX shell also). The grave quotes retain their special meaning inside the double quotes.

There may be situations where you want to escape the special meaning of these characters when they appear within the double quotes. Therefore, the backslash (\) also maintains its special meaning, to escape the special meaning of the **$** or ` when they do appear with the double quotes.

NOTE:	All quoting mechanisms can be used in a single command line.

© 2003 Hewlett-Packard Development Company, L.P.

10–6. SLIDE: Quoting Summary



Student Notes

The slide shows a summary of the quoting characters and their actions.

© 2003 Hewlett-Packard Development Company, L.P.

10–7. LAB: Quoting

Directions

Complete the following exercises and answer the associated questions.

1. Type an echo command that will produce the following output:

```
$1 million dollars    ...    and that's a bargain !
```

2. Assign the following string to a variable called *long_string*:

```
$1 million dollars    ...    and that's a bargain !
```

Display the value of *long_string* to confirm the successful assignment

3. When you execute the following command, what happens?

```
$ banner good day
$ banner 'good day'
```

How many arguments are on each of the above command lines?

NOTE:	Linux users do not, normally, have the **banner** command available to them. Also, the Linux version of the banner command is not intended for use with screen display output. If you attempt the commands, as shown above, in a Linux environment, you should see only error messages.

4. Assign to your prompt the string: *Way to go YOUR_USER_NAME $*

© 2003 Hewlett-Packard Development Company, L.P.

5. How would you display the following message?

```
Exercises #1, #2, and #3 are now complete.
```

6. Assuming that the variable *abc* is not defined, what happens when you enter the following?

```
echo '$abc'
```

What happens when you enter the following?

```
echo "$abc"
```

7. Use the **touch** command to create a file called: White Space
Use the **touch** command to create a file called: (4 blanks)
Use the **touch** command to create a file called: (3 blanks)
How do these files appear when you do a file listing? Can you do a file listing such that you can determine how many blanks are in the file name with 4 blanks or the file name with 3 blanks?

© 2003 Hewlett-Packard Development Company, L.P.

© 2003 Hewlett-Packard Development Company, L.P.

Module 11 — Input and Output Redirection

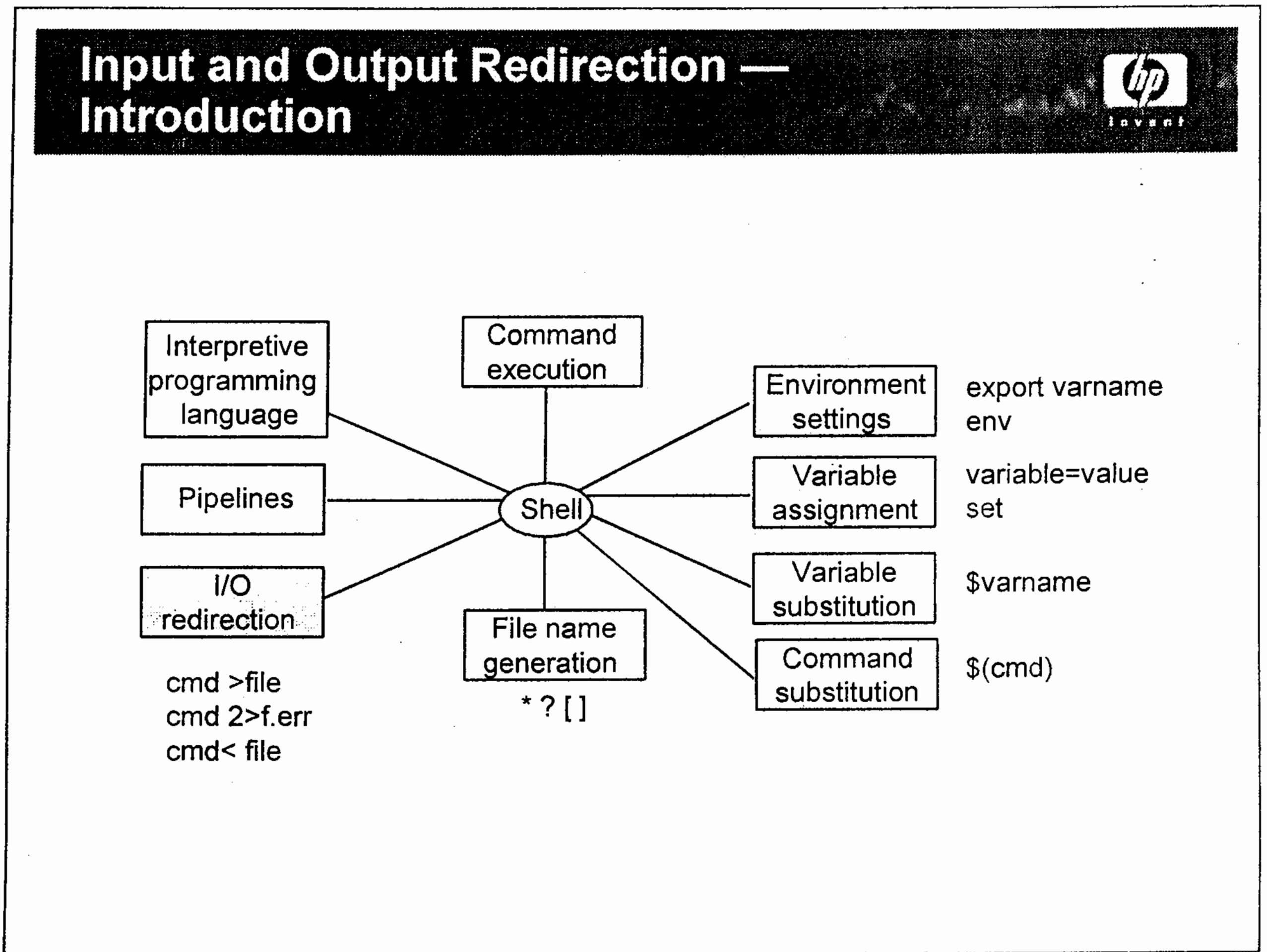
Objectives

Upon completion of this module, you will be able to do the following:

- Change the destination for the output of UNIX system commands.
- Change the destination for the error messages generated by UNIX system commands.
- Change the source of the input to UNIX system commands.
- Define a filter.
- Use some elementary filters such as `sort`, `grep`, and `wc`.

© 2003 Hewlett-Packard Development Company, L.P.

11–1. SLIDE: Input and Output Redirection — Introduction



Student Notes

Another feature that the shell provides is the capability to redirect the input or output of a command. Most commands send their output to your terminal; examples include **`date`**, **`banner`**, **`ls`**, **`who`**, etc. Other commands get input from your keyboard; examples include **`mail`**, **`write`**, **`cat`**.

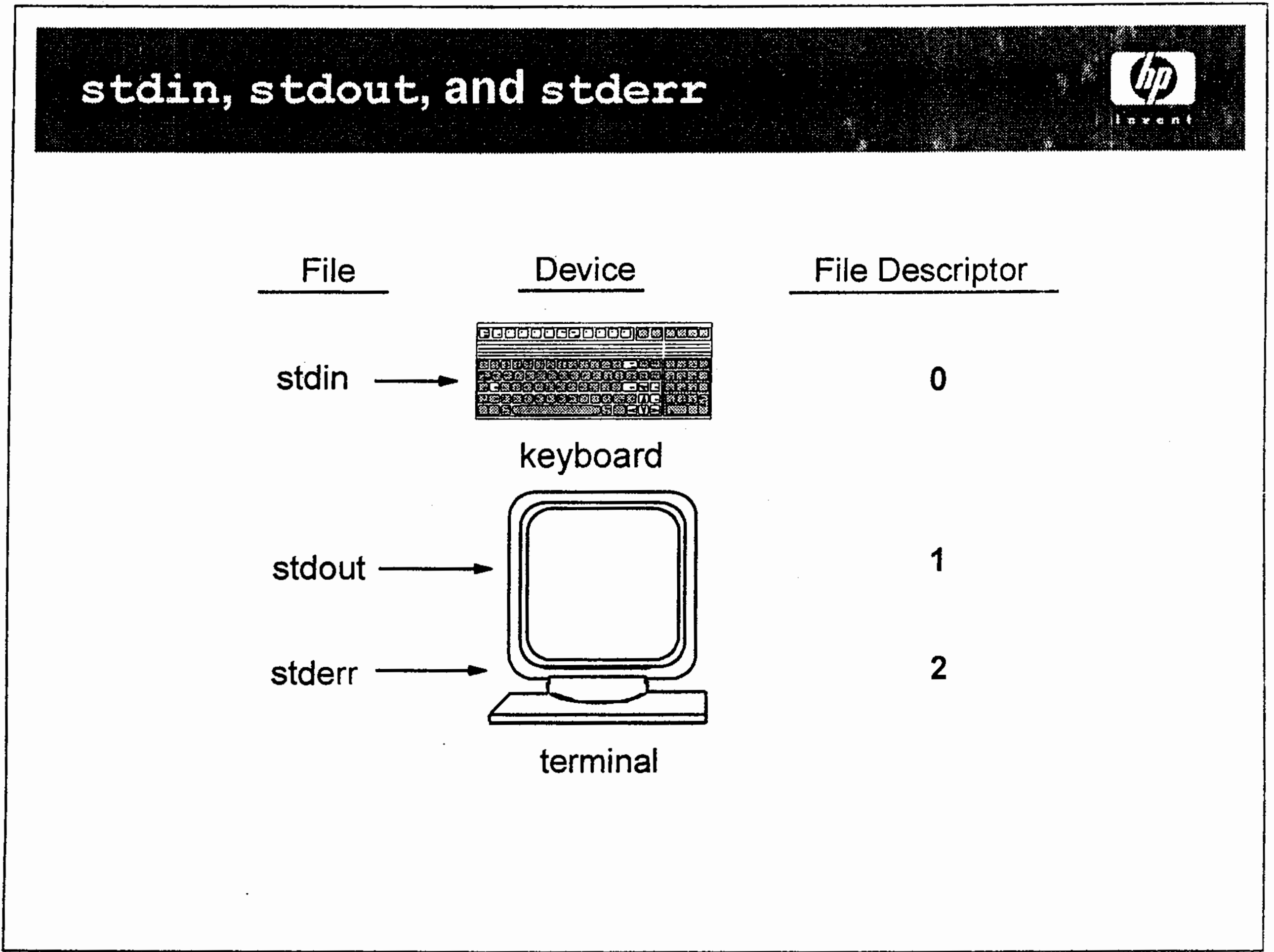
In the UNIX system *everything* is a file, including your terminal and keyboard. **Output redirection** allows you to send the output of a command to some file other than your terminal. Likewise, **input redirection** allows you to get the input for a command from some file other than the keyboard.

Output redirection is useful for capturing the output of a command for logging purposes or even for further processing. Input redirection allows you to use an editor to create a file, and then send that file into the command, instead of entering it interactively with no edit capabilities (for example the **`mail`** command).

This chapter will present input and output redirection, and introduce you to some UNIX system filters. Filters are special utilities that can be used to further process the contents of a file.

© 2003 Hewlett-Packard Development Company, L.P.

11–2. SLIDE: stdin, stdout, and stderr



Student Notes

Every time a shell is started, three files are automatically opened for your use. These files are called **stdin**, **stdout**, and **stderr**.

The **stdin** file is the file from which your shell reads its input. It is usually called **standard input**. This file is opened with the C language file descriptor, 0, and is usually attached to your keyboard. Therefore, when the shell needs input, it must be typed in at the keyboard.

Commands that get their input from standard input include **mail**, **write**, and **cat**. They are characterized by entering the command and arguments and a [Return], and then the command waits for you to provide input that it will process. The input is concluded by entering [Return] [Ctrl] + [d].

The **stdout** file is the file to which your shell writes its normal output. It is usually called **standard output**. This file is opened with the C language file descriptor, 1, and is usually attached to your terminal. Therefore, when the shell produces output, it is displayed to your screen.

© 2003 Hewlett-Packard Development Company, L.P.

Most UNIX system commands generate standard output. Examples include **date**, **banner**, **ls**, **cat** and **who** .

The **stderr** file is the file to which your shell writes its error messages. It is usually called **standard error.** This file is opened with the C language file descriptor, **2**. Like the **stdout** file, the **stderr** file is usually attached to the monitor part of your terminal. The **stderr** file can be redirected independently of the **stdout** file.

Most UNIX system commands will generate an error message when the command has been improperly invoked. To see an example of an error message enter: **cp** [Return]. The **cp** usage message will be displayed to your screen but actually was transmitted through the standard error stream.

The purpose of this module is to show you how to change the default assignments of **stdin**, **stdout**, and **stderr**, thus taking the input from a file other than the keyboard, and producing output (and error messages) somewhere other than the terminal.

© 2003 Hewlett-Packard Development Company, L.P.

11–3. SLIDE: Input Redirection — <

Input Redirection — <

Any command that reads its input from stdin can have its input redirected to come from another file.

Example:

```
$ cat remind
Your mother's birthday is November 29
$ mail user3 < remind
$ mail
From user3 Mon July 15 11:30 EDT 1993
Your mother's birthday is November 29
?d
$
```

Student Notes

For commands that take their input from standard input, we can redirect the input so that it comes from a file instead of from the keyboard. The **mail** command is often used with input redirection. We can use an editor to create a file containing some text that we want to mail, and then we can redirect the input of **mail** so that it uses the text in the file. This is useful if you have a very long mail message, or want to save the mail message for future reference.

Commands that receive input from standard input are characterized by entering the command and then the [Return], and the command will wait for the user to provide input from the keyboard. The input is concluded with [Return] [Ctrl] + [d].

Many commands that accept standard input also accept file names as arguments. The files specified as arguments will be processed by the command. The **cat** command is a good example. The **cat** command can display text that is entered directly from the keyboard, display the contents of files provided as arguments, or the contents of files redirected through standard input.

© 2003 Hewlett-Packard Development Company, L.P.

Input from stdin:	Operate on cmd line arg(s):	Redirect input:
`$ cat` [Return] *input text here* [Crtl] + [d] to conclude. *Contents of input text displayed here*	`$ cat` file *display file contents*	`$ cat` <file *display file contents*

NOTE: Input redirection causes *no* change to the contents of the input file.

© 2003 Hewlett-Packard Development Company, L.P.

11–4. SLIDE: Output Redirection — > and >>

Output Redirection — > and >>

Any command that produces output to stdout can have its output redirected to another file.

Examples:

Create/Overwrite

```
$ date > date.out
$ date > who.log
$ cat > cat.out
input text here
Ctrl + d
```

Create/Append

```
$ ls >> ls.out
$ who >> who.log
$ ls >> who.log
```

Student Notes

Many commands generate output messages to your screen. Output redirection allows you to capture the output and save it to a text file.

If a command line contains the output redirection symbol (>) followed by a file name, the standard output from the command will go to the specified file instead of to the terminal. If the file didn't exist before the command was invoked, then the file is automatically created. If the file *did* exist before the command was invoked, then the file will be *overwritten*; the command's output will completely replace the previous contents of the file.

If you want to append to a file instead of overwriting, you can use the output redirection append symbol (>>). This will also create the file if it didn't already exist. There must be *no* white space between the two > characters.

© 2003 Hewlett-Packard Development Company, L.P.

CAUTION:	The shell cannot open a file for input redirection and output redirection at the same time. So the only restriction is that the input file and the output file *must* be different. You will lose the original contents of the file, and the output redirection will also fail. Example: **`cat f1 f2 > f1`** will cause the contents of file `f1` to be lost.

© 2003 Hewlett-Packard Development Company, L.P.

11–5. SLIDE: Error Redirection — 2> and 2>>

Error Redirection — 2> and 2>>

Any command that produces error messages to stderr can have those messages redirected to another file.

Examples:

```
$ cp 2> cp.err      Create/Overwrite
$ cp 2>> cp.err     Create/Append
$

$ more cp.err
Usage: cp [-f|-i] [-p] source_file target_file
       cp [-f|-i] [-p] source_file ...target_directory
       cp [-f|-i] [-p] -R|-r
source_directory...target_directory
Usage: cp [-f|-i] [-p] source_file target_file
       cp [-f|-i] [-p] source_file ... target_directory
       cp [-f|-i] [-p] -R|-r source_directory...target_directory
```

Student Notes

If a command is typed incorrectly such that the shell cannot properly interpret it, an error message will often be generated. Even though the error messages are displayed on your screen, they actually are transmitted through a different file from the ordinary output messages. The error messages are transmitted through the error stream, known as **`stderr`**. **`stderr`** is associated with file descriptor 2.

Therefore, when specifying error output redirection, you must designate that you want to capture the messages being transferred out of stream 2. To redirect **`stderr`**, use (2>). There must be *no* white space between the 2 and the > characters. Similar to output redirection, this will create a file if necessary, or overwrite the file if it exists. You can append to an existing file using the (2>>) symbol.

This mechanism is very useful from an administrative viewpoint. Quite often, you are only interested in the situations when commands fail or experience problems. Since the error messages are separated from the regular output messages, you can easily capture the error messages, and maintain a log file which records the problems your program encountered.

© 2003 Hewlett-Packard Development Company, L.P.

11–6. SLIDE: What Is a Filter?

What Is a Filter?

- Reads standard input and produces standard output.
- Filters the contents of the input stream or a file.
- Sends results to screen, never modifies the input stream or file.
- Processes the output of other commands when they are used in conjunction with output redirection.

Examples: `cat`, `grep`, `sort`, `wc`

Student Notes

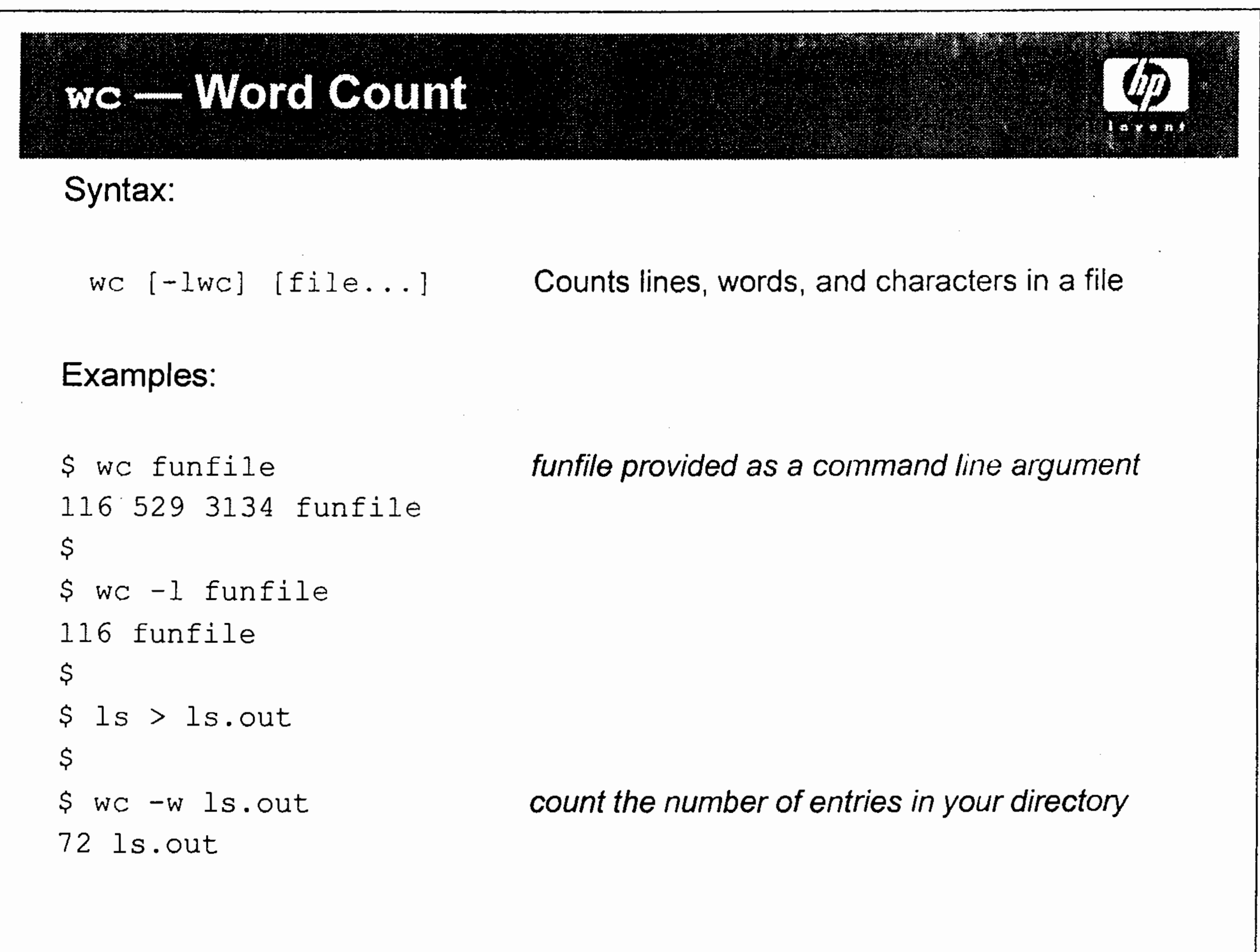
You have seen on the previous pages how to redirect the input or output of a command. Some commands accept input from standard input *and* generate output to standard output. These commands are known as **filters**. Filters never modify the contents of the file that is being processed. Filtered results are usually transmitted to the terminal.

Filters are very useful for processing the contents of a file, such as counting the number of lines (**wc**), performing an alphabetical sort (**sort**), or searching for lines that contain a pattern (**grep**).

In addition, filters can be used to further process the output of *any* command. Since filters can operate on files and the output of commands can be redirected to a file, the two operations can be combined to perform powerful and flexible processing of the output of any command. Since most filters send their results to standard output, the filtered results can be further processed by capturing the filtered output to a file and executing another filter on the filtered file.

© 2003 Hewlett-Packard Development Company, L.P.

11–7. SLIDE: wc — Word Count



Student Notes

The wc command counts the number of lines, words, and characters submitted on standard input or in a file. The command has options -l, -w, and -c. The -l option will display the number of lines, the -w option will display the number of words, and the -c option will display the number of characters. Regardless of the order of the options, the order of the output will always be lines, words, and characters.

Since wc accepts input from standard input and writes its output to standard output, wc is a *filter*. Executing wc on a file does not affect the contents of the file because all of the results are sent to the screen.

Other Examples

```
$ wc [Return]                    count input provided through standard input
ab cde
fghijkl
mno pqr stuvwxyz
[Ctrl] + [d]
3 6 32
$ wc < funfile                   standard input replaced by file funfile
```


© 2003 Hewlett-Packard Development Company, L.P.

```
105 718 3967                    no file name shown
$ wc -w funfile

718 funfile
```

wc will accept input from standard input as illustrated in the first example above. Since the wc command accepts input from standard input, you can redirect a file into the wc command that replaces the standard input stream. The syntax of the wc command also supports file names as arguments, as shown on the slide, with the name of the file written out on the result.

© 2003 Hewlett-Packard Development Company, L.P.

11–8. SLIDE: `sort` — Alphabetical or Numerical Sort

sort — Alphabetical or Numerical Sort

Syntax:

```
sort [-ndutX] [-k field_no] [file...] Sorts lines
```

Examples:

```
$ sort funfile              funfile provided as a command line argument

$ tail -1 /etc/passwd
user3:xyzbkd:303:30:studentuser3:/home/user3:/usr/bin/sh
 1      2      3  4      5              6            7
$ sort -nt: -k 3 < /etc/passwd

$ who > whoson
$ sort whoson               sort logged in users alphabetically
$ sort -u -k 1,1 whoson     sort and suppress duplicate lines
```

Student Notes

The **sort** command is powerful and flexible. It can be used to sort the lines of a file(s) in numerical or alphabetical order. A specific field on a line can also be selected upon which to base the sort. **sort** is also a filter, so it will accept input from standard input, but it will also sort the contents of files that are specified as command line arguments.

There are several options available to designate what kind of sort to be performed:

Sort Option	Sort Type
none	lexicographical (ASCII)
-d	dictionary (disregards all characters that are not letters, numbers, or blanks)
-n	numerical
-u	unique (suppress all duplicate lines)

The default delimiter between fields is a blank character — either a space or a tab. You can also specify a delimiter with the **-t** *X* option, where *X* represents the delimiter character.

© 2003 Hewlett-Packard Development Company, L.P.

Since the colon (`:`) holds no special meaning to the shell, it is a common selection as a delimiter between fields in a file.

After you have determined what the delimiter between fields will be, you can inform the **`sort`** command which field you would like to base your sort on by using the **`-k`** ***n*** option, where *n* represents the field number the sort should sort upon. The **`sort`** command assumes that the field numbering starts with one.

The **`sort`** command supports several options to perform more complex sort operations. Please refer to **`sort(1)`** in the *HP-UX Reference Manual* for a full discussion of its capabilities.

Other Examples

```
$ sort [Return]                    sort input provided through standard input

mmmmm
xxxx
aaaa
[Ctrl] + [d]
aaaa
mmmmm
xxxx
$ sort < funfile                   standard input replaced by file funfile
```

`sort` will accept input from standard input, as illustrated in the first example above. Therefore, you can also get the input from a file using input redirection.

NOTE: The shell cannot open a file for input redirection and output redirection at the same time. However the **`sort`** option **`-o output_file`** can be used to produce the output inside the argument given instead of the standard output. Then this file may be the same name as the input file.

Example: **`sort -o whoson -d whoson`** will perform a dictionary sort inside the file **`whoson`**.

© 2003 Hewlett-Packard Development Company, L.P.

11–9. SLIDE: grep — Pattern Matching

grep — Pattern Matching

Syntax:

```
grep [-cinv] [-e] pattern [-e pattern] [file...]
grep [-cinv] -f  patterns_list_file [file...]
```

Examples:

```
$ grep user /etc/passwd
$ grep -v user /etc/passwd
$ grep -in -e like -e love funfile

$ who > whoson
$ grep rob whoson
```

Student Notes

The **grep** command is very useful. It takes a (usually quoted) pattern as its first argument, and it takes any number of file names as its remaining arguments. It is possible to make the **grep** command searching for several patterns once by using the **-e** option before each pattern or the **-f** option followed by a patterns list file. It searches the named files for lines that contain the specified pattern. The **grep** command then displays the lines that contain the pattern.

There are four popular options to **grep**: **-n**, **-v**, **-i** and **-c**.

-c	only a count of matching lines is printed
-i	tells grep to ignore the case of the letters in the pattern
-n	prepends line numbers to each line displayed
-v	displays the lines that *do not* contain the pattern

© 2003 Hewlett-Packard Development Company, L.P.

As with all filters, if no file is specified, `grep` reads from standard input and sends its output to standard output.

The `grep` command is capable of more complex searches. You can give a pattern of the text you want to search for. Such a pattern is called a *regular expression*. Here is a list of some special characters for the regular expressions (for further details see `regexp(5)`).

`^`	match beginning of the line
`$`	match end of the line
`.`	match any single character
`*`	the preceding pattern is to be repeated zero or more times
`[ ]`	character class, specify a set of characters
`[ - ]`	the hyphen characters (-) specifies a range of characters
`[^ ]`	inverts the selection process

To avoid problems with the interpretation of the special characters through the shell, it is best to enclose the regular expression in quotes.

The GNU/Linux version of the `grep` program has a number of additional options. In addition, certain options must be specified in a different format than that used in the HP-UX version. Please refer to the appropriate man pages for a listing of available options.

© 2003 Hewlett-Packard Development Company, L.P.

11–10. SLIDE: Input and Output Redirection — Summary

Input and Output Redirection — Summary

`cmd < file`	Redirects input to `cmd` from *`file`*
`cmd > file`	Redirects standard output from `cmd` to *file*
`cmd >> file`	Redirects standard output from `cmd` and append to *`file`*
`cmd 2> file.err`	Redirects errors from `cmd` to *`file.err`*
`A filter`	A command that accepts stdin and generates stdout
`wc`	Line, word, and character count
`Sort`	Sorts lines alphabetically or numerically
`grep`	Searches for lines that contain a pattern

Student Notes

© 2003 Hewlett-Packard Development Company, L.P.

11–11. LAB: Input and Output Redirection

Directions

Complete the following exercises and answer the associated questions.

1. Redirect the output of the `date` command to a file called `date.out` in your `HOME` directory.

2. Append the output of the `ls` command to the file `date.out`. Look at the contents of `date.out`. What do you notice?

3. Using input redirection, mail the file `date.out` to your mail partner.

4. Create two very short files called `f1` and `f2` using `cat` and output redirection.

5. Use the `cat` command to view their contents. Use the `cat` command to create a new file called `f.join` that contains the contents of both `f1` and `f2`. Do you see any output on the screen?

© 2003 Hewlett-Packard Development Company, L.P.

6. Use the `cat` command to display the contents of the file `f1`, `f2` and `f.new`. NOTE: `f.new` should NOT exist. What do you see on your screen? Is it obvious which messages went through standard output and which messages went through standard error?

7. Again, use the `cat` command to display the contents of the file `f1`, `f2` and `f.new`. NOTE: `f.new` should NOT exist. This time capture any error messages that are generated and send them to the file called `f.error`. What do you see on your screen? Was a new file created? Check its contents.

8. Again, use the `cat` command to capture the contents of the file `f1`, `f2` and `f.new`.

   NOTE: `f.new` should *not* exist. This time, ON ONE COMMAND LINE, capture the standard output messages to a file called `f.good` AND the error messages to a file called `f.bad`. What do you see on your screen? Were any new files created? Check their contents.

9. Type the `cp` command with no arguments. What happens? Now try redirecting the output from this command to the file `cp.error`. What happens? What must you do to redirect that error message to a file? Does the `cp` command generate any standard output messages?

10. Display the contents of the file `/etc/passwd` sorted out by user name.

© 2003 Hewlett-Packard Development Company, L.P.

11. Sort the file **/etc/passwd** on the third field. What happens? Now do a numeric sort on the third field. Any difference?

12. Display all of the lines in the file **/etc/passwd** that contain the string user. Save this output to a file called **grepped**. Use a filter to determine how many lines in **/etc/passwd** contain the string **user**.

13. Using redirection and filters, how many users are logged in on the system?

14. How many login accounts are set up on the system? What command did you use to find out? (HINT: There is one line per account in the file **/etc/passwd**.)

15. Sort your names file and save the output in a file called **names.sort**. Sort the names file in reverse order and save that output to **names.rev**. What commands did you use? Check the manual entry for the **sort** command and find the option that allows you to save the sorted output back to the file **names**.

© 2003 Hewlett-Packard Development Company, L.P.

16. Send a **banner** message to your mail partner's terminal. Hint: What device file is associated with your mail partner's terminal? What does it mean if you get a "*Permission denied*" message?

Linux Systems

Because the **banner** command will, probably, not be available for use, try using the **echo** command instead.

© 2003 Hewlett-Packard Development Company, L.P.

© 2003 Hewlett-Packard Development Company, L.P.

Module 12 — Pipes

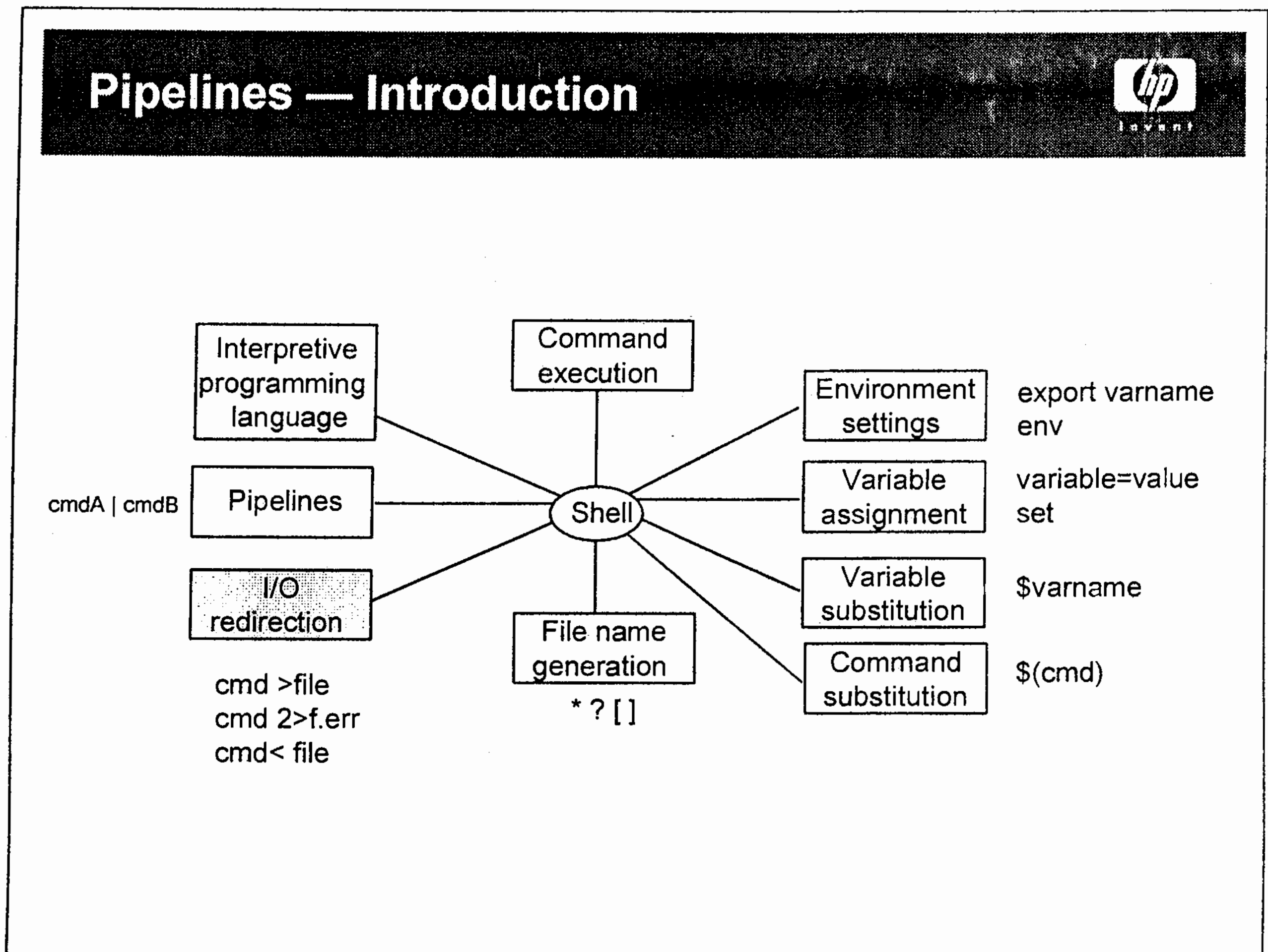
Objectives

Upon completion of this module, you will be able to do the following:

- Describe the use of pipes.
- Construct a pipeline to take the output from one command and make it the input for another.
- Use the `tee`, `cut`, `tr`, `more`, and `pr` filters.

© 2003 Hewlett-Packard Development Company, L.P.

12–1. SLIDE: Pipelines — Introduction



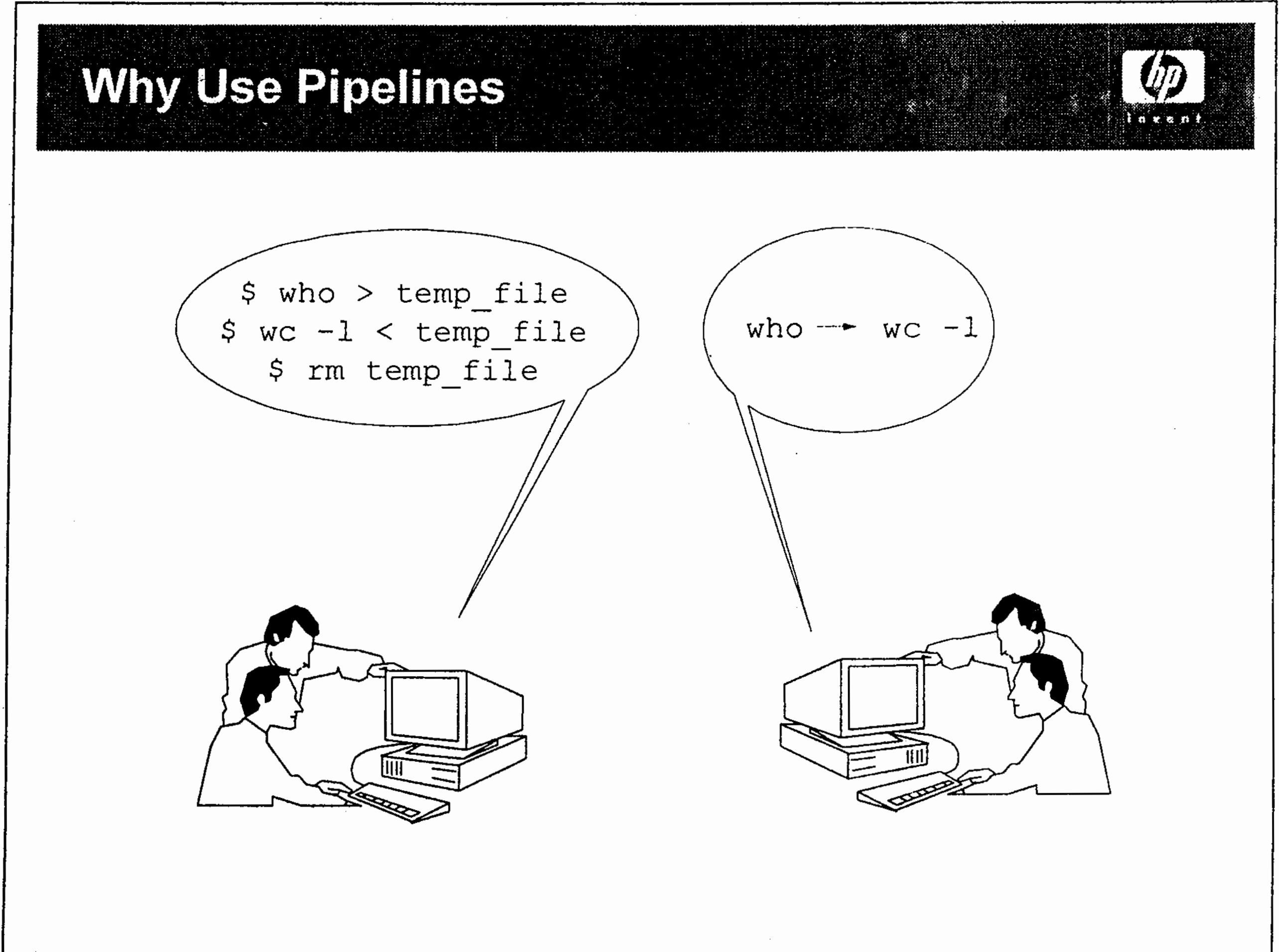
Student Notes

A useful feature that the shell provides is the capability to link commands together through pipelines. The UNIX system operating environment demonstrates its flexibility with the capability of filtering the contents of files. With pipelines, you will be able to filter the output of a command.

This chapter will introduce pipelines and then present some filters (`cut`, `tr`, `tee`, and `pr`) for further processing of your files or command output.

© 2003 Hewlett-Packard Development Company, L.P.

12–2. SLIDE: Why Use Pipelines?



Student Notes

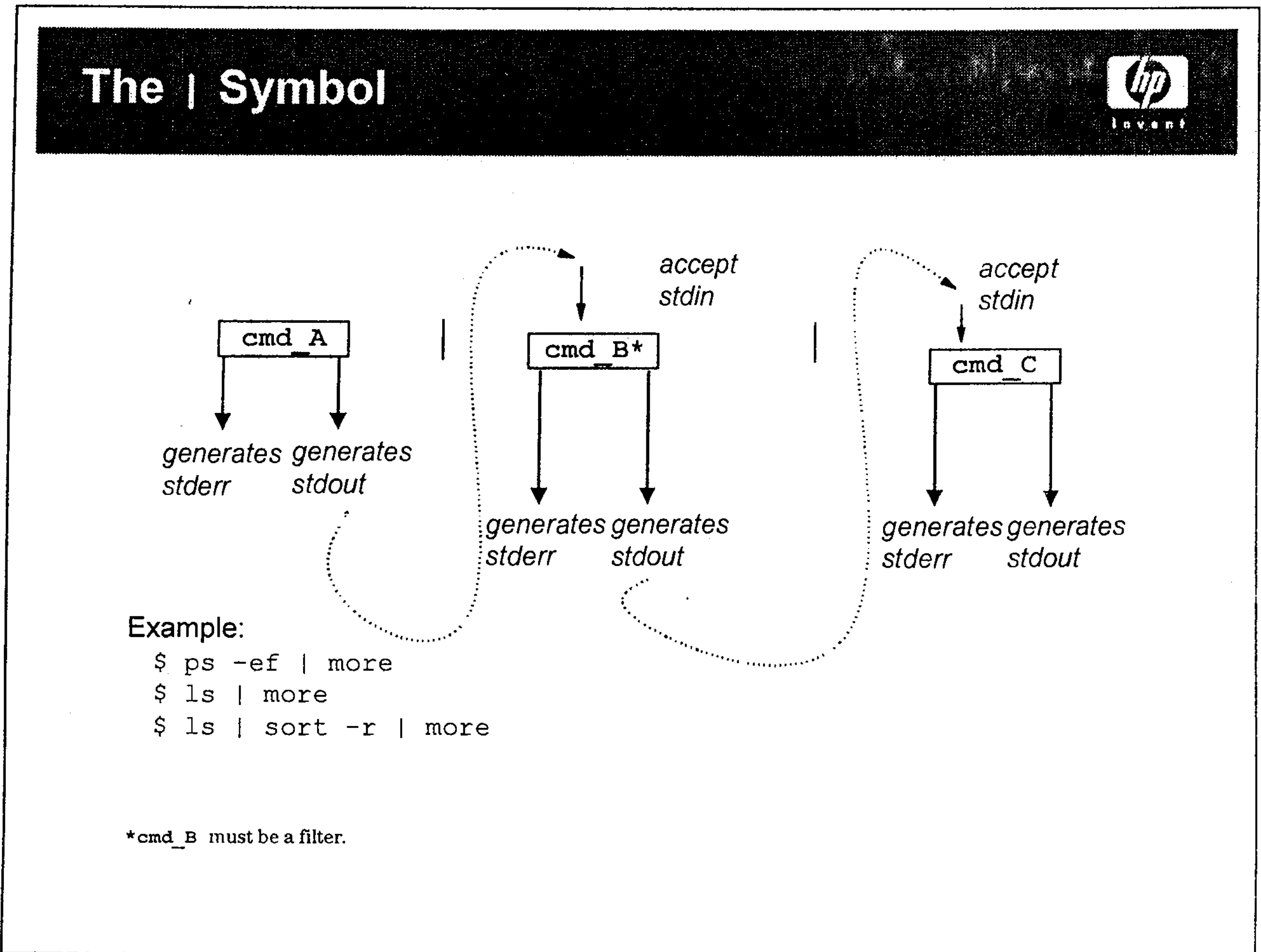
You use I/O redirection for extensive filtering of command output by capturing the output of a command to a temporary file and then filtering the contents of the temporary file. After your processing is complete, you have to remove the temporary file; it is not necessary for any other operations. Although this provides extensive capability, it is inconvenient to have to remove the temporary files.

Pipelines allow you to transfer the output of one command directly as the input of another command. You do not have to create an intermediate file; therefore, no cleanup is required when you have completed the processing.

This is where the flexibility and power of the UNIX system are demonstrated. Command after command can be chained together, allowing extensive processing capabilities in the context of a single command line.

© 2003 Hewlett-Packard Development Company, L.P.

12–3. SLIDE: The | Symbol



Student Notes

The | symbol (read as the **pipe** symbol) is used for linking two commands together. The standard output (**stdout**) of the command to the left of the | symbol will be used as the standard input (**stdin**) for the command to the right. A command that appears in the middle of a pipeline, therefore, must be able to accept standard input and produce output to standard output.

Filters such as **wc**, **sort**, and **grep** accept standard input and generate standard output, so they can appear in the middle of a pipe. By chaining commands and filters together, you can perform very complex processes.

The following summarizes the requirements for commands in each position in the pipeline:

- Any command to the left of a | symbol must produce output to stdout.
- Any command to the right of a | symbol must read its input from stdin.
- Any command between two | symbols must accept standard input and produce output to standard output. (It must be a filter.)

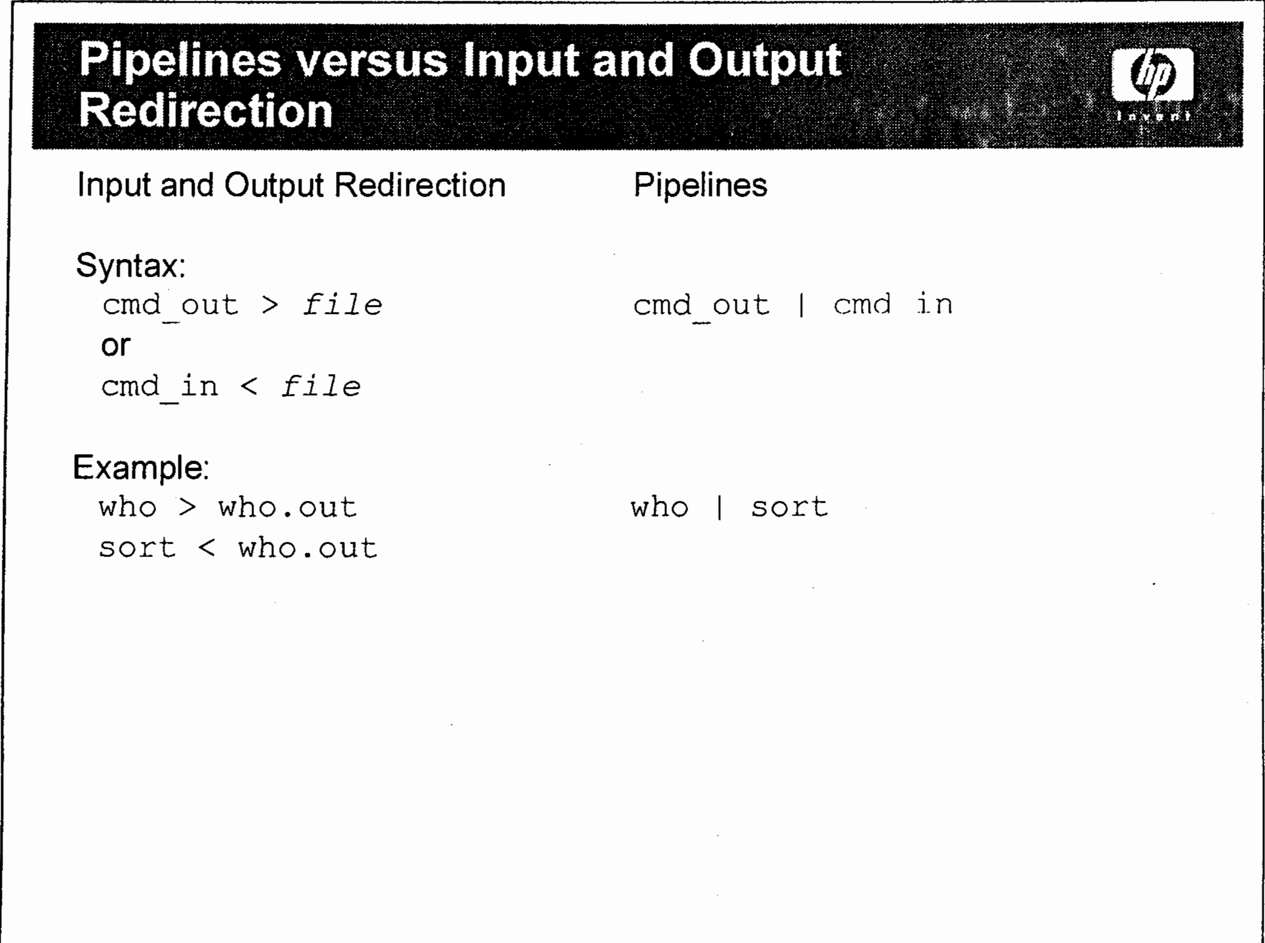
© 2003 Hewlett-Packard Development Company, L.P.

The `more` Command

The **`more`** command is used to display the contents of a file one screen at a time. The **`more`** command is capable of reading standard input as well. Therefore it can appear on the right of a pipe and be used to control the output of *any* command that generates output to standard output. This is very useful when a command generates extensively long output to your screen that you would like to view one screen at a time.

© 2003 Hewlett-Packard Development Company, L.P.

12–4. SLIDE: Pipelines versus Input and Output Redirection



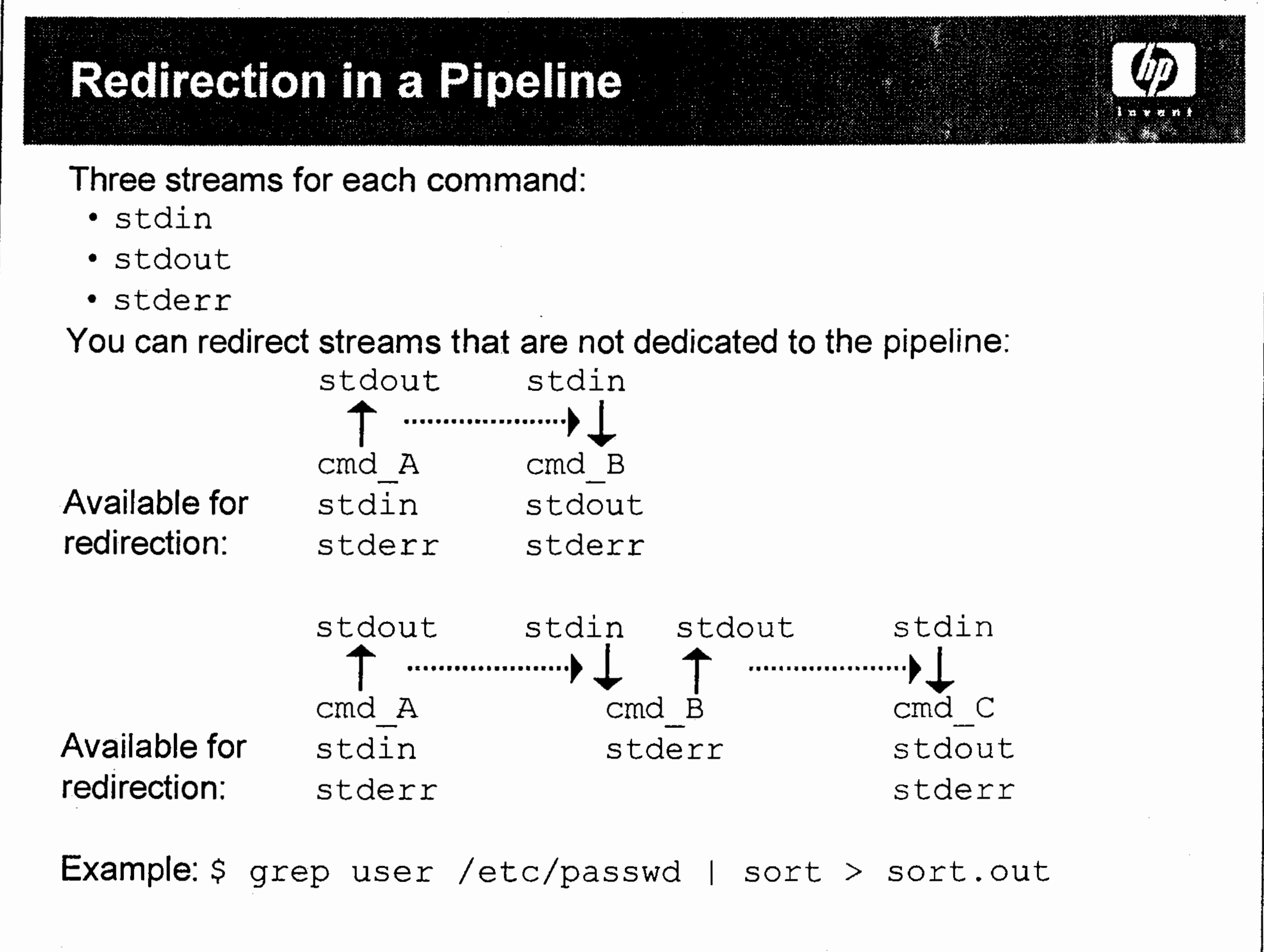
Student Notes

Input and output redirection will always be between a command and a file. Output redirection will capture the standard output of a command and send it to a file. Output redirection is commonly used for logging purposes or long-term storage of the output of a command. Input redirection redirects the input to come from a file instead of from the keyboard. Input redirection is rarely executed explicitly because most commands that accept standard input also accept file names as command line arguments (exceptions include **`mail`** and **`write`**). But the capability for input redirection is a requirement for a command that can appear on the right side of a pipe symbol.

Pipelines always will be used to join together two commands. If you intend the output of a command to be further processed by a command that accepts standard input, you should build a pipeline. Input and output redirection is used to direct between a process and a file. Pipelines are used to direct between processes.

© 2003 Hewlett-Packard Development Company, L.P.

12–5. SLIDE: Redirection in a Pipeline



Student Notes

Every command has three available streams: standard in (**stdin**), standard out (**stdout**), and standard error (**stderr**). Each command in a pipeline will reserve certain streams. The streams that are not dedicated to the pipeline can be redirected.

Following is a summary of the redirection available in the different components of a pipeline:

- Any command on the left of a pipe symbol can redirect input and errors because its output is passed on to the next command in the pipeline.
- Any command on the right of a pipe symbol can redirect output and errors because its input is coming from the previous command in the pipeline.
- Any command between two pipe symbols can redirect errors, because its input is coming from the previous command and its output is going to the next command in the pipeline.

© 2003 Hewlett-Packard Development Company, L.P.

Examples

The most common implementation is to redirect the output of the end of the pipeline to save the filtered output of the pipeline. When you redirect the output at the end of a pipeline, will you see any output go to the screen?

```
$ grep user /etc/passwd | sort > sorted.users
$ grep user < /etc/passwd 2> grep.err | sort > sorted.users 2> sort.err
$ grep user < /etc/passwd | sort 2> sort.err | wc -l > wc.out 2> wc.err
```

The output in the examples above will be sent to a file; no standard command output will be seen on the screen.

© 2003 Hewlett-Packard Development Company, L.P.

12–6. SLIDE: Some Filters

Some Filters

`cut`	Cuts out specified columns or fields and display to stdout
`tr`	Translates characters
`tee`	Passes output to a file and to stdout
`pr`	Prints and format output to stdout

Student Notes

Filters like **`sort`** or **`grep`** provide a flexible mechanism to perform processing on the output of many commands. The remainder of this chapter will provide you with pipeline practice by implementing three new filters. As with all filters, these commands accept standard input, so they can appear on the right side of a pipeline, and they generate standard output, so they can also appear on the left side of a pipeline (or in the middle of a pipeline).

The **`cut`** command allows you to cut out columns or fields of text from standard input or a file, and send the result to **`stdout`**.

The **`tee`** command allows you to send the output of a command to a file *and* to **`stdout`**.

The **`pr`** command is used to format output. It is usually invoked to prepare to send a file to the printer.

As with all filters, these commands will not modify the original file. The processed results will be sent to standard output.

© 2003 Hewlett-Packard Development Company, L.P.

12–7. SLIDE: The `cut` Command

The `cut` Command

Syntax:

```
cut  -clist [file...]                   Cuts columns or fields
cut  -flist [-dchar][-s][file...]      from files or stdin
```

Examples:

```
$ date | cut -c1-3
$ tail -1 /etc/passwd
user3:mdhbmkdj:303:30:student user3:/home/user3:/usr/bin/sh
  1       2       3  4         5          6            7
$ cut -f1,6 -d: /etc/passwd
$ cut -f1,6 -d: /etc/passwd | sort -r
$ ps -ef | cut -c49- | sort -d
```

Student Notes

The **`cut`** command is used to extract certain columns or fields from standard input or a file. The specified columns or fields will be sent to standard output. The **`-c`** option is for cutting columns, and the **`-f`** is for cutting fields. The **`cut`** command can accept its input from standard input or from a file. Since it accepts standard input, it can appear on the right side of a pipe.

A *list* is a number sequence used to tell **`cut`** which fields or columns are desired. The field specification is similar to the **`sort`** command. There are several permissible formats specifying the list of fields or columns:

`A-B` Fields or columns *A* through *B* inclusive

`A-` Field or column *A* through the end of the line

`-B` Beginning of line through field or column *B*

`A,B` Fields or columns *A* and *B*

© 2003 Hewlett-Packard Development Company, L.P.

Any combination of the above is also permissible. For example:

```
cut -f1,3,5-7 /etc/passwd
```

would cut fields one, three, and five through seven from each line of **`/etc/passwd`.**

The default delimiter between fields is specified as the [Tab] character. If you require some other delimiter, you can use the **`-d`** ***`char`*** option where ***`char`*** is the character that separates the fields in your input. (This is similar to the **`sort`** command's **`-t`** ***`X`*** option.) The colon is a common delimiter, as it has no special meaning for the shell.

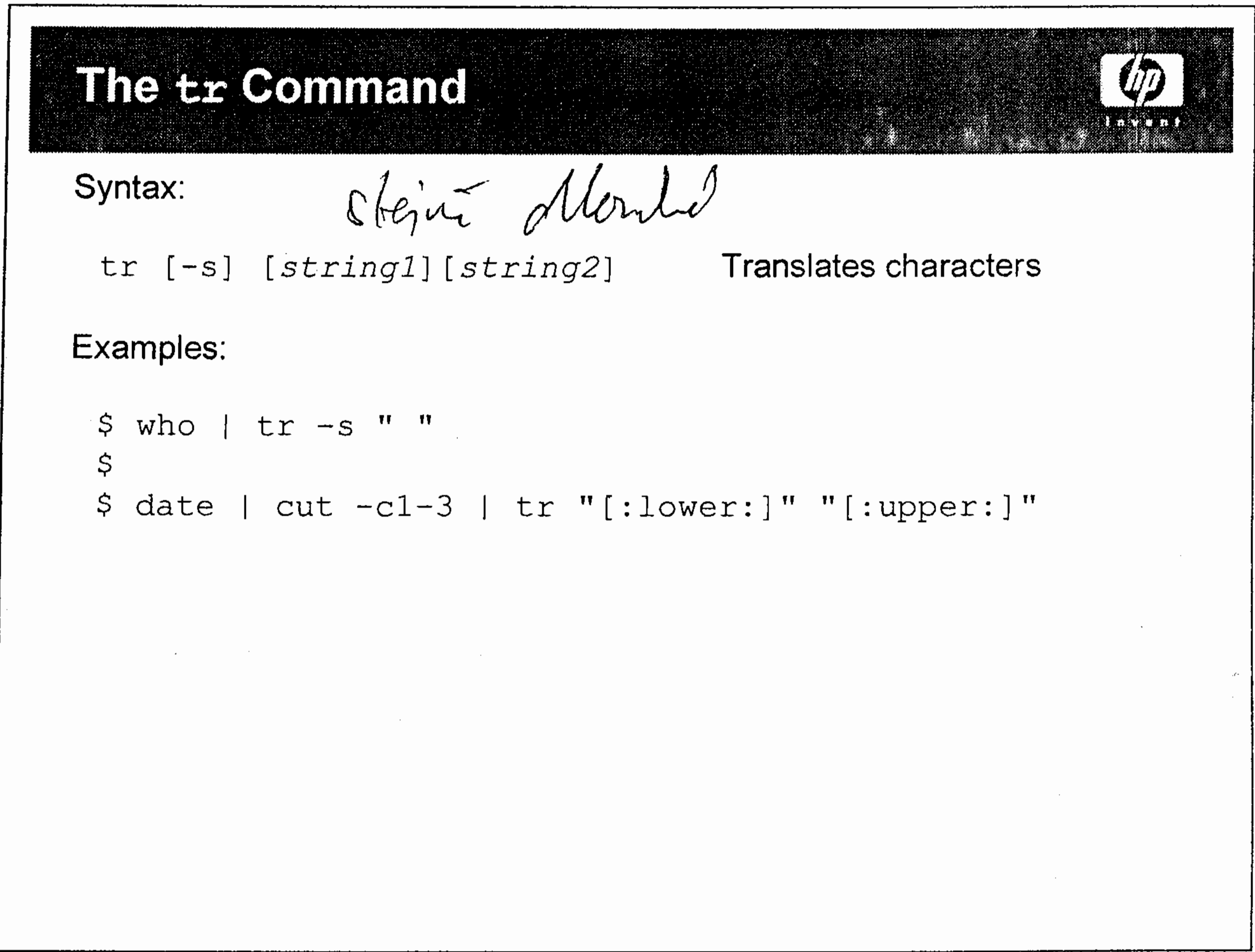
Also, the **`-s`** option, when cutting fields, will discard any lines that do not have the delimiter. Usually, these lines are passed through with no changes.

Example

```
$ cut -c1-3 [Return]
12345
123
abcdefgh
abc
[Ctrl] + [d]

$ date | cut -c1-3
```


© 2003 Hewlett-Packard Development Company, L.P.

12–8. SLIDE: The tr Command



Student Notes

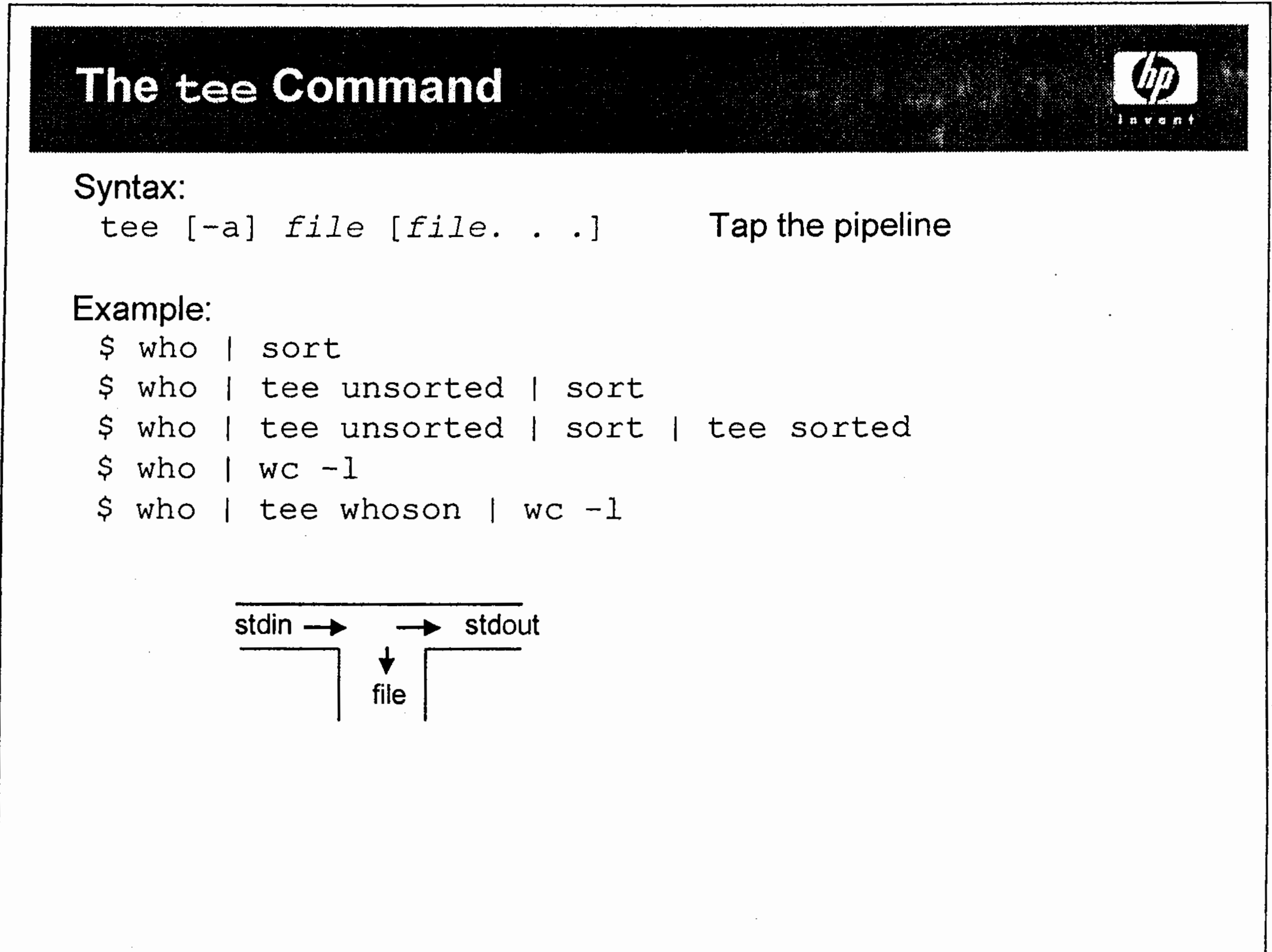
The `tr` command is useful to translate characters. It accepts standard input as well as file names; therefore, it can be used in a pipeline.

The `tr` command can be used to convert many consecutive blank spaces to a single blank space, as in the first example on the slide. You may have noticed that many UNIX system commands will insert a variable number of spaces between their fields. Therefore, `tr` can be a convenient predecessor to the `cut` command in a pipeline, when you would like to use a *single space* as the delimiter between fields.

The `tr` command also can be used to substitute literal strings or convert text from lowercase to uppercase and vice versa, as illustrated in the second example on the slide.

© 2003 Hewlett-Packard Development Company, L.P.

12–9. SLIDE: The tee Command



Student Notes

Generally, when you are executing a complex pipeline, the output of the intermediate commands is submitted to the next command in the pipe and you will not be able to view the intermediate output. The **tee** command is used to tap a pipeline. **tee** reads from standard input and writes its output to standard output *and* to the specified file. If the **-a** option is used, then **tee** appends its output to the file instead of overwriting it.

The **tee** command is used predominantly under two circumstances:

- To collect intermediate output in a pipeline:
  When you put a **tee** into the middle of a pipeline, you can capture the intermediate processing, yet pass the output to the next command in the pipeline.

- To send final output of a command to the screen and to a file:
  This is a useful logging mechanism. You may want to run a command interactively and see its output, but also save that output to a file. Remember when you just redirect the output of a command to a file, no output is sent to the screen. So this implementation can be used at the end of a pipeline, or at the end of any command that generates output.

© 2003 Hewlett-Packard Development Company, L.P.

12–10. SLIDE: The pr Command

The pr Command

Syntax:

```
pr [-option] [file...]    Formats stdin and produces stdout
```

Examples:

```
$ pr -n3 funfile
$ pr -n3 funfile | more
$ ls | pr -3
$ grep home /etc/passwd | pr -h "User Accounts"
```

Student Notes

The **pr** command stands for *print to stdout*. It is used to format the standard input stream or the contents of specified files. It sends its output to the screen, not to the printer. The **pr** command is typically executed, though, to format files in preparation for sending them to the printer.

The **pr** command is useful for printing long files because it will insert a header on the top of each new page that includes the file name (or header specified with the -h option), and a page number.

The **pr** command supports many options. The following is a summary of some of the more common ones:

-k	Produces *k*-column output; prints down the column
-a	Produces multicolumn output; used with **-k**; prints across
-t	Removes the trailer and header

© 2003 Hewlett-Packard Development Company, L.P.

`-d`	Double spaces the output
`-wN`	Sets the width of a line to *N* characters
`-lN`	Sets the length of a page to *N* lines
`-nCK`	Produces *K*-digit line numbering, separated from the line by the character *C*; *C* defaults to Tab
`-oN`	Offset the output *N* columns from the left margin
`-p`	Pauses and waits for Return before each page
`-h`	Uses the following *string* as the header text

© 2003 Hewlett-Packard Development Company, L.P.

12–11. SLIDE: Printing from a Pipeline

Printing from a Pipeline

`... | lp` Located at end of pipe; sends output to printer

Examples:

```
$ pr -158 funfile | lp
Request id is laser-226 (standard input).
$
$ ls -F $HOME | pr -3 | tee homedir | lp
Request id is laser-227 (standard input).
$
$ grep home /etc/passwd | pr -h "user accounts" | lp
Request id is laser-228 (standard input).
```

Student Notes

The **lp** command is used to queue a job for the printer. You submit a job by specifying a file name as an argument to **lp**. The **lp** command also accepts standard input, so you can pipe to the **lp** command as well. This allows the output of any command that generates standard output to be printed.

Generally, the **pr** command is used to format the output of a command prior to submitting it to the **lp** command for printing.

Because most pipelines will send their filtered output to **stdout**, it is easy to submit the output of most filter operations to the printer. If you need to save the output of the pipeline and send it to the printer, insert a **tee** prior to the **lp** command in the pipeline.

NOTE: In Linux, the **lpr** command would be used instead of the **lp** command.

© 2003 Hewlett-Packard Development Company, L.P.

12–12. SLIDE: Pipelines — Summary

Student Notes

© 2003 Hewlett-Packard Development Company, L.P.

12–13. LAB: Pipelines

Directions

Complete the following exercises and answer the associated questions.

1. Construct a pipeline that will count the number of users presently logged on.

2. Construct a pipeline that counts the number of lines in `/etc/passwd` that contain the pattern **home**. Now count the lines that do not contain the pattern.

3. Modify your pipeline from the above exercise so that you save all of the entries from `/etc/passwd` that contain the pattern home to a file called `all.users` before passing the output to be counted.

4. Construct a pipeline that will sort the contents of the **names** file found under your *HOME* directory, and display the sorted output in three-column format with no header or trailer.

5. Create an alias, called **whoson,** which will display an alphabetical listing of the users currently logged into your system.

© 2003 Hewlett-Packard Development Company, L.P.

6. Construct a pipeline to obtain a listing of just the user names of those users presently logged into the system.

7. Construct a pipeline to obtain a long listing of just file permissions and file names currently in your working directory.

8. Construct a pipeline that lists only the user name, size, and file name of each file in your `HOME` directory into a file called `listing.out`. At the same time, display on your screen only the total number of files.

9. Create a pipeline that will only capture the user name, user number, and `HOME` directory of every user account on your system. First, output the list in alphabetical order by user name. Now modify the pipeline so it sorts the list of user accounts by UID number instead of user name. Hint: the information can be found in `/etc/passwd`.

© 2003 Hewlett-Packard Development Company, L.P.

© 2003 Hewlett-Packard Development Company, L.P.

Module 13 — Using Network Services

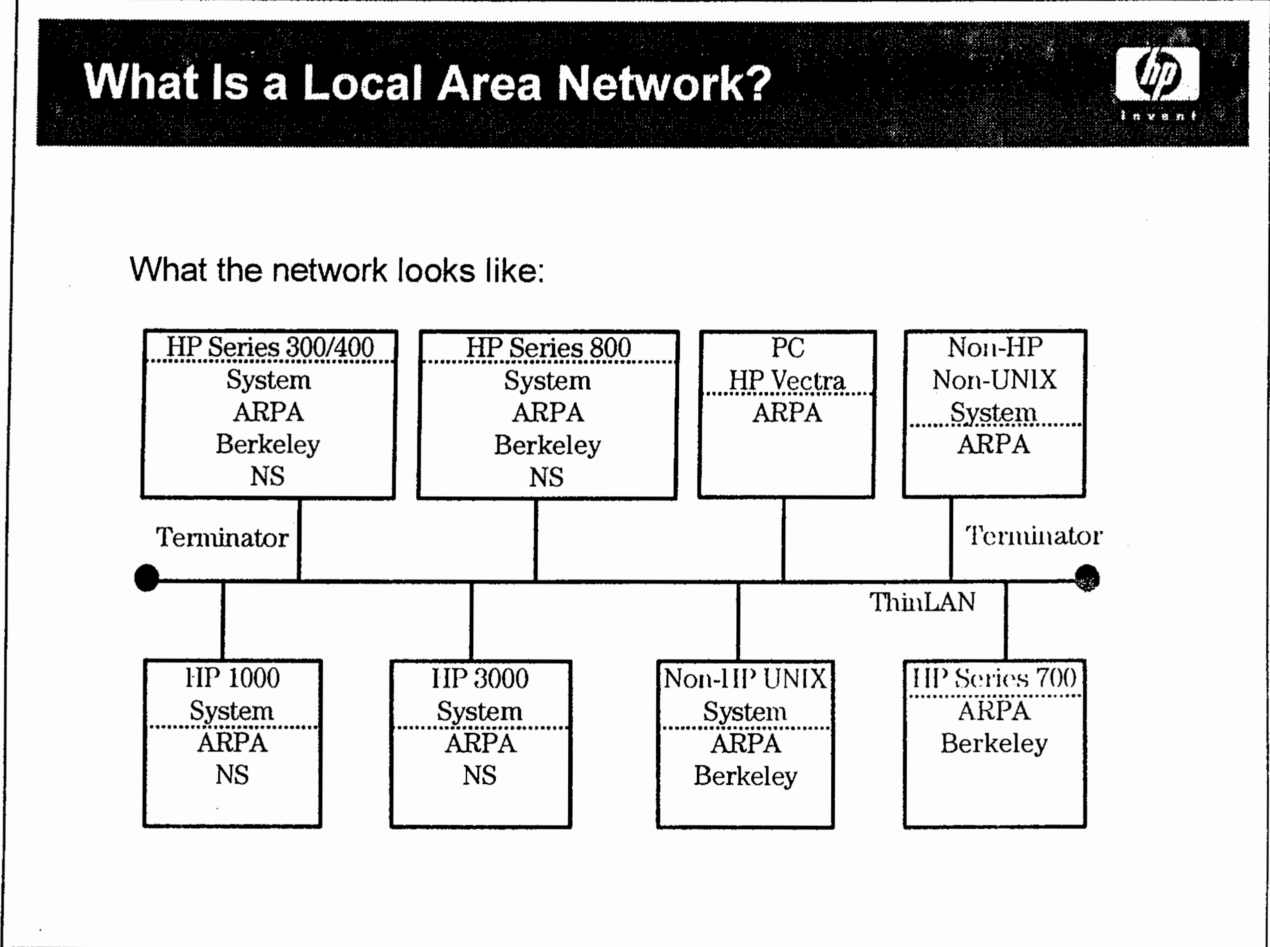
Objectives

Upon completion of this module, you will be able to do the following:

- Describe the different network services in HP-UX.
- Explain the function of a local area network (LAN).
- Find the host name of the local system and other systems in the LAN.
- Use the ARPA/Berkeley Services to perform remote logins, remote file transfers, and remote command execution.

© 2003 Hewlett-Packard Development Company, L.P.

13–1. SLIDE: What Is a Local Area Network?



Student Notes

A **Local Area Network** (LAN) is a method of connecting two or more computer systems over a small area. Most installations that have more than one computer will install a LAN to allow the users to work on several different computers without physically picking up all of their work and moving to the computer they want to work on.

The LAN services discussed in this module are the programs that allow us to use the LAN to perform many tasks between computers. Some of these tasks are the following:

- Copy files from one computer to another. Without a LAN, you would have to make a tape copy of your files, walk it over to the other computer, and reload the tape.
- Log in to another computer from a terminal on the local computer. Normally you would have to actually go to the other computer to log in.
- Execute commands on another computer and see the results locally. Again, you would have to move to the other computer if you did not have a LAN.
- Access files on a remote computer. This means we will use the files on another computer's disk without copying the files to the local disk.

© 2003 Hewlett-Packard Development Company, L.P.

13–2. SLIDE: LAN Services

LAN Services

- Two groups of LAN services are
  - ARPA Services
  - Berkeley Services
- The services allow you to perform
  - remote logins
  - remote file copies
  - remote file access

Student Notes

In this module we will look at two different *groups* of services to perform the basic LAN functions we have discussed. These services are the following:

- ARPA Services
- Berkeley Services

The ARPA Services were first defined by the Defense Advanced Research Projects Agency (DARPA) in the late 1960s. These services became a standard for communicating to many different brands of computers across a single LAN. The ARPA Services that we will discuss are `telnet` and `ftp`.

DARPA hired the University of California at Berkeley and Bolt, Baranek and Newman (BBN of Massachusetts) to develop these services. In the mid 1970s Berkeley started working with the new UNIX operating system. They eventually developed a more robust set of services to be used between computers running the UNIX operating system. These are now called the Berkeley Services. We will introduce the Berkeley services `rcp`, `rlogin`, and `remsh` in this module.

© 2003 Hewlett-Packard Development Company, L.P.

13–3. SLIDE: The `hostname` Command

The hostname Command

Syntax:

`hostname`	Reports your computer's network name

Example:

```
$ hostname
fred
$
$ more /etc/hosts
192.1.2.1    fred
192.1.2.2    barney
192.1.2.3    wilma
192.1.2.4    betty
```

Student Notes

Your computer has a host name. This is the name that identifies your system on the LAN. To find your system's host name, use the **`hostname`** command.

```
$ hostname
fred
```

If you want to communicate with another computer on the LAN, you must know its host name. You can do this simply by asking the administrator of the other computer what the host name is. You should also check that you have a user account on the machines that you want to work with.

NOTE: In order to use any of the LAN services, you must be a valid user on the remote computer.

You can also find host names in the **`/etc/hosts`** file. However, if you are part of a large LAN installation, this file may contain several hundred entries.

© 2003 Hewlett-Packard Development Company, L.P.

13–4. SLIDE: The `telnet` Command

The telnet Command

Syntax:

`telnet hostname`	ARPA Service to remotely log in to another computer

Example:

```
$ telnet fred
Trying ...
Connected to fred.
Escape character is '^]'.

HP-UX fred 10.0  9000/715
login:
```

Student Notes

`telnet` is the remote login facility of the ARPA Services.

If you type the command

```
$ telnet hostname
```

you will see the login prompt for the computer called *hostname* on your screen. At this point, you can enter the user name and password that you use on that machine and you will be logged in.

Once you are logged in, your terminal looks as if it were a terminal on the remote computer. You can run shell commands or programs and even use the remote computer's line printer. *All of the work you do is being executed on the remote computer.* Your local computer is just passing the information to and from your terminal through the LAN.

To close a **`telnet`** connection, simply log off the remote computer using [Ctrl]+ [d] or **`exit`**.

© 2003 Hewlett-Packard Development Company, L.P.

13–5. SLIDE: The `ftp` Command

The `ftp` Command

Syntax:

`ftp hostname`	ARPA Service to copy files to and from a remote computer

ftp Commands:

`get`	Gets a file from the remote computer
`put`	Sends a local file to the remote computer
`ls`	Lists files on the remote computer
`?`	Lists all `ftp` commands
`quit`	Leaves `ftp`

Student Notes

To copy a file to or from a remote computer using the ARPA Services, use the **`ftp`** command. **`ftp`** stands for *file transfer protocol*. As with **`telnet`**, you must specify the host name of the remote machine:

```
$ ftp hostname
```

`ftp` will prompt you for your user name and password on the remote system. It requires that you have a password set on the remote computer. Once you give it the correct login information, you will be connected to *hostname*.

At this point you get the **`ftp>`** prompt. At this prompt you can use the numerous **`ftp`** commands to do your work. Here are a few of the common **`ftp`** commands for performing remote file transfers:

`get` ***`rfile lfile`*** This copies the file ***`rfile`*** on the remote computer to the file ***`lfile`*** on your local computer. You can also use full path names as file names.

© 2003 Hewlett-Packard Development Company, L.P.

put *lfile rfile*	This will copy the local file ***lfile*** to the remote file named ***rfile***.
ls	List the files on the remote computer. This works just like the **ls** command we have been using.
?	List all of the **ftp** commands.
help *command*	Display a brief (very brief) help message for ***command***.
quit	Disconnect from the remote computer and leave **ftp**.

If, for example, you want to copy your local file called **funfile** to the **/tmp** directory on another computer whose host name is fred, your session would look something like the following. (The underlined text is what you type.)

```
$ ftp fred
Connected to fred.
220 fred FTP server (Version 1.7.109.2 Tue Jul 28 23:46:52 GMT 1992)
ready.

Name (fred:gerry): [Return]
Password (fred:gerry): Enter your password and press
[Return]
331 Password required for gerry.
230 User gerry logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> put funfile /tmp/funfile
200 PORT command successful.
150 Opening BINARY mode data connection for /tmp/funfile.
226 Transfer complete.
3967 bytes sent in 0.19 seconds (20.57 Kbytes/sec)

ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls .
-rw-rw-rw-  1 root sys 347 Jun 14 1993 exercises
-rw-rw-rw-  1 root sys  35 Oct 23 1993 cronfile
-rw-r----- 1 root sys  41 Jul 6 17:19 fio
-rwxrw-rw-  1 root sys 153 Oct 23 1993 initlaserjet
-rw-rw-rw-  1 root sys  37 Nov 21 1994 funfile
226 Transfer complete.
ftp> bye
221 Goodbye.
```

The first thing you will notice about **ftp** is that it is very verbose. It has a response for every command you type. (You can tell that it was not originally a UNIX system facility!)

© 2003 Hewlett-Packard Development Company, L.P.

Due to the default system security policies, applied to Linux, it may be more difficult for a user to remotely copy files, using the `ftp` command, to or from a Linux network host.

© 2003 Hewlett-Packard Development Company, L.P.

13–6. SLIDE: The rlogin Command

The rlogin Command

Syntax:

`rlogin` *hostname*	Berkeley Service to remotely log in to another computer; `rlogin` attempts to log you in using local user name

Example:

```
$ hostname
barney
$ rlogin fred
Password:
$ hostname
fred
$ exit
$ hostname
barney
```

Student Notes

The **rlogin** command performs functions similar to the **telnet** command. If you type

```
$ rlogin hostname
```

you will be logged in automatically to the system named *hostname*. **rlogin** assumes that you are logging in to the remote computer with the same name you used to log in to the local system. As a result, it does not have to prompt you for your user name.

If your system administrator has a file called **/etc/hosts.equiv** configured, **rlogin** will not even prompt you for a password. This makes it very quick and easy to use. A file called **.rhosts** can be created in your **HOME** directory which would also let you log in remotely to that computer without using a password. See **hosts.equiv(4)** for more information on the format of **.rhosts**.

As with **telnet**, to disconnect from the remote computer, simply log off.

© 2003 Hewlett-Packard Development Company, L.P.

13–7. SLIDE: The `rcp` Command

The `rcp` Command

Syntax:

`rcp` *source_pathname target_pathname* — Berkeley Service to copy files to and from a remote computer; works just like the `cp` command

Remote file names are specified as *hostname:pathname*

Example:

```
$ rcp funfile fred:/tmp/funfile
$
```

Student Notes

rcp stands for remote **cp**. That is because it works just as the **cp** command does. It works between two computers running the Berkeley Services. The general format of the command is

```
$ rcp host1:source host2:dest
```

in which the arguments mean copy the file **source** from **host1** to the file called **dest** on **host2**. **source** and **dest** could be full path names, of course.

If you are copying to or from a local file, you can leave off the local host name and the colon (`:`). Some examples will help make **rcp** clearer:

- Copy the file **funfile** on the local machine (called **bambam**) to **/tmp/funfile** on the system called **fred**:

  ```
  $ rcp funfile fred:/tmp/funfile
  ```

- Copy **/tmp/funfile** on **fred** to the **/tmp** directory on **barney**:

  ```
  $ rcp fred:/tmp/funfile barney:/tmp
  ```


© 2003 Hewlett-Packard Development Company, L.P.

All of the rules that apply to the `cp` command also apply to the `rcp` command.

NOTE: The file **`/etc/hosts.equiv`** or **`.rhosts`** must be configured correctly for **`rcp`** to work.

Due to the default system security policies applied to Linux, it may be more difficult for an end user to copy files to or from a Linux network host remotely using the **`rcp`** command.

© 2003 Hewlett-Packard Development Company, L.P.

13–8. SLIDE: The `remsh` Command

The `remsh` Command

Syntax:

`remsh  hostname command`	Berkeley Service to run a command on a remote computer

Example:

```
$ hostname
barney
$ remsh fred ls /tmp
backuplist
croutOqD00076
fred.log
Update.log
$ ls
EX000662    tmpfile   Update.log
$
```

Student Notes

remsh allows you to run a program on a remote computer and see the results on your terminal. The general form of the command looks like the following:

```
$ remsh hostname command
```

For example, if you want to see what is running on the system **fred**, you can execute:

```
$ remsh fred ps -ef
```

List the files in **fred**'s **/tmp** directory:

```
$ remsh fred ls /tmp
fredfile
funfile
reconfig.log
update.log
```


© 2003 Hewlett-Packard Development Company, L.P.

Or, if you want to view the **/etc/hosts** file on **fred**:

```
$ remsh fred cat /etc/hosts | more
```

Notice that **cat /etc/hosts** is the only command being executed on **fred**. The output is coming to our terminal and that output is being piped to **more**.

You can also use **remsh** to print files on a printer connected to another computer:

```
$ cat myfile | remsh fred lp
```

NOTE:	The file **/etc/hosts.equiv** or **.rhosts** must be configured correctly for **remsh** to work.

In Linux, the **rsh** command would be used to execute a remote shell. As mentioned previously, there are default Linux security policies that may inhibit the use of the remote shell capability.

© 2003 Hewlett-Packard Development Company, L.P.

13–9. SLIDE: Berkeley — The rwho Command

Berkeley — The rwho Command

Syntax:

`rwho` Displays users on machines in the LAN running the `rwh` daemon; produces output similar to `who`

Example:

```
$ rwho
user1    barney:tty0p1 Jul 18   8:23   :10
user2    wilma:tty0p1  Jul 18  10:13   :03
user3    fred:tty0p1   Jul 18  11:32   :06
```

Student Notes

The **rwho** command operates similarly to the **who** command, but will look for users on all of the systems in your LAN that are running the **rwho** daemon.

In addition to the **rwho** command, there is the **rusers** command. Both commands are meant to list the names of users who are logged into the local area network. However, in practice, both commands display a tendency not to work as expected, and invariably produce no output.

© 2003 Hewlett-Packard Development Company, L.P.

13–10. SLIDE: Berkeley — The `ruptime` Command

Berkeley — The `ruptime` Command

Syntax:

`ruptime` Displays the status of each machine in the LAN; Each system must be running the `rwho` daemon.

Example:

```
$ ruptime
barney  up      3:10  1 users load 1.32, 0.80, 0.30
fred    up    1+5:15  4 users load 1.47, 1.16, 0.80
wilma   down    0:00
```

Student Notes

The **`ruptime`** command will display the status of the systems in the LAN, whether they are up or down, how many users are currently running on each system, and machine loading information.

Looking at the entry for **`fred`** on the slide:

- **`fred`** in presently up.
- **`fred`** has been up for 1 day, 5 hours and 15 minutes.
- **`fred`** has 4 users logged in.
- Over the last 1-minute interval, an average of 1.47 jobs have been in the run queue.
- Over the last 5-minute interval, an average of 1.16 jobs have been in the run queue.
- Over the last 15-minute interval, an average of 0.80 jobs have been in the run queue.

The **`ruptime`** command in Linux is also dependent on the **`rwho`** daemon process.

© 2003 Hewlett-Packard Development Company, L.P.

13–11. LAB: Exercises

Directions

Ask your instructor which exercises you can do in the classroom. Also find out the host names of the computers with which you can communicate.

NOTE: Linux system users may not be able to carry out questions 3 and 4, because of the default security policy applied to Linux systems and configuration issues.

1. Use the **`hostname`** command to determine the name of your local system. What systems can you communicate with?

2. Use **`telnet`** to log in to another computer. Use the **`hostname`** command to verify that you are connected to the correct computer. Log off the remote computer when you have finished.

3. Transfer one of your files to your *HOME* directory on a remote computer using **`ftp`**, and then use **`rcp`** to copy another file to the remote machine. Notice the differences.

4. Use **`remsh`** to list the contents of the remote directory to verify that the copy worked.

© 2003 Hewlett-Packard Development Company, L.P.

Module 14 — Introduction to the vi Editor

Objectives

Upon completion of this module, you will be able to do the following:

- Use `vi` to effectively edit text files.

© 2003 Hewlett-Packard Development Company, L.P.

14–1. SLIDE: What Is vi?

What Is vi?

- A screen-oriented text editor
- Included with most UNIX system distributions
- Command driven
- Categories of commands include
  - General administration
  - Cursor movement
  - Insert text
  - Delete text
  - Paste text
  - Modify text

Student Notes

`vi` (pronounced *vee eye*, meaning *visual*) is the standard text editor that is supplied with most UNIX system distributions. A text editor is an interactive computer program that allows you to enter or modify text in a file. You can use `vi` to create new files or alter existing ones.

William Joy developed vi at the University of California at Berkeley. It is a screen-oriented interactive editor. The contents of the file will be displayed to your screen, and as you make changes to the file, they are immediately displayed on the screen. (The UNIX system also supports batch-oriented text editors such as **ed**, **sed**, and **awk** where you submit a batch request to execute file changes.)

The `vi` editor was designed to be terminal independent, and commands have been mapped to almost every key of the standard keyboard. It was originally used on teletypes that had no special function or cursor keys. Therefore, it may or may not take advantage of special keys that are available on your terminal.

The advantage of this design philosophy is that `vi` can be used on almost any type of terminal. As it is available on most UNIX systems, you do not have to learn a new editor or a new set of editing keys every time you sit down to a different UNIX system.

© 2003 Hewlett-Packard Development Company, L.P.

The `vi` interface is easily customizable. Mappings allow any key on the keyboard to be customized, including the special feature keys.

This module is designed to provide you with basic `vi` literacy. Using `vi` proficiently is a skill that requires some practice. The more you practice, the better you will become. This chapter will provide you with a good foundation for basic file editing and enable you to enhance your skills at your own pace.

In the Linux environment, the `vim` text editor is used as a substitute for vi. The `vim` editor *(Vi IMproved)* is upwardly compatible with the `vi` editor. Refer to the appropriate man pages for details on differences and additional capabilities. The `vim` program also can be invoked using the command name of `vi`.

© 2003 Hewlett-Packard Development Company, L.P.

14–2. SLIDE: Why `vi`?

Why `vi`?

- On every UNIX platform
- Runs on any type terminal
- Very powerful editor
- Shell command stack uses it
- Other UNIX tools require it

Student Notes

No matter which UNIX machine or operating system you find yourself working with, `vi` will always be there. The `vi` editor is screen-like and was designed to operate on any type of ASCII terminal, regardless of the manufacturer. Screen editors require specific types of terminals, but `vi` will operate on any type. It is true that `vi` is not very friendly, but it will always be there when you need to edit a file.

Many newer editors are really word processors, which can also perform some editing functions. Word processors are usually very user friendly and are fairly intuitive. For small files, word processors or windows-type editors are very appropriate.

If a large shell program is to be edited, experienced users of vi find they are more productive with `vi` than a Windows editor. Although very cryptic, vi is a very efficient and powerful editor, able to edit multiple files simultaneously, and cut and paste text from one file to another quickly.

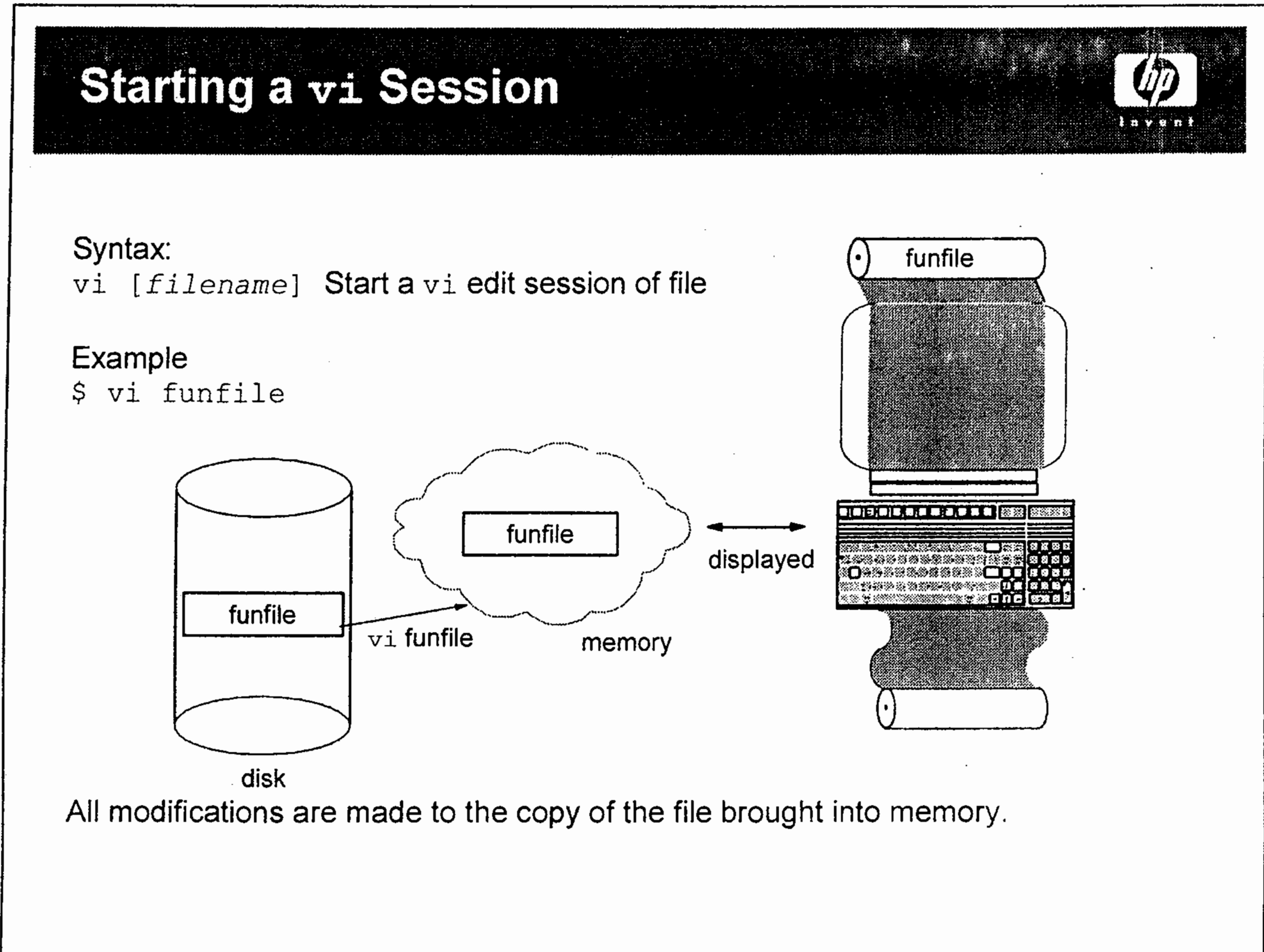
If you use the Korn shell or the POSIX shell, you may also have noticed that the commands used in manipulating the command stack are `vi` commands. Both the Korn and POSIX shells use `vi` as the preferred editor.

© 2003 Hewlett-Packard Development Company, L.P.

Many other tools in UNIX will put you into a vi session as a means of modifying configuration. To be considered an experienced user of UNIX, you must be a proficient user of this tool.

© 2003 Hewlett-Packard Development Company, L.P.

14–3. SLIDE: Starting a vi Session



Student Notes

Invoking the **vi** command will start an edit session. If the requested file already exists, the first screen of text will be displayed. Otherwise, if you are editing a new file, you will see a blank screen with tildes (~) running down the left column. **vi** brings a copy of your file into a temporary memory **buffer**. All of your modifications will be made to this temporary copy in memory. Only when you issue the command that saves the buffer to your disk will the copy of the file on the disk be updated. Therefore, if you determine that you have made unnecessary changes to your file, the temporary buffer can be discarded, and the image on the disk is not affected.

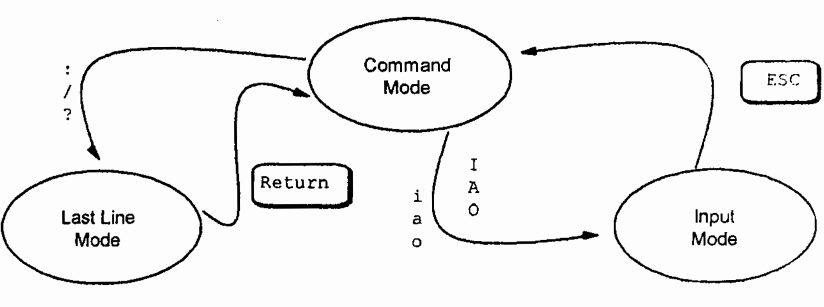
When editing a file, your screen becomes a window into the file that you are editing. You will generally make changes to the file at the character, word, or line that contains the cursor. Therefore, *you should focus on the cursor's location at all times*. As you make your modifications to the file they will be immediately displayed.

© 2003 Hewlett-Packard Development Company, L.P.

14–4. SLIDE: vi Modes

vi Modes

- **Command Mode** – keystrokes interpreted as commands
  - Initial mode when a vi session is started
  - ESC puts you in command mode
  - vi commands (i, a, o, etc.) not echoed to the screen
  - ex commands, prefixed with a colon (:), echoed to the screen
  - search commands (/ and ?), echoed to the screen
- **Last Line Mode** – ex and search inputs
- **Input Mode** – vi input with keystrokes entered into the file



Student Notes

vi is a *command-driven* editor. When you start a **vi** edit session, *you are in command mode*. Therefore, if you type any keys, **vi** will try to execute the associated commands. Almost every key on the keyboard is assigned to some **vi** function. Commands are available to input text, move the cursor, modify text, delete text, and paste text. Generally, **vi** commands are silent, which means that as you enter **vi** commands they will not be echoed to the screen. You will only see their effects.

ex is an extended line-oriented editor, whose commands are available from within your **vi** edit session. **ex** commands are entered at the colon prompt, and unlike **vi** commands, are echoed to the screen and are submitted by entering a Return. These commands are commonly used for multi-line modifications and session customization.

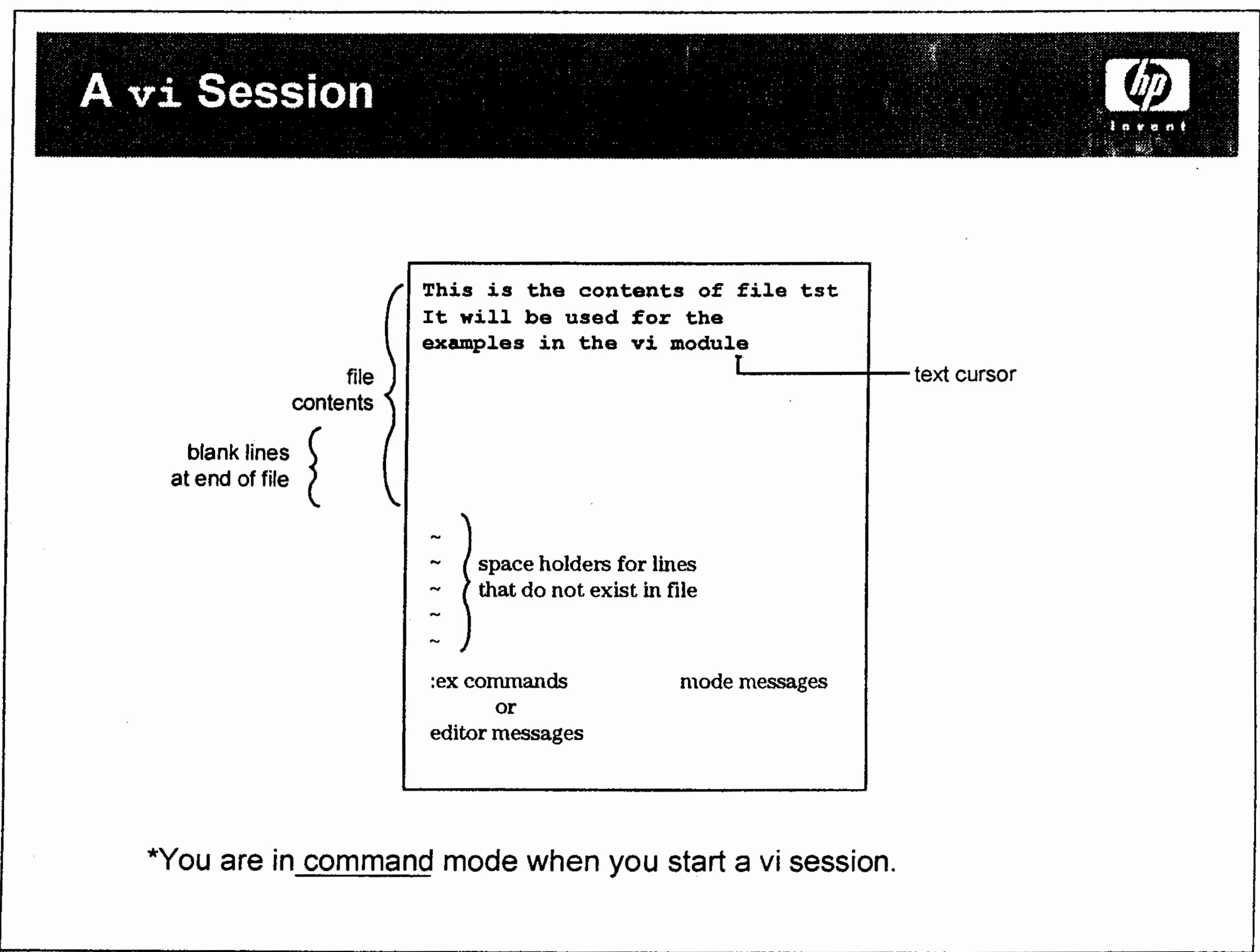
There are vi commands available to get into the *input mode*, where everything you type will be entered into your file. To return to the *command mode* just press the Esc key.

© 2003 Hewlett-Packard Development Company, L.P.

NOTE:	Some `vi` commands require multiple keystrokes. If you ever get lost in the middle of a `vi` command, just press the [Esc] key to terminate the current command.

© 2003 Hewlett-Packard Development Company, L.P.

14–5. SLIDE: A vi Session



Student Notes

When you start a **vi** session, the screen will appear similar to the illustration on the slide. On your screen you will view a *window* of your file. Be aware of the following components of the **vi** display:

text cursor	points to a character in the file
text area	displays the contents of the file
ex command area	echoes **ex** commands and **vi** editor messages
mode message area	displays mode status

You should always focus on the location of the cursor, since it will generally point to the character, the word of the line that you want to modify.

You should also be aware of the messages that appear in the *mode message* area. The editor can remind you when you are in *input* mode or *replace* mode. These visual cues will greatly assist you during your edit sessions.

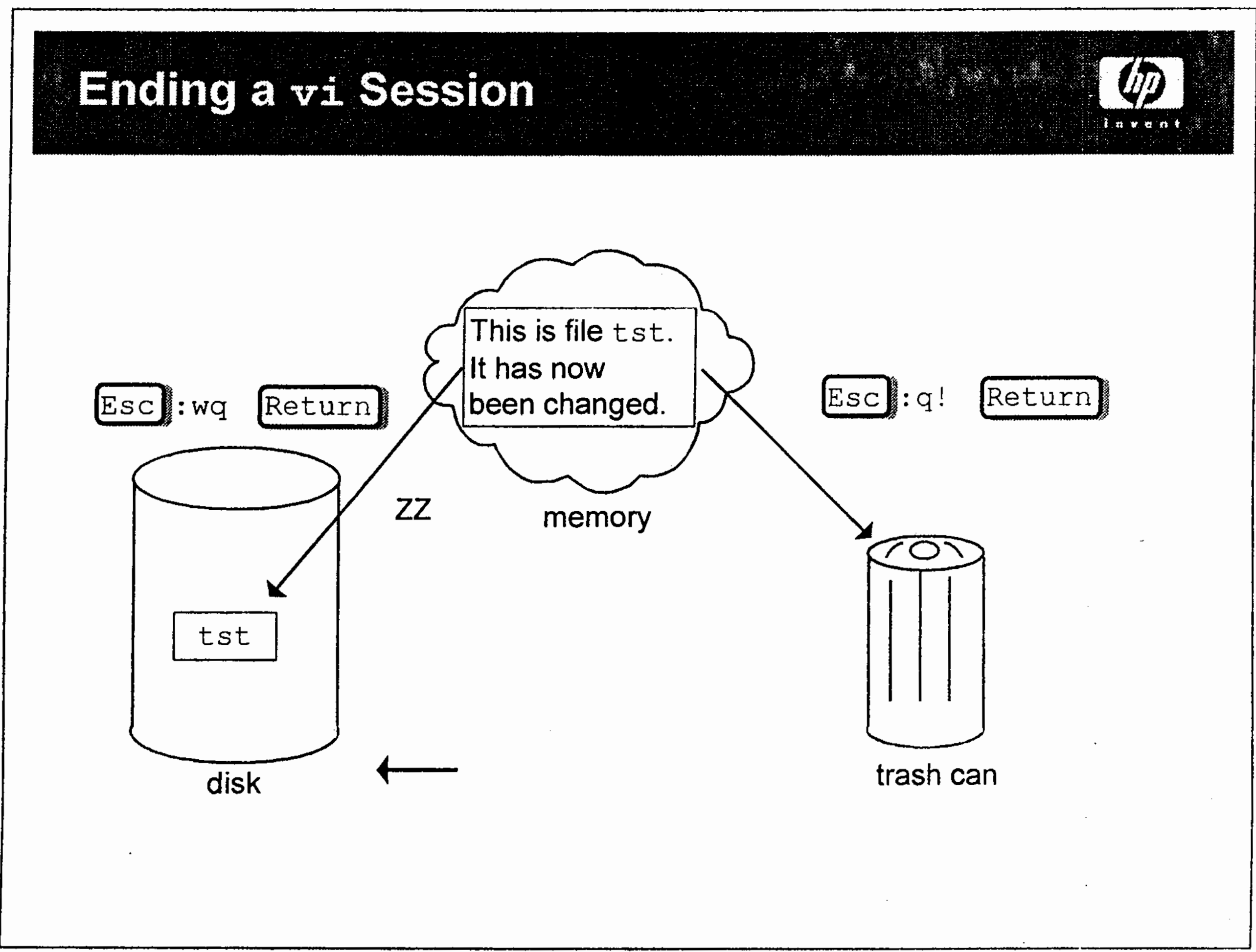
© 2003 Hewlett-Packard Development Company, L.P.

The tilde's (~) that you see on the slide signify space holders for display purposes. The file does not contain these lines. If you want blank lines at the end of your file, you must physically insert them.

NOTE:	If you go into *input* mode and do not see a mode message signifying *input* mode, enter: Esc **:set showmode** Return

© 2003 Hewlett-Packard Development Company, L.P.

14–6. SLIDE: Ending a vi Session



Student Notes

When you have completed making changes to your file, you will need to save the temporary buffer contents to your disk. There are two commands available to save your file; one is a `vi` command, the other is an **ex** command. You must be in *command mode* to issue either command, so remember to press the [Esc] key to confirm that you are in command mode.

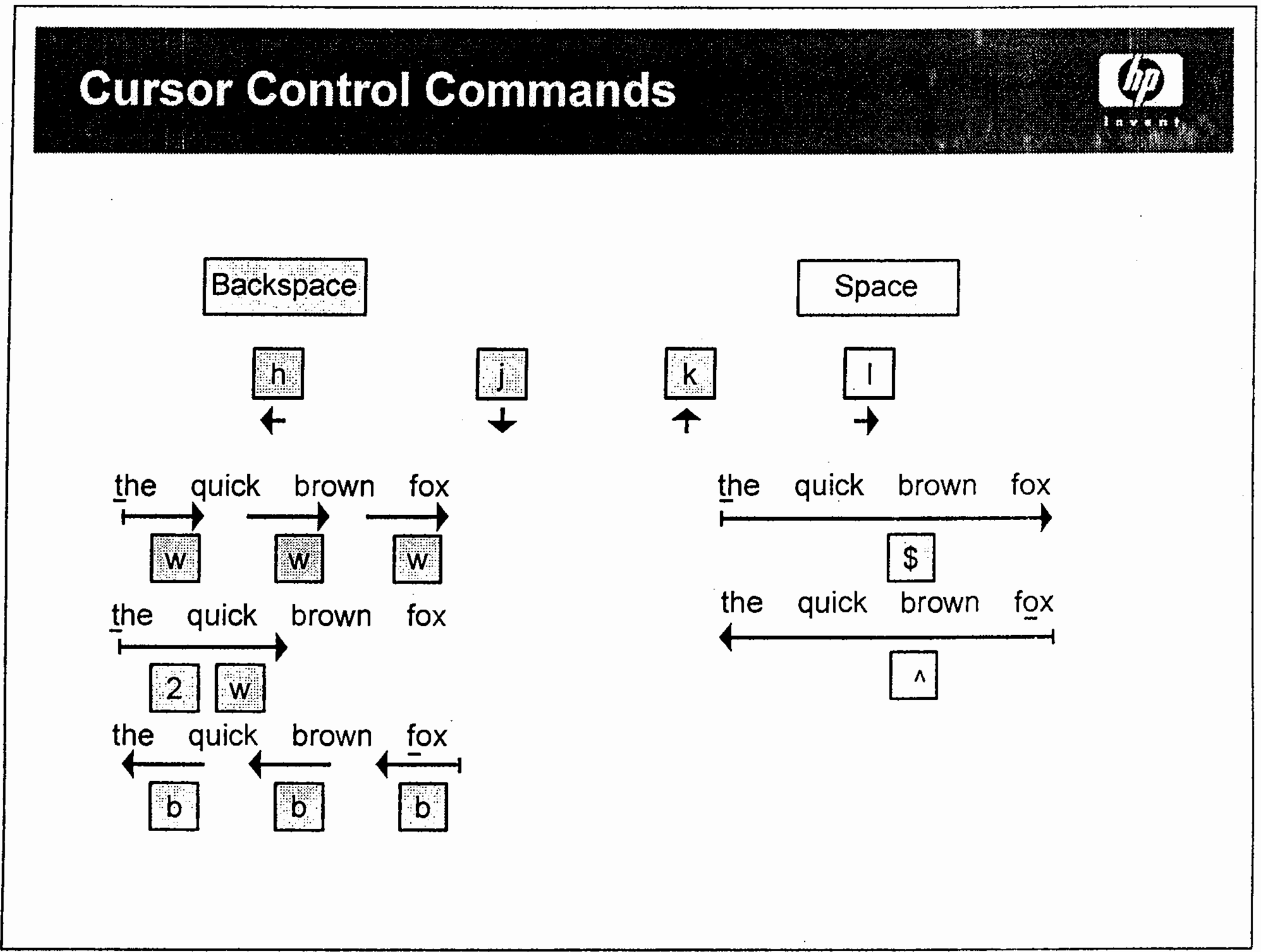
[Esc] **ZZ**	`vi` command — not echoed to the screen *put the file to bed*
[Esc] **:wq** [Return]	**ex** command — prefixed by a colon echoed to the lower left corner of your screen write and quit

There may be times when you do *not* want to save the changes that are in your buffer. An **ex** command is available to discard the buffer:

[Esc] **:q!** [Return]	Means quit! I really mean it! *(I know that I'm throwing my changes away.)*

© 2003 Hewlett-Packard Development Company, L.P.

14–7. SLIDE: Cursor Control Commands



Student Notes

The first category of commands that you will learn will allow you to move the cursor throughout your text. Remember the cursor points to the position in the file that you want to modify.

You will notice that simple cursor movement is executed through the `h`, `j`, `k`, and `l` commands. Remember that teletypes did not have cursor keys on them, so other keys had to be defined to move the cursor. Most current `vi` configurations, though, will support the use of the cursor keys (↓, ←, →, ↑) to move the cursor. If you are a touch typist, you should find the cursor control commands very convenient.

Most `vi` commands are an abbreviation for some associated meaning. Learn the abbreviation and its corresponding meaning and the command will be easier to recall. Where appropriate, the command meanings will be provided in the command summary tables.

© 2003 Hewlett-Packard Development Company, L.P.

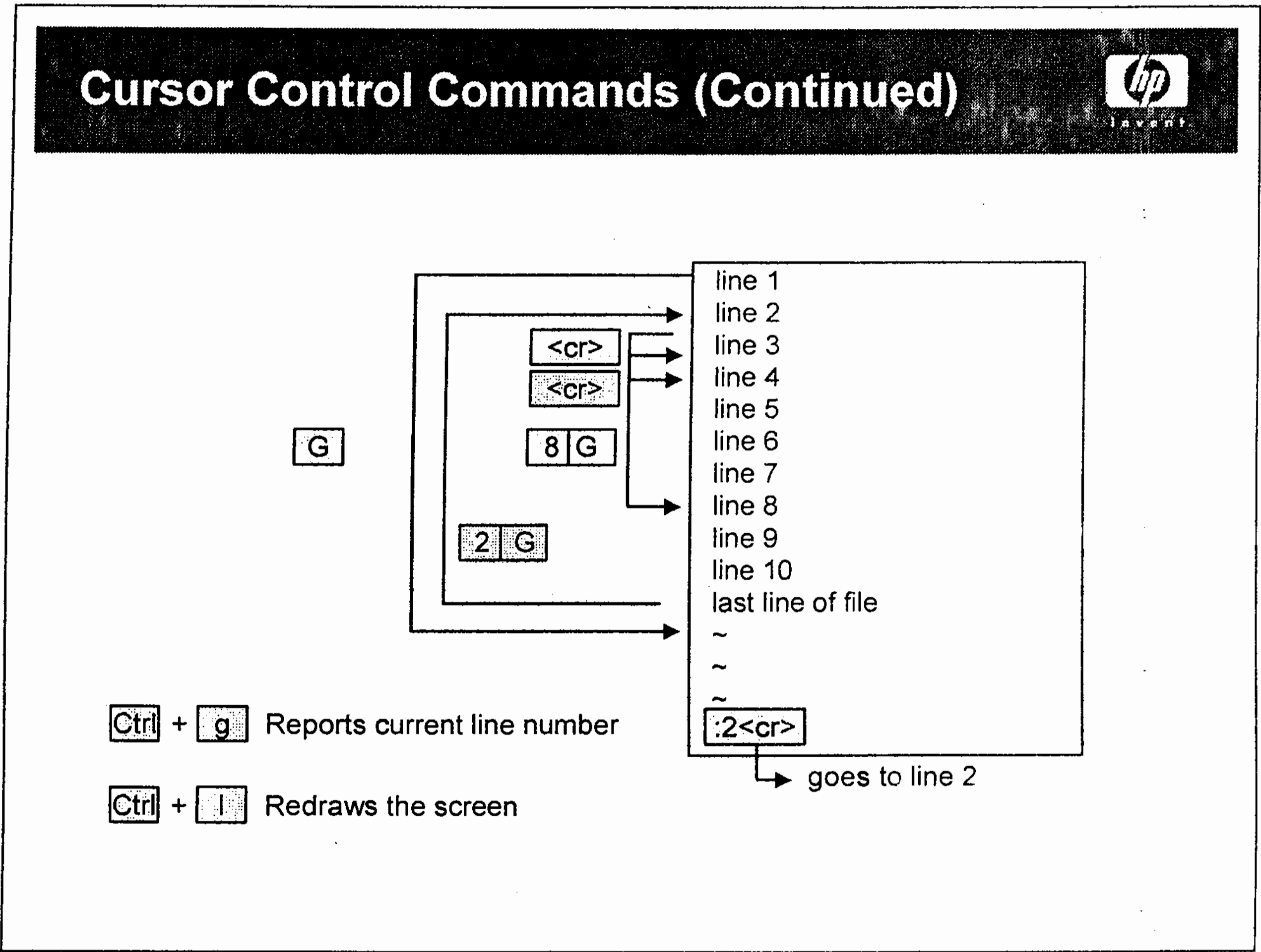
Cursor Control Summary

`h` or [Backspace]	Move left one character.
`j`	Move down one line.
`k`	Move up one line.
`l` or [Space]	Move right one character.
`# w`	Move forward **word** by **word** (`W` ignores punctuation.)
`# b`	Move **backwards** word by word. (`B` ignores punctuation.)
`# e`	Move to the **end** of the next word. (`E` ignores punctuation.)
`$`	Go to the end of the current line.
`^` or `0`	Go to the beginning of the current line.

Many vi commands can be prefixed with a number to repeat the command. Therefore, if you want to move forward by *6 words* you would issue the `6w` command, or if you want to move *3 words backward* you would issue `3b` command.

© 2003 Hewlett-Packard Development Company, L.P.

14–8. SLIDE: Cursor Control Commands (Continued)



Student Notes

Additional cursor control commands allow you to move to a specific line in the file or scroll your display.

Cursor Control Summary (Continued)

G	**Go** to the end of the file.
#G	**Go** to the line number **#**.
: #	Go to the line number **#**.
Ctrl + g	Reports the line you have **gone** to (the current line).
Return	Go to first non-blank character on next line.
Ctrl + b	Scroll **back** to previous window of text.
Ctrl + f	Scroll **forward** to next window of text.

© 2003 Hewlett-Packard Development Company, L.P.

Ctrl + `u`	Scroll **up** half a window of text.
Ctrl + `d`	Scroll **down** half a window of text.
`L`	Go to the **last** line on the screen.
`M`	Go to the **middle** line on the screen.
`H`	Go **home** (first line, first character) on the screen.
Ctrl + `l`	Redraws the screen (helpful if someone writes a message to you in the middle of an edit session).

NOTE: If you like to see line numbers while you are editing your file you can enter the command:

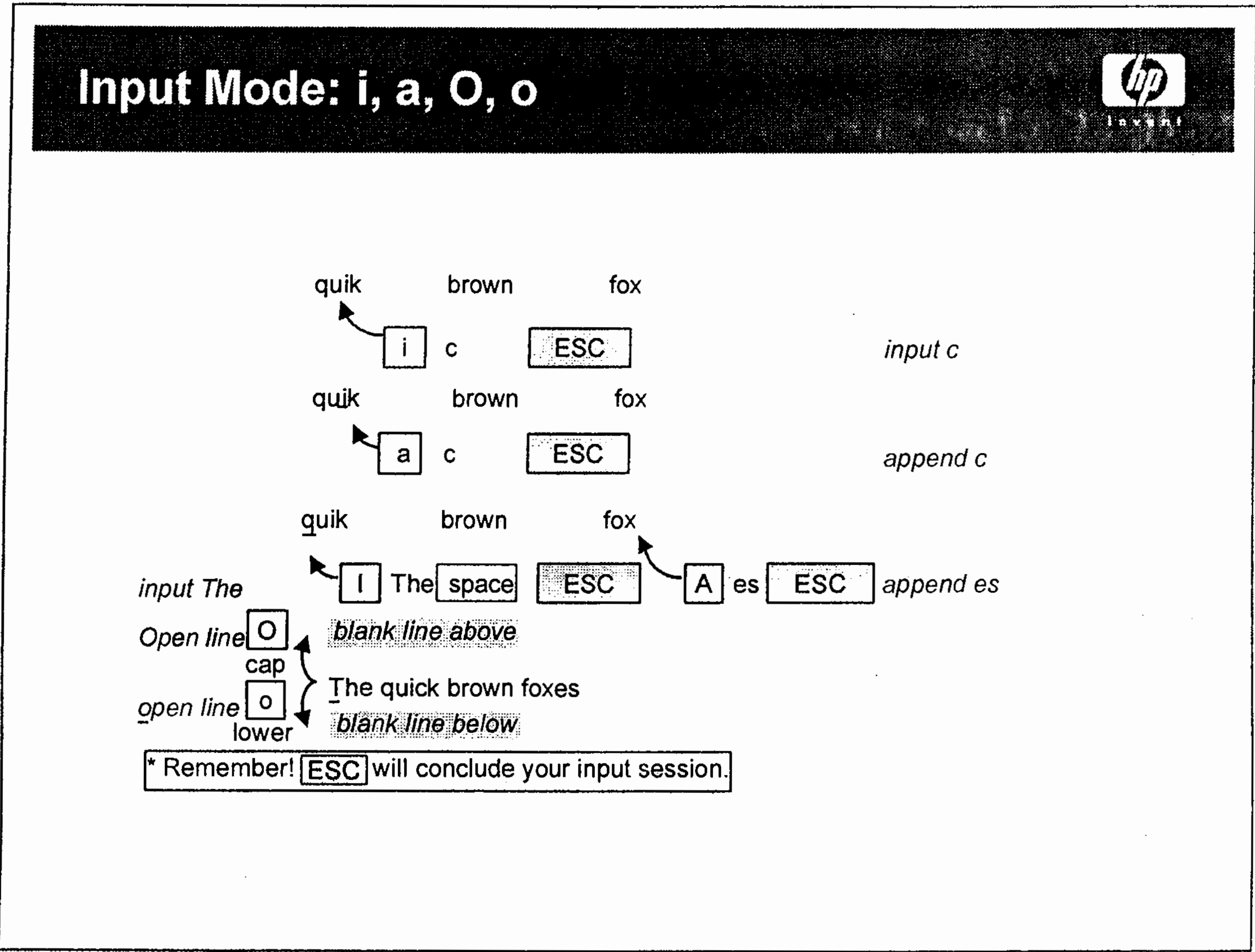
`:set number` Return

You can disable line numbers with:

`:set nonumber` Return

© 2003 Hewlett-Packard Development Company, L.P.

14–9. SLIDE: Input Mode: i, a, O, o



Student Notes

In order to input text into your file, you must go into *input mode* . There are actually several commands that will toggle you into *input* mode .

Input Mode Summary

`a`	append new text after the cursor
`i`	insert new text before the cursor
`O`	Open a line for text above the current line
`o`	open a line for text below the current line
`A`	Append new text at end of the line
`I`	Insert new text at beginning of the line

When you are in input mode you should see an *input mode* message appear in the lower right corner of your screen.

© 2003 Hewlett-Packard Development Company, L.P.

Once in *input mode* you can enter text into your file. In *input mode* a [Return] will provide a new, blank line. If you need to split a line, you move the cursor to where you want to split the line, and then insert a carriage return character into the file (remember that in *command mode* [Return] *moves* your cursor to the first non-blank character on the next line).

NOTE: To get back to *command mode*, type the [Esc] key. Note that when you toggle from input mode to command mode, the cursor will back up (move left) one character.

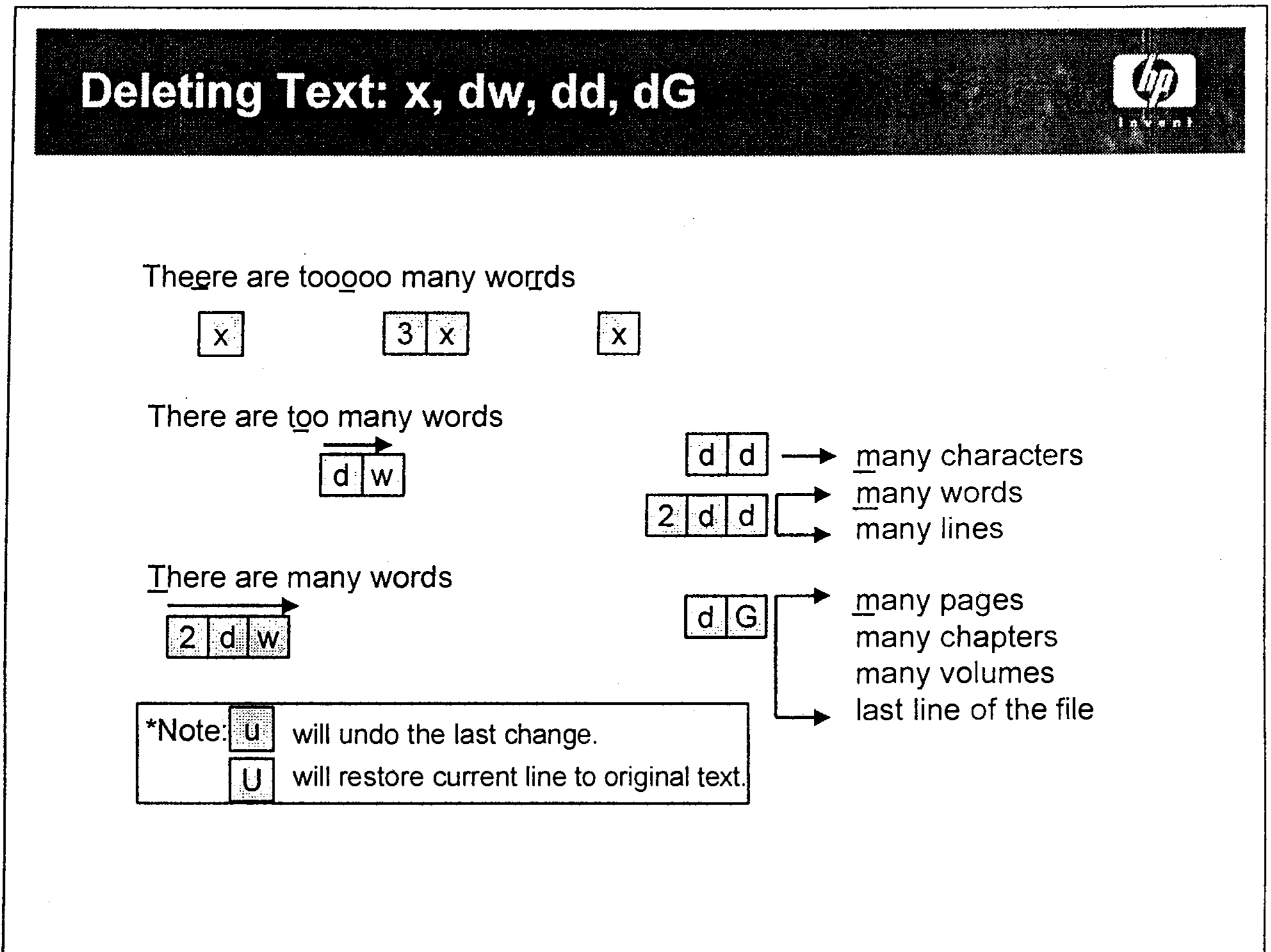
Correcting Typing Mistakes

While you are in *input mode*, you can use the [Backspace] key to backup to where the mistake occurred and reenter your text. *Beware*, as you backup through your text, the characters are *not* erased from your display, but they are erased from the buffer.

You can use [Backspace] to correct typing mistakes for the *current* input session. Pressing the [Esc] key concludes an input session. To modify text that was entered in a previous input session you must use `vi` commands.

© 2003 Hewlett-Packard Development Company, L.P.

14–10. SLIDE: Deleting Text: x, dw, dd, dG



Student Notes

Two commands are available to delete text:

#x **x** out (delete) the character at the cursor

#d *object* **Delete** the named object

The d (delete) command is an active command that requires an object to act upon. The specified object will be deleted. Objects are defined with the cursor motion commands.

Delete Summary

#dw delete the current **word**

#dd delete the current line. (**vi** convention: when the **action** is repeated, it affects the entire line.)

dG delete through the last line of the file.

d$ delete to the end of the line.

© 2003 Hewlett-Packard Development Company, L.P.

`d^` delete to the beginning of the line.

The delete command can be prefixed with a number to repeat the command. Therefore, if you want to *delete 6 words* you would issue the command `6dw`, or if you want to *delete 3 lines* you would issue the command `3dd`.

NOTE: Focus on your cursor position. Most objects are defined relative to the current cursor position.

The Undo Command

As a new `vi` user, you might delete or modify something that was not intended to be deleted or modified. The `u` (undo) command will undoubtedly come to your rescue.

`u` **undo** the last modification.

`U` **Undo** *all* modifications to *current line.*

`u` will undo the previous change that you made to your file. If you immediately issue another `u`, this will undo the undo, reverting the text to its previous state before the first undo. If you have made several changes to a line of text, you can issue the `U`, which will return the line to the text it held when your cursor first entered the line. Therefore, for `U` to work, it has to be issued *before your cursor leaves the line.*

© 2003 Hewlett-Packard Development Company, L.P.

14–11. LAB: Adding and Deleting Text and Moving the Cursor

Directions

Use `vi` to start an editing session on the file `tst` found in your HOME directory. Complete the following exercises and answer the associated questions.

1. `vi` the file `tst`.

2. Insert the word *only* between the words *will be*.

3. Add the words *many*, *many* on the end of the line "*It will be used for.*"

4. Add a new blank line at the end of the file, and enter your name. DON'T PRESS THE [Esc]!

5. Using the [Backspace], remove your name and enter your partner's name.

6. Open a new line at the top of your file (Hint: O).

© 2003 Hewlett-Packard Development Company, L.P.

7. Enter `12345`.

8. [Backspace] 2 times. Do any of the numbers disappear from your display?

9. Enter `1234`. What happens to the numbers that you backspaced over?

10. [Backspace] 3 times.

11. Press [Esc]. What happens to the characters you backspaced over? Where does the cursor end up?

12. Type in 4 a's. How many a's appear? Why?

© 2003 Hewlett-Packard Development Company, L.P.

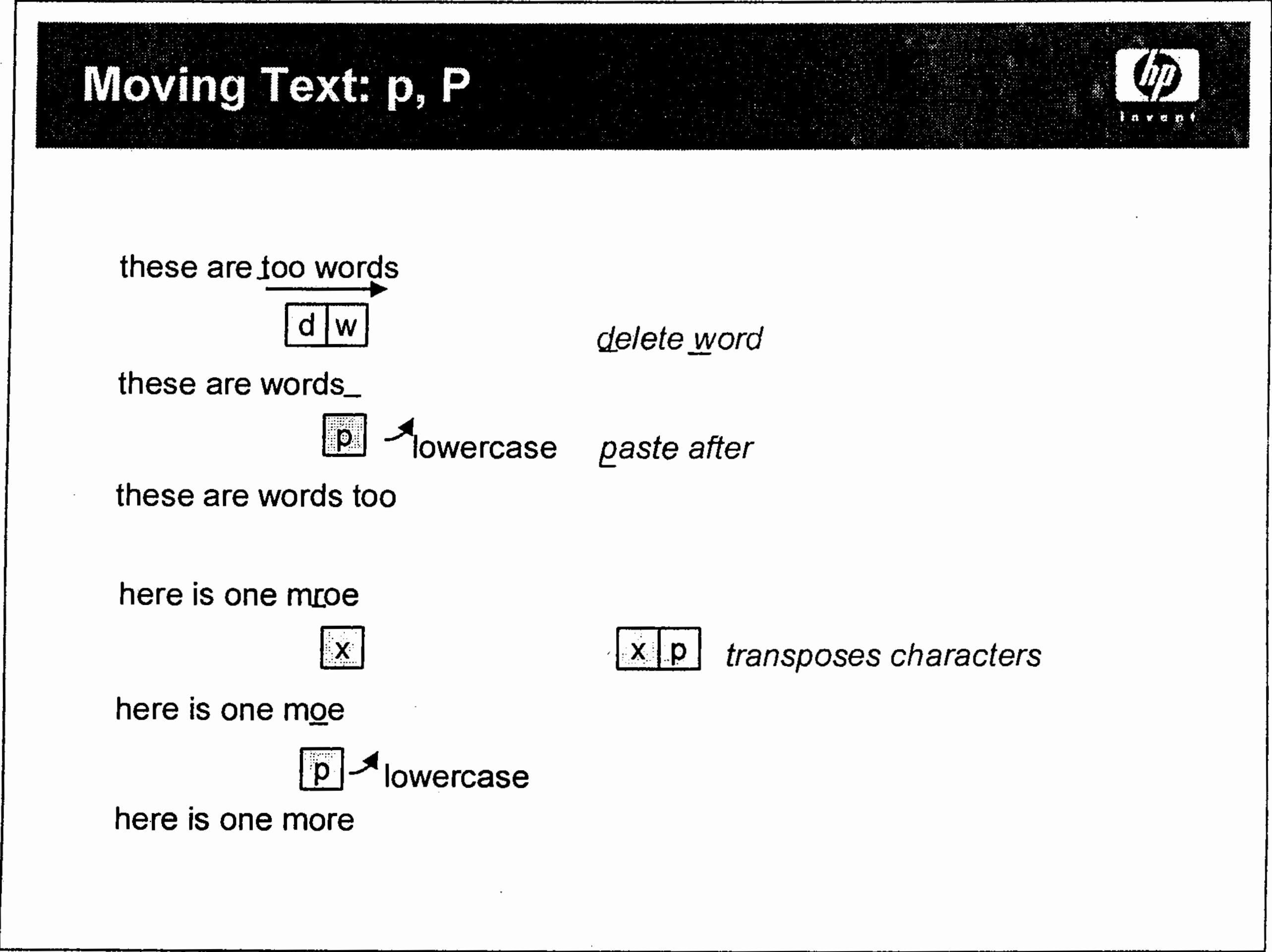
13. [Backspace] 5 times. What happens? Why?

14. Press [Esc]. What happens?

15. Quit your `vi` session saving the changes you made to the file `tst`.

© 2003 Hewlett-Packard Development Company, L.P.

14–12. SLIDE: Moving Text: p, P



Student Notes

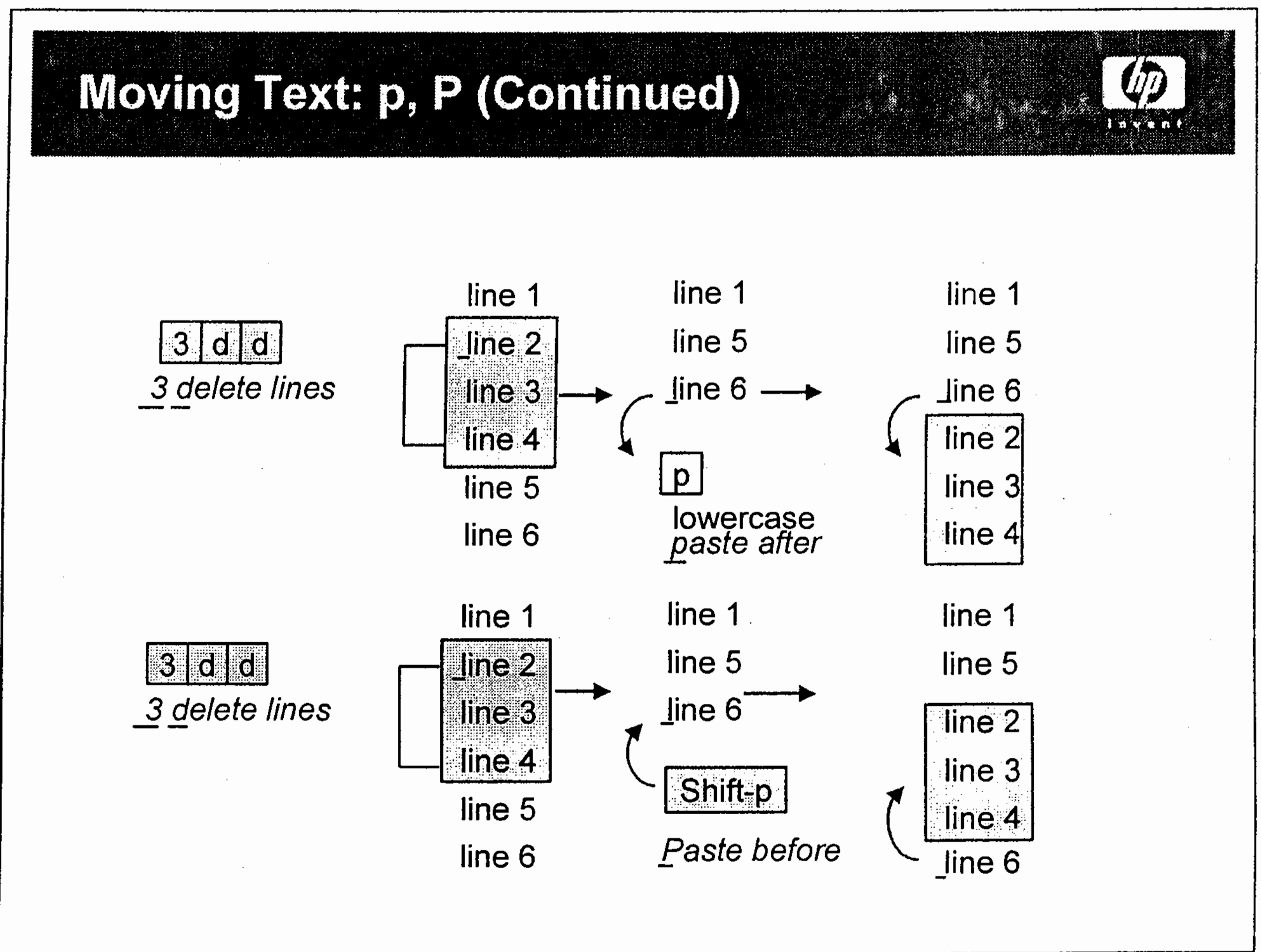
Whenever you delete an object, it is saved in a temporary cut buffer. The contents of the next delete operation that you perform will replace the contents of this cut buffer. The p (paste) command allows you to retrieve the text from the cut buffer and paste it back into your file. Text is pasted relative to the cursor position.

You can easily move text from one position to another by deleting a block of text into the cut buffer, moving the cursor to the desired destination, and pasting the contents of the cut buffer at the cursor.

Since the contents of the cut buffer are replaced when you execute the next delete operation, you must be careful to retrieve the contents *before* you complete your next delete.

© 2003 Hewlett-Packard Development Company, L.P.

14–13. SLIDE: Moving Text: p, P (Continued)



Student Notes

Delete and Paste Summary

d*object* Delete ***object*** into the cut buffer.

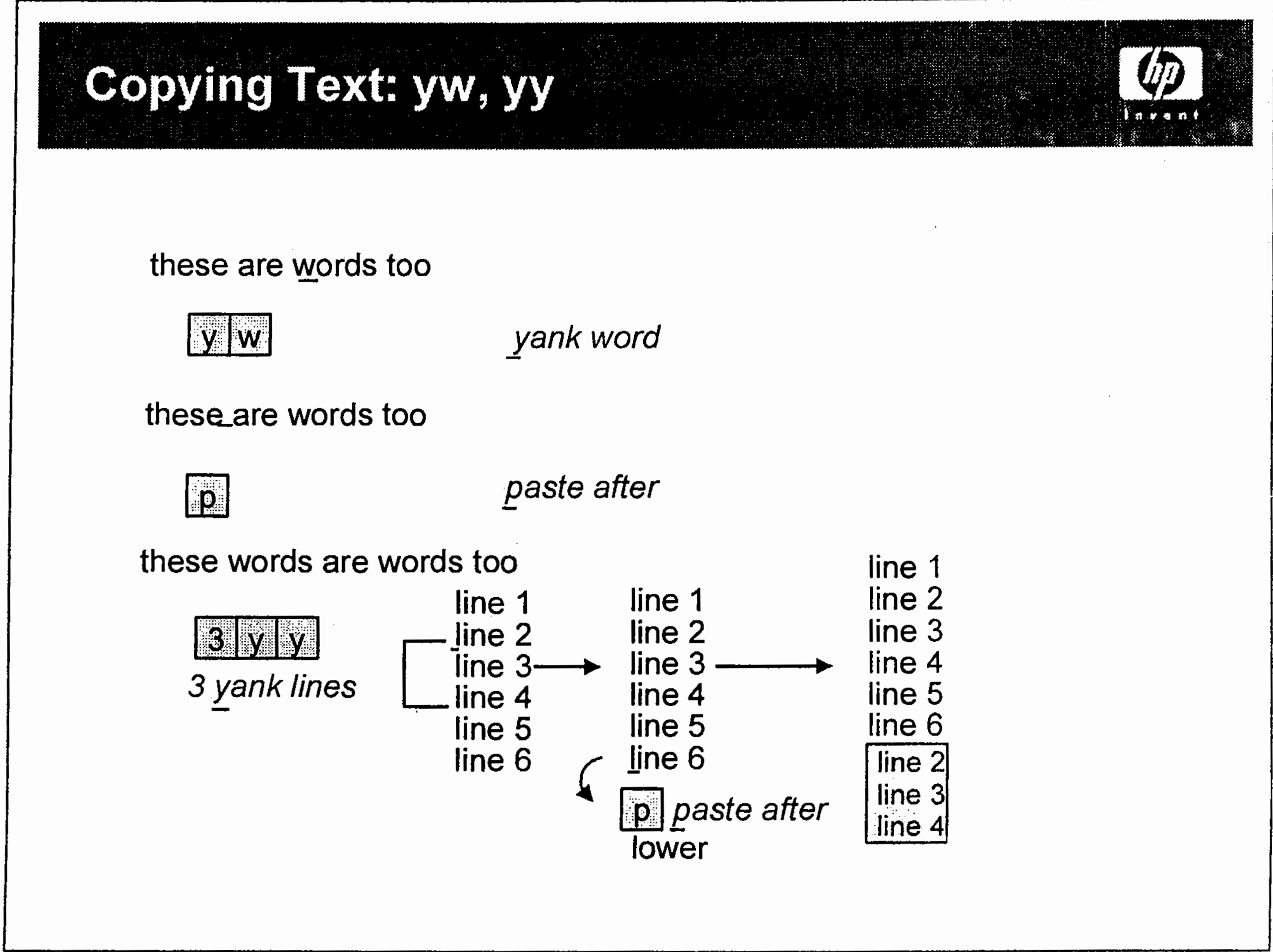
p (lowercase) **paste** contents of the cut buffer back into the text *after* the cursor.

P (uppercase) **Paste** contents of the cut buffer back into the text *before* the cursor.

This delete and paste operation allows you to easily move blocks of text, or transpose characters. If the cut buffer contains whole lines, the **p** (lowercase) will open a line below the current line, and the **P** (uppercase) will open a line above the current line.

© 2003 Hewlett-Packard Development Company, L.P.

14–14. SLIDE: Copying Text: yw, yy



Student Notes

The **y** (yank) command is an active command that also requires an object to act upon. The specified object will be yanked (copied) into the cut buffer. You can easily copy this text to another position in the file by moving the cursor, and pasting the contents of the cut buffer. This is very similar to the move operation.

Yank Summary

#yw	**yank** the current **word**
#yy	yank the current line. (`vi` convention: when the action is repeated, it affects the entire line.)
yG	yank through the last line of the file.
y$	yank to the end of the line.
y^	yank to the beginning of the line.

© 2003 Hewlett-Packard Development Company, L.P.

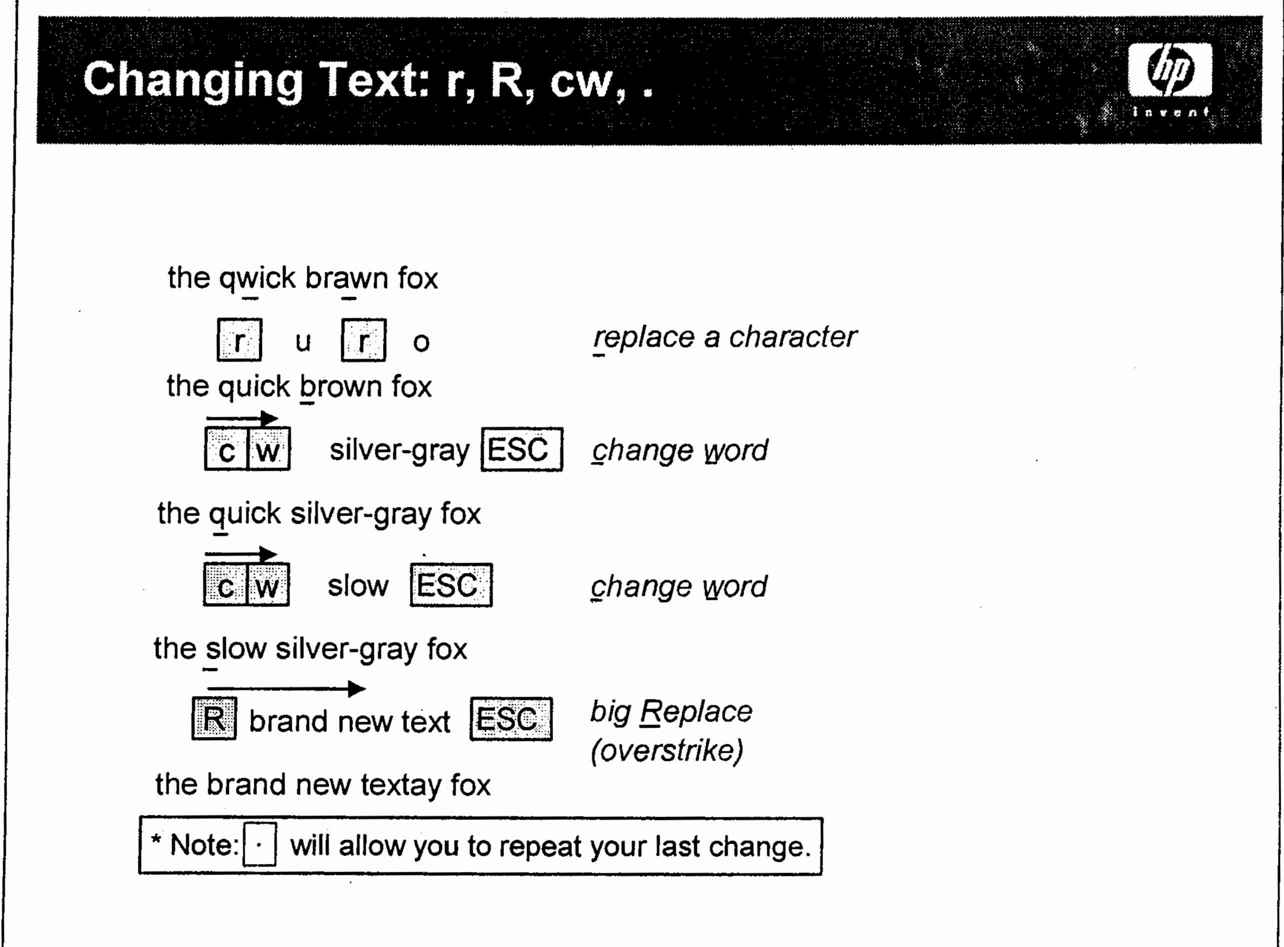
Copy and Paste Summary

`yobject`	**yank** *`object`* into the cut buffer.
`p` (lowercase)	**paste** contents of the cut buffer back into the text *after* the cursor.
`P` (uppercase)	**Paste** contents of the cut buffer back into the text *before* the cursor.

The yank and paste operation allows you to easily copy blocks of text. If the cut buffer contains whole lines, the `p` (lowercase) will open a line below the current line, and the `P` (uppercase) will open a line above the current line.

© 2003 Hewlett-Packard Development Company, L.P.

14–15. SLIDE: Changing Text: r, R, cw, .



Student Notes

Once you know how to input and delete text, you are equipped to make any changes to your file. This can be somewhat cumbersome though because you have to manually toggle back and forth between command and input mode. The commands that allow you to change text make text modification more convenient.

There are three common command primitives that are used to modify text:

r*character* — **replaces** the character at the current cursor position with the named ***character***.

R — **REPLACES** all characters (goes into overstrike mode) until ESC is pressed.

c*object* — **Changes** the named ***object***. This replaces the identified object (the end of the object is marked with a **$** symbol) with the text that you enter. This must also be concluded with an ESC.

The **c** (change) command is also an action command that acts upon objects. The objects are defined by the motion control commands.

© 2003 Hewlett-Packard Development Company, L.P.

Change Summary

#cw	**change** the current **word**
#cc	change the current line entirely. (Duplicate the action — remember?)
cG	change through the last line of the file.
c$	change to the end of the line.
c^	change to the beginning of the line.

The Dot (.) Command

The dot command is probably one of the handiest commands available in the `vi` command collection. This allows you to repeat your last change operation (this includes deletes too).

An Example

Assume that the original text of a file contains:

```
walk walk walk walk walk walk
```

And you would like it to read:

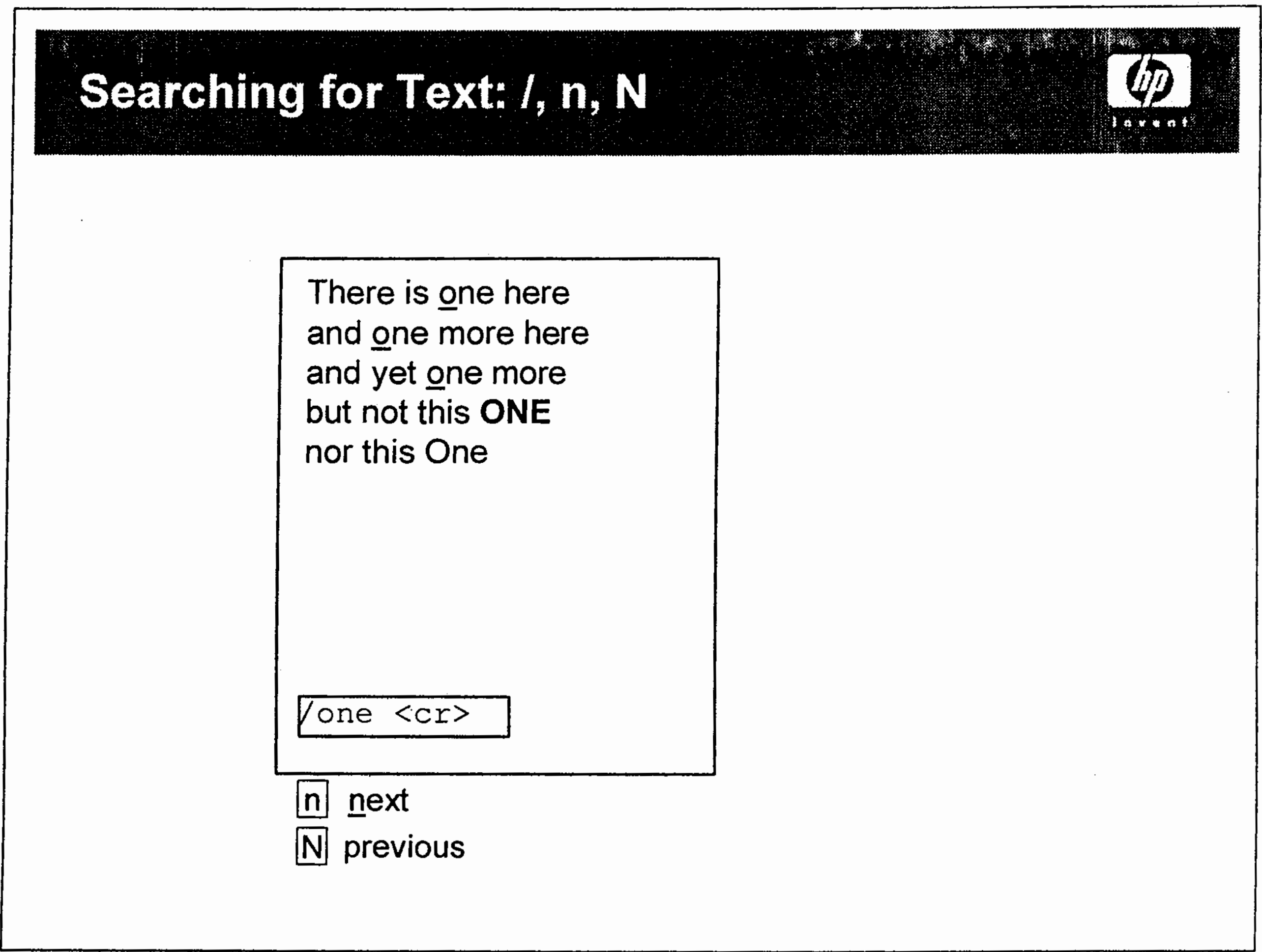
```
run walk run walk run walk
```

You could delete the entire line and retype it, but look what the dot command can do for you:

1. Move your cursor to the first occurrence of the word walk that you want to change.
2. Execute the **vi** command to change a word: **cw**
   enter: run [ESC]
3. Advance your cursor to the next word you want to change: **ww**
4. Now instead of executing another **cw** command, just issue a dot (.). This will repeat the last change, which was to do a *change word to run.* You can repeat this as many times as you like.

© 2003 Hewlett-Packard Development Company, L.P.

14–16. SLIDE: Searching for Text: /, n, N



Student Notes

A common requirement when editing a file is to *search* for a specific text string. The `/` command allows you to locate an occurrence of the requested string. The `n` command allows you to find the **next** occurrence.

Text Search Summary

`/text`	Search for `text` from the current line towards the end of the file, with wrap around.
`?text`	Search for `text` from the current line towards the beginning of the file, with wrap around.
n	Find the next occurrence of the previously searched for text, in the same direction.
N	Find the next occurrence of the previously searched for text, in the reverse direction.

© 2003 Hewlett-Packard Development Company, L.P.

Wrap around means that if the text is not found by the end (or beginning) of the file, the search will continue at the opposite end of the file.

© 2003 Hewlett-Packard Development Company, L.P.

14–17. SLIDE: Searching for Text Patterns

Student Notes

As mentioned on the previous slide, string searches are case sensitive. The previous examples would only succeed in matching the string literally specified. The constructs on this slide allow you to search for string patterns. These pattern specifiers are known as **regular expressions** and are recognized by several UNIX system utilities.

Regular Expression Summary

`[a-zA-Z0-9]`	Define a class of characters to match from. The characters `a-z` denotes a range of characters to match from. `[]` represents only *one* character position.
`^text`	Anchor ***`text`*** to the beginning of the line.
`text$`	Anchor ***`text`*** to the end of the line.
.	Match any single character.
`character*`	Match zero or more occurrences of ***`character`***.

© 2003 Hewlett-Packard Development Company, L.P.

Examples

`/[Tt]he`	Searches for the next occurrence of the string *The* or *the* .
`/[oO][nN][eE]`	Searches for the string one with any character in any case.
`/bo*t`	Searches for the string b followed by zero or more *o*'s, followed by *t*. This would match *bt*, *bottom*, *boot*, *booot*, and so on.
`/^[abc].*`	Searches for the next occurrence of a line that begins with an *a*, *b*, or *c*. This is read as: a line that begins with an *a*, *b*, or *c* followed by zero or more of any character.
`/finally.$`	Searches for the next occurrence of a line that ends with the string *finally* followed by any character. A line that ends with *finally* matches the pattern as well as a line that ends with *finallyA* or *finallyZ*.

© 2003 Hewlett-Packard Development Company, L.P.

14–18. SLIDE: Global Search and Replace — ex Commands

Global Search and Replace – ex Commands

```
:m,ns/old_pattern/new_text/
:1,$s/one/two/
```

From line 1 to the end of the file ($), substitute only the first occurrence found in each line of the text pattern "one" with the text string "two".

```
:m,ns/old_pattern/new_text/g
:.,10s/[oO][nN][eE]/two/g
```

From the current line through line 10, substitute every occurrence of the text pattern "one" in any case with the text string "two", globally within each line.

Student Notes

A global search and replace feature in `vi` is available through the `ex` commands. `ex` is a line-oriented editor that will accept the addresses of lines to operate upon within a file. The following global substitute and replace operations show the different components of the `ex` command syntax:

```
:m,ns/old_pattern/new_text/g
```

m and *n*	defines the lines the command should be executed on
`s`	designates the **substitute** command
old pattern	identifies the text *pattern* to search for
new pattern	designates the replacement text *string*
`g`	performs the command **globally** within the line

Example

```
:1,$s/one/two/
```

Substitute *just the first occurrence* in each line of the string *one* with the string *two* on lines 1 through the end of the file (`1,$`).

© 2003 Hewlett-Packard Development Company, L.P.

```
:.,10s/[oO][nN][eE]/two/g
```

Substitute every occurrence of the string *one*, including uppercase and lowercase combinations with the string *two* from the current line through line 10, globally within each line.

© 2003 Hewlett-Packard Development Company, L.P.

14–19. SLIDE: Some More ex Commands

`:w`	write the current buffer to disk
`:m,nw file`	write lines *m* through *n* of the current buffer to *file*
`:w file`	write the current buffer to *file*
`:e file`	bring *file* into the edit buffer discarding the old buffer
`:e!`	discard *all* changes to the buffer reload the file from the disk
`:r file`	read in the contents of *file* after the current cursor location
`:! cmd`	execute the shell command, `cmd`
`:set all`	show all edit session options
`:set nu`	turn on line numbering option

Student Notes

There are many options available which can make your editing with `vi` easier. These options are set with **ex** commands. The syntax to turn an option *on* is

```
:set option
```

The syntax to turn an option *off* is

```
:set nooption
```

A list of all current options and their settings can be displayed with

```
:set all
```


© 2003 Hewlett-Packard Development Company, L.P.

Many of the common options are presented on the slide. Some additional options include the following:

`:set autoindent`	while in input mode, the cursor will return to the column aligned with the indentation of the previous line, override with Ctrl + **d**
`:set tabstop= n`	assigns the Tab key to move the cursor ***n*** spaces
`:set wrapmargin = n`	automatic word break and newline ***n*** characters from end of line
`:set showmatch`	displays the matching opening brace (`{`, `(`, `[`) when the closing brace (`}`,`)`,`]`) is entered
`:set redraw`	**`vi`** will redraw the screen after each screen update. If you are using a slow baud rate (with a modem), you may want this option turned off. You can force a screen redraw with the **`vi`** command Ctrl + **`r`**.
`:set showmode`	turn on mode messages
`:map`	display keyboard mappings used in command mode
`:map!`	display keyboard mappings used in input mode

If you want any of these options be set automatically every time you enter `vi`, you must create a file called **`.exrc`** in your **`HOME`** directory and type the following lines:

`set option` *(note that there is no colon)*

© 2003 Hewlett-Packard Development Company, L.P.

14–20. TEXT PAGE: vi Commands — Summary

Cursor Control	Input Mode	Delete Text	Change Text	Write and Quit
h or Backspace	a	x	r	:wq
j	I	dw	R	ZZ
k	o	dd	cw	:q!
l or Space	O	dG	cc	:w *file*
w	A	d$	c$	:m,n w *file*
b	I			:e!
e				:e *file*
$				
^				
G				
#G		**Copy Text**	**Paste Text**	**Miscellaneous**
:#		yw	p (lowercase)	u
Ctrl + g		yy	P (uppercase)	U
Return		y$		.
Ctrl + b				/ *text*
Ctrl + f				n
Ctrl + u				:!cmd
Ctrl + d				
L				
M				
H				
Ctrl + l				

© 2003 Hewlett-Packard Development Company, L.P.

14–21. LAB: Modifying Text

Directions

Use the `vi` editor to complete the following exercises:

1. Start a vi session on the file `vi.tst`, and make the modifications as directed in that file. Following is a copy of the contents of `vi.tst`.

```
Enter your name here ->

Change the following to your favorite color -> lavender
Change the following to your favorite flower -> rose
Change the following to your favorite book -> A Tale of Two Cities

Correct the typos in the next two lines:
Corect teh typooos im thiss line.
Ther awe mroe mistakkes in thsi linne.

The above two lines should read:
Correct the typos in this line.
There are more mistakes in this line.

Delete every occurrence of the word "jog" in the next line:
walk jog run walk jog run walk jog run walk jog run

Change every occurrence of the word "walk" to "WALK" in the above line.

line1
line2
line3
line4
line5
line6
line7
line8

Complete the following exercises on line1 through line8 above:

1. Move the lines containing line1 through line5 and paste them after
the line containing line8.

2. Copy the lines containing line2 through line4 and paste them before
the line containing line6, and also after the line containing line3.

Quit your edit session on "vi.tst" saving the changes that you have
made.
```

2. Start a new session by editing the file called `funfile` in your HOME directory and change all occurrences of *bug* to *FEATURE*.

© 2003 Hewlett-Packard Development Company, L.P.

3. Write the first forty lines of the **funfile** out to another file called `new.40`.

4. Go to the last line in **funfile**.

5. Find and execute the command to place your cursor midway down the window. Insert the following line:

   ```
   This file is silly.
   ```

6. Without quitting **vi**, write your new version of the file out to a file called **funfile.123**.

7. Without leaving **vi**, load the file `new.40` into the buffer, overwriting the previous contents.

8. Turn on line numbering with the **ex** *number* option.

© 2003 Hewlett-Packard Development Company, L.P.

9. Search for an occurrence of `FEATURE` in `new.40`.

10. Change all occurrences of *FEATURE* to *BUG* and save it into `new.new.40`.

11. Copy `funfile` to `funfile.new`. In `funfile.new`, search for all occurrences of the string *System* or *system* and using `/`, `cw`, `n`, and `.` change all but one of them to *XXXXX*.

12. Write your current edit session and quit the editor.

© 2003 Hewlett-Packard Development Company, L.P.

Module 15 — Process Control

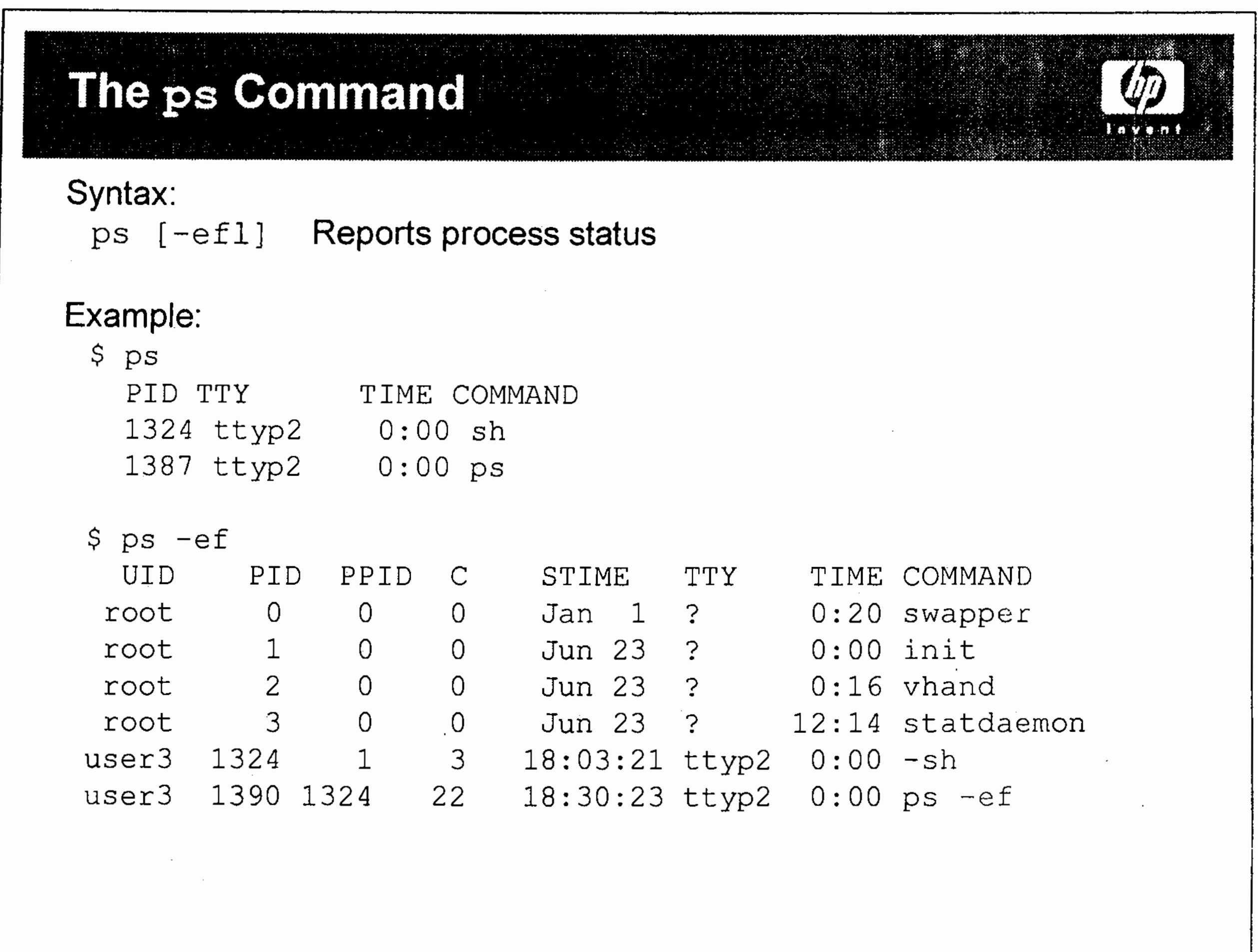
Objectives

Upon completion of this module, you will be able to do the following:

- Use the **ps** command.
- Start a process running in the background.
- Monitor the running processes with the **ps** command.
- Start a background process that is immune to the hangup (log off) signal.
- Bring a process to the foreground from the background.
- Suspend a process.
- Stop processes from running by sending them signals.

© 2003 Hewlett-Packard Development Company, L.P.

15–1. SLIDE: The ps Command



Student Notes

Every process that is initiated on the system is assigned a unique identification number, known as a process ID (**PID**). The **ps** command displays information about processes currently running (or sleeping) on your system, including the PID of each process and the PID of each process's parent (**PPID**). Through the PID and PPID numbers, you can trace the lineage of any process that is running on your system. The **ps** command will also report who owns each process and which terminal each process is executing through.

The ps command is commonly invoked with no options, which gives a short report about processes associated only with your terminal session. The **-e** option reports about every process running on the system, not just your own. The **-f** and **-l** options report full and long listings, which include additional detail on the processes.

In this slide we show two invocations of **ps**. The first just reports information about processes associated with our terminal. As we would expect, the processes associated with our terminal consist of a shell (our login shell) and the **ps** command that is currently running.

The second example shows a portion of the output of **ps** giving a full (**-f** option) listing of *every* (**-e** option) process on the system.

© 2003 Hewlett-Packard Development Company, L.P.

NOTE:	Be aware that the **ps** command is CPU intensive, and you may notice a slower response while it is executing.

© 2003 Hewlett-Packard Development Company, L.P.

15–2. SLIDE: Background Processing

Background Processing

Syntax:

```
command line > cmd.out &
```

Example:

```
$ grep user * > grep.out &
[1] 194

$ ps
  PID  TTY    TIME  COMMAND
  164  ttyp2  0:00  sh
  194  ttyp2  0:00  grep
  195  ttyp2  0:00  ps
```

Student Notes

The command line

`command line > cmd.out &`

- Schedules ***command line*** to run as a job in the background.
- Prompt returns as soon as job is initiated.
- Redirect output of scheduled command, so command output does not interfere with interactive commands.
- Logging out will terminate processes running in the background. The user will get a warning the first time exit is attempted: "There are running jobs". **exit** or Ctrl + d must be typed again to effectively terminate the session.

Some commands take a long time to complete, such as searching for a single file throughout the entire disk or using one of the text-processing utilities to format and print a manual transcript. The UNIX operating system allows you to start a time consuming program and run it in the background where the UNIX system will take care of continuing the execution of your program. Unlike other commands you have executed up to this point, the shell *does not*

© 2003 Hewlett-Packard Development Company, L.P.

wait for the completion of commands requested to run in the background. You will get your prompt back as soon as the command has been scheduled, allowing you to continue with other activities.

To request a command to run in the background, terminate the command line with an ampersand (`&`). It is common to redirect the output of the background command, so that output generated by background processes does not interfere with your interactive terminal session. If the output is not redirected, any output that normally goes to standard output from the command running in the background will be sent to your terminal.

Since the shell will have control over standard input, commands that are running in the background are not able to accept input from standard input. Therefore, any commands running in the background that require standard input must get their input from a file using input redirection.

When a command is put into the background, the shell reports the job number and process ID number of the background command, if the `monitor` option is set (`set -o monitor`). The job number identifies the number of the requested job relative to your terminal session, and the process ID identifies the system-wide unique process identifier that is assigned by the UNIX system to every process that is executed. The `monitor` option will also cause a message to be displayed when the background process is completed.

```
[1]+ Done grep user * > grep.out &
```

Since a command that is running in the background is disconnected from the keyboard, you cannot stop a background command with the interrupt key, Ctrl + `c`. Background commands can be terminated with the `kill` command or by logging out.

NOTE: A background process should have *all* of its input and output explicitly redirected.

NOTE: A background job may consist of multiple commands. Simply put the commands in parentheses (*`cmd1; cmd2; cmd3`*) and the operating system will treat them as one job.

© 2003 Hewlett-Packard Development Company, L.P.

15–3. SLIDE: Putting Jobs in Background/Foreground

Putting Jobs in Background/Foreground

`jobs`	Displays jobs currently running
Ctrl + z	Suspends a job running in the foreground `stty susp ^Z`
`fg [%number]` `fg [%string]`	Brings job number to the foreground or any job whose command line begins with string.
`bg [%number]` `bg [%string]`	Transfers job number to the background or any job whose command line begins with string.

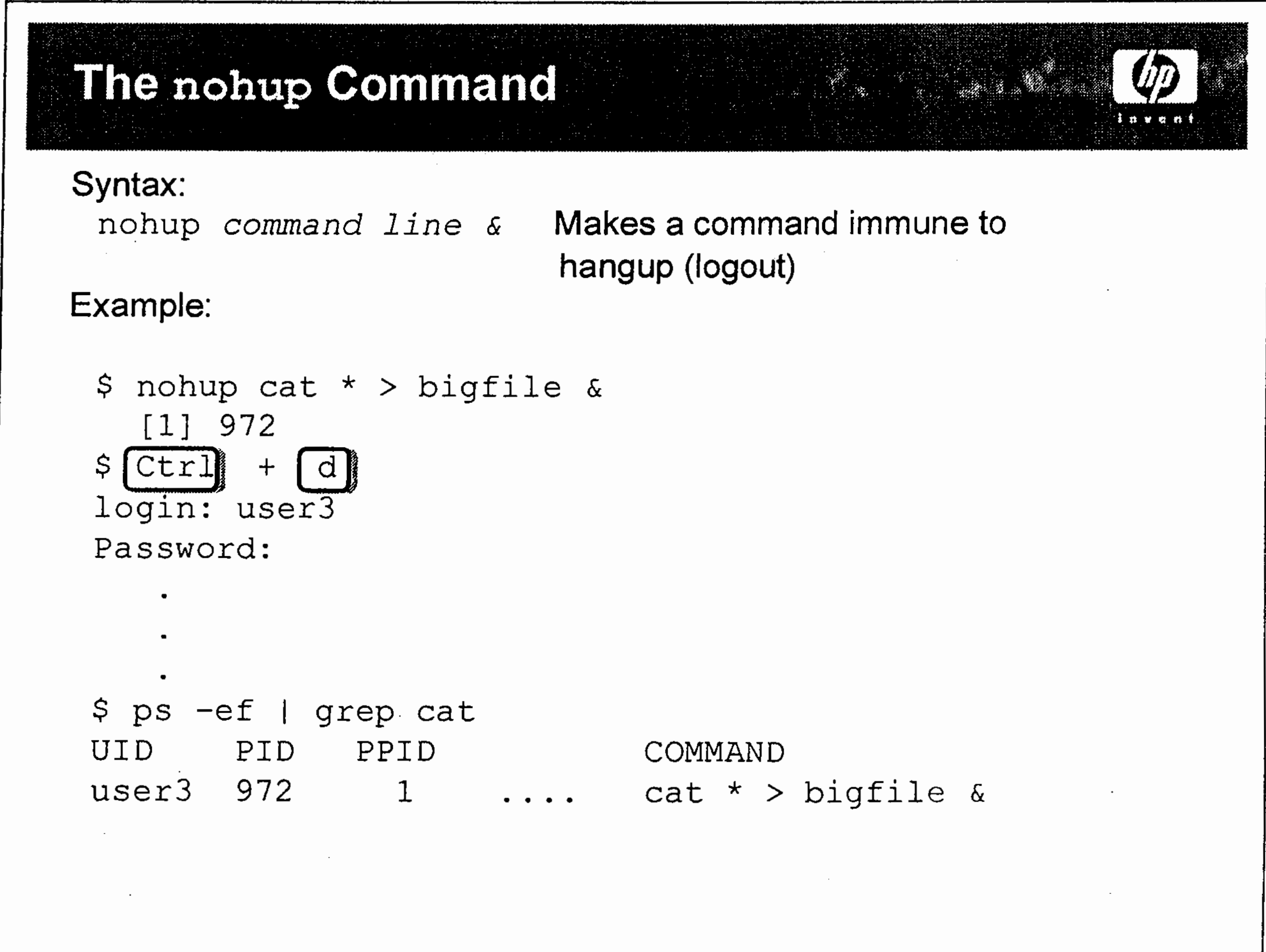
Student Notes

In the POSIX shell, processes can be placed in the foreground or the background. If you are currently running a lengthy process in the foreground, you can issue the *susp* character, which is usually set to Ctrl + **z**. The suspend character is commonly designated at login through **.profile**, with the entry, **stty susp ^Z**. This will temporarily stop your foreground process and provide a shell prompt. You can then use the **bg** ***%num*** or the **bg** ***%string*** to transfer your job to the background. ***num*** is the job number returned from the **jobs** command, and ***string*** is the beginning of the command line of the job.

Likewise, if you have a process running in the background that you would like to bring to the foreground, you can use the **fg** command. The foreground command will then control your terminal until it is completed or suspended.

© 2003 Hewlett-Packard Development Company, L.P.

15–4. SLIDE: The nohup Command



Student Notes

The UNIX operating system provides the **nohup** command to make commands immune to hanging up and logging off. The **nohup** command is one of a group of commands in the UNIX system known as **prefix commands**, which precede another command. It is most often used in conjunction with commands that you want to run in the background. Remember that logging out usually terminates background jobs. When a background command is **nohup**'ed, you can log out and the UNIX system will complete the execution of your process even though the program's parent shell is no longer running. Notice that when the parent shell of the **nohup** command is terminated, the command will be adopted by process 1 (**init**). You can later log in and view the status or results of the **nohup** command.

When using **nohup**, the user will normally redirect the output to a file. If the user *does not* specify an output file, **nohup** will automatically redirect the output to a file called **nohup.out**. Note that **nohup.out** will accumulate both **stdout** and **stderr**.

© 2003 Hewlett-Packard Development Company, L.P.

Linux Systems

On a Linux system, any process that is executed as a background process will be treated as though the **nohup** command had been applied. This makes the use of the **nohup** command on a command line redundant in Linux environments.

© 2003 Hewlett-Packard Development Company, L.P.

15–5. SLIDE: The `nice` Command

The `nice` Command

Syntax:

`nice [-N] command_line`	Runs a process at a lower priority `N` is a number between 1 and 19.

Example:

```
$ nice -10 cc myprog.c -o myprog
$ nice -5 sort * > sort.out &
$
```

Student Notes

The UNIX operating system is a time-sharing system, and process priorities are the basis for determining how often a program will have access to the system's resources. Jobs with lower priorities will have less frequent access to the system than jobs with higher priorities. For example, your terminal session has a relatively high priority to guarantee a prompt, interactive response.

The **`nice`** command is another prefix command that allows you to execute a program at a lower priority. It is useful when issuing commands whose completion is not required immediately, such as formatting the entire collection of manual pages.

The syntax is

```
nice [-increment] command line
```

Where *increment* is an integer value between one and nineteen. The default increment is 10. A process with a higher **`nice`** value will have a lower relative system priority. The **`nice`** value is *not* an absolute priority modifier.

© 2003 Hewlett-Packard Development Company, L.P.

You can view process priorities with the `ps -l` command. The priorities are displayed under the column headed PRI. Jobs that have a higher priority will have a *lower* priority value. The `nice` value is displayed under the column headed NI.

Most systems are started up with a default `nice` value of 20 for foreground processes, and 24 for background processes. The maximum value is 39, so the maximum increments are 19 and 15. Greater increments will not cause the value to rise above 39. Only `root` can use negative increments.

© 2003 Hewlett-Packard Development Company, L.P.

15–6. SLIDE: The kill Command

The kill Command

Syntax:

```
kill [-s signal_name] PID [PID...]
```

Sends a signal to specified processes.

Example:

```
$ cat /usr/share/man/cat1/* > bigfile1 &
   [1] 995
$ cat /usr/share/man/cat2/* > bigfile2 &
   [2] 996
$ kill 995
[1] - Terminated    cat /usr/share/man/cat1/* > bigfile1 &
$ kill -s INT %2
[2] + Interrupt     cat /usr/share/man/cat2/* > bigfile2 &
$ kill -s KILL 0
```

Student Notes

The **kill** command can be used to terminate any command including **nohup** and background commands. More specifically, **kill** sends a signal to a process. The default action for a process is to die when most signals are received. The issuer must be the owner of the target commands; **kill** cannot be used to kill another user's commands unless the superuser issues it.

In the UNIX system, it is not possible to actually kill a process. The most the UNIX system will do is request that a process terminate itself. By default, kill sends the **TERM** signal (software termination signal) to the specified processes. This normally kills processes that do not catch or ignore the signal. Other signals, listed in the table below, can be specified using the **-s** option. The closest thing to a sure kill that a UNIX system provides is the **KILL** signal (kill signal).

To kill a process, you can specify the process ID or the job number. When specifying the job number, it must be prefixed with the **%** metacharacter. If the process specified is 0, then **kill** terminates all processes associated with the current shell, *including* the current shell.

© 2003 Hewlett-Packard Development Company, L.P.

Signal name	Signal meaning
EXIT	Null signal
HUP	Hang up signal
INT	Interrupt
QUIT	Quit
ILL	Illegal instruction (not reset when caught)
TRAP	Trace trap (not reset when caught)
ABRT	Process abort signal
EMT	EMT instruction
FPE	Floating point exception
KILL	Kill (cannot be caught or ignored)
BUS	Bus error
SEGV	Segmentation violation
SYS	Bad argument to system call
PIPE	Write on a pipe with no one to read it
ALRM	Alarm clock
TERM	Software termination signal from kill
USR1	User-defined signal 1
USR2	User-defined signal 2
CHLD	Child process terminated or stopped
PWR	Power state indication
VTALRM	Virtual timer alarm
PROF	Profiling timer alarm
IO	Asynchronous I/O signal
WINCH	Window size change signal
STOP	Stop signal (cannot be caught or ignored)
TSTP	Interactive stop signal
CONT	Continue if stopped
TTIN	Read from control terminal attempted by a member of a background process group
TTOU	Write to control terminal attempted by a member of a background process group
URG	Urgent condition on I/O channel
LOST	Remote lock lost (NFS)

NOTE: The command **`kill -l`** will write all values of *signal_name* supported by the implementation. No signals are sent with this option. When the **`-l`** option is specified, the symbolic name of each signal is written to the standard output:

```
$ kill -l
HUP INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV SYS PIPE
ALRM TERM USR1 USR2 CHLD PWR VTALRM PROF IO WINCH STOP
TSTP CONT TTIN TTOU URG LOST
```

There are some differences in the signal values used in Linux, but the major values of 1, 2, 3, 9, and 15 are identical to the values as used in HP-UX.

© 2003 Hewlett-Packard Development Company, L.P.

15–7. LAB: Process Control

Directions

Complete the following exercises and answer the associated questions.

1. Under your *HOME* directory you will find a program called **infinite**. Execute this program in the foreground and notice what it does. Enter a Ctrl + **c** to terminate the program.

```
$ infinite
hello world
hello world
hello world
Ctrl + c
$
```

2. Run **infinite** in the background and redirect its output to a file called **infin.out**

```
$ infinite > infin.out &
```

Execute the **ps -f** command. Take note of the PID and PPID of the **infinite** program. Now log out, log in again, and execute the **ps -ef | grep *user_id***, where *user_id* is your login identifier. Where is the **infinite** process? Remove **infin.out** before the next exercise.

3. The **nohup** command protects a process from terminating upon the death of its parent process. Rerun the **infinite** command in the background, but protect it from logging out by issuing it with **nohup**.

```
$ nohup infinite > infin.out &
```

Now log out and log in again. Execute the **ps -ef | grep *user_id*** again. Is **infinite** still running? Who is its parent now?

© 2003 Hewlett-Packard Development Company, L.P.

4. Use the `kill` command to terminate your `infinite` program.

5. Run the `infinite` program in the *foreground* and redirect its output to `infin.out`. Suspend the program by issuing Ctrl + `z`. You will see a message on the screen telling you that the process has been stopped. Send `infinite` to the background, and note the message. Terminate the `infinite` program with the `kill` command.

© 2003 Hewlett-Packard Development Company, L.P.

Module 16 — Introduction to Shell Programming

Objectives

Upon completion of this module, you will be able to do the following:

- Write basic shell programs.
- Pass arguments to shell programs through environment variables.
- Pass arguments to shell programs through the positional parameters.
- Use the special shell variables, *, and #.
- Use the **`shift`** and **`read`** commands.

© 2003 Hewlett-Packard Development Company, L.P.

16–1. SLIDE: Shell Programming Overview

Shell Programming Overview

- A shell program is a regular file containing UNIX system commands.
- The file's permissions must be at least "read" and "execute."
- To execute, type the name of the file at the shell prompt.
- Data can be passed into a shell program through
  - environment variables
  - command line arguments
  - user input

Student Notes

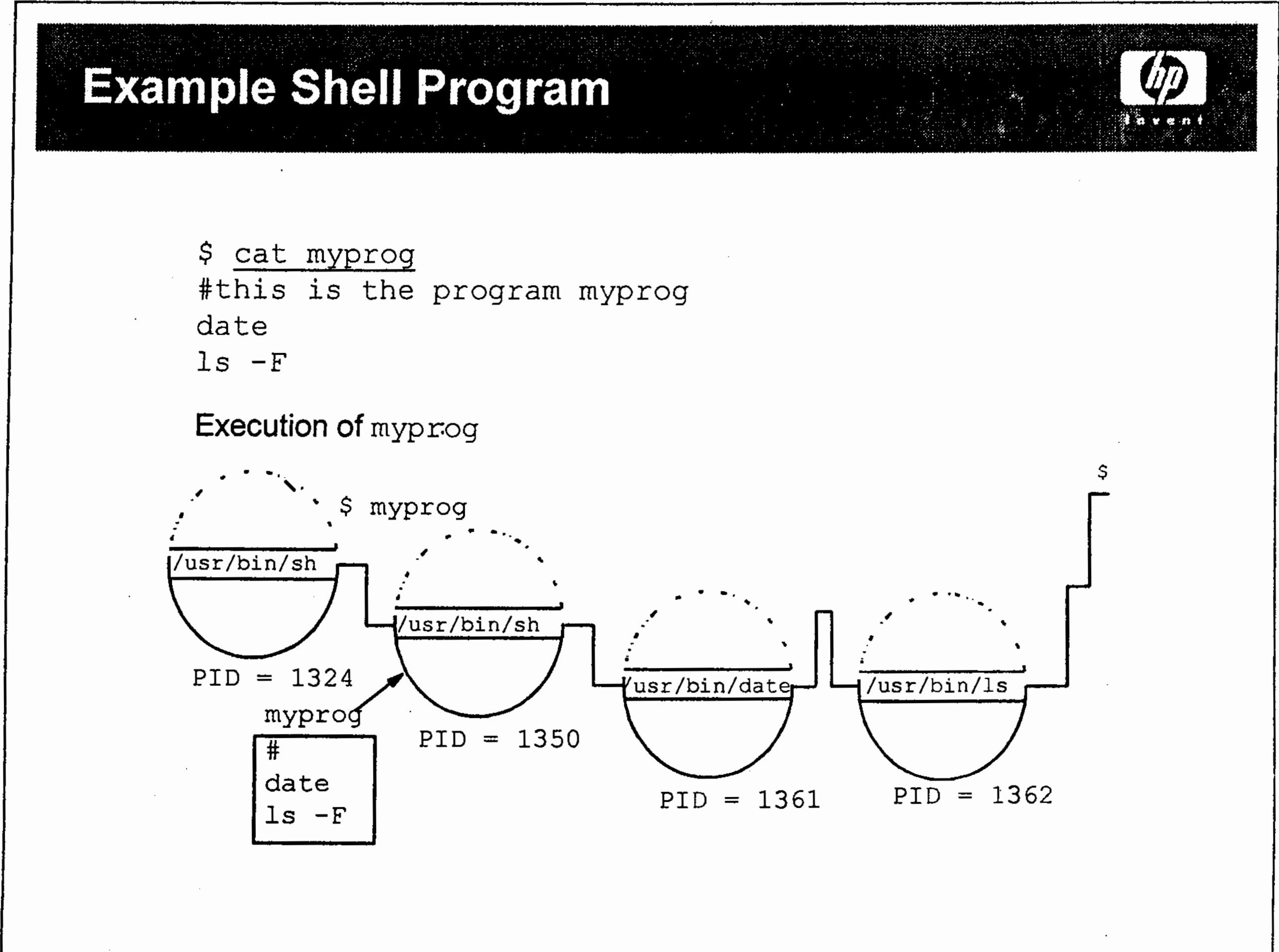
The shell is a command interpreter. It interprets the commands that you enter at the shell prompt. However, you can have a group of shell commands that you wish to enter many times. The shell provides the capability to store these commands in a file and execute this file just like any other program provided with your UNIX system. This command file is known as a **shell program** or a **shell script**. When running the program, it will execute just as if the commands were entered interactively at the shell prompt.

In order for the shell to access your shell program for execution, the shell must be able to read the program file and execute each line. Therefore, the shell program's permissions must be set to read and execute. So that the shell can find your program, you can enter the complete path of the program, or the program must reside in one of the directories designated in your *PATH* variable. Many users will create a `bin` directory under their *HOME* directory to store scripts that they have developed and include `$HOME/bin` in their *PATH* variable.

Rather complex shell scripts can be developed because the shell supports variables, command line arguments, interactive input, tests, branches, and loops.

© 2003 Hewlett-Packard Development Company, L.P.

16–2. SLIDE: Example Shell Program



Student Notes

To create and run a shell program, consider the following:

```
$ vi myprog                          A file containing shell commands
# this is the program myprog
date
ls -F
$ chmod +x myprog                    File mode includes execution
$ myprog                             Enter file name to execute program
Thu Jul 11 11:10 EDT 1994

f1 f2 memo/ myprog*
```

First the shell program **myprog** is created using a text editor. Before the program can be run, the program file must be given execute permission. Then the program name can be typed at the shell prompt. As seen on the slide, when **myprog** is executed, a child shell process is created. This child shell reads its input from the shell program file **myprog** instead of from the command line. Each command in the shell program is executed, in turn, by the child shell. Once all of the commands have been executed, the child shell terminates and returns control to the original parent shell.

© 2003 Hewlett-Packard Development Company, L.P.

Comments in a Shell Program

It is recommended that you provide comments in your shell program that identify and clarify the contents of the program. Comments are preceded by a # symbol. The shell will not attempt to execute anything that follows the #, which can appear anywhere in the command line.

NOTE: You should never call a shell program `test`, because test is a built-in shell command.

© 2003 Hewlett-Packard Development Company, L.P.

16–3. SLIDE: Passing Data to a Shell Program

Passing Data to a Shell Program

```
$ color=lavender

$ cat color1
echo You are now running program: color1
echo the value of the variable color is: $color

$ chmod +x color1

$ color1
You are now running program: color1
the value of the variable color is:

$ export color
$ color1
You are now running program: color1
the value of the variable color is: lavender
```

Student Notes

One way to pass data to a shell program is through the environment. In the example on the slide, the local variable *color* is assigned the value *lavender*. Then the shell program `color1` is created; its permissions are changed to include execute permission; it is then executed. `color1` attempts to echo the value of the variable *color*. However, since `color` is a local variable that is private to the parent shell, the child shell running `color1` does not recognize the variable, and can therefore not print its value. When *color* is exported into the environment, it is then accessible to the shell program commands running in the child shell.

Also, since a child process cannot change the environment of its parent process, reassigning the value of an environment variable in a child shell will not affect the value of that variable in the parent's environment. Consider the following shell script, `color2`, which is found in your *HOME* directory:

```
echo The original value of the variable color is $color
echo This program will set the value of color to amber
color=amber
echo The value of color is now $color
echo When your program concludes, display the value of the color variable.
```


© 2003 Hewlett-Packard Development Company, L.P.

Observe what happens when we set the value of color, export it, and then execute **`color2`**:

```
$ export color=lavender
$ echo $color
lavender
$ color2
The original value of the variable color is lavender
This program will set the value of color to amber
The value of color is now amber
When your program concludes, display the value of the color variable.
$ echo $color
lavender
```


© 2003 Hewlett-Packard Development Company, L.P.

16–4. SLIDE: Arguments to Shell Programs

Arguments to Shell Programs

Command line:

```
$ sh_program arg1 arg2 . . . argX
      $0       $1   $2  . . .  $X
```

Example:

```
$ cat color3
echo You are now running program: $0
echo The value of command line argument \#1 is: $1
echo The value of command line argument \#2 is: $2

$ chmod +x color3

$ color3 red green
You are now running program: color3
The value of command line argument #1 is: red
The value of command line argument #2 is: green
```

Student Notes

Most UNIX system commands accept command-line arguments, which often inform the command about files or directories upon which the command should operate (**`cp f1 f2`**), specify options that extend the capabilities of the command (`ls -l`), or just supply text strings (**`banner hi there`**).

Command-line argument support is also available for shell programs. They are a convenient mechanism to pass information into your utility. When you develop your program to accept command-line arguments, you can pass file or directory names that you want your utility to manipulate, just as you do with the UNIX system commands. You can also define command line options that will allow command-line access to extend capabilities of your shell program.

The arguments on the command line are referenced within your shell program through special variables that are defined relative to an argument's position in the command line. Such arguments are called **positional parameters** because the assignment of each special variable depends on an argument's position in the command line. The names of these variables correspond to their numeric position on the command line, thus the special variable names are the numbers `0,1,2`, and so on, up through the last parameter passed. The values of these variables are accessed in the same way as any other variable's value is accessed — by prefixing the name with the **$** symbol. Therefore, to access the command line arguments in

© 2003 Hewlett-Packard Development Company, L.P.

your shell program, you would reference `$0,$1,$2`, and so on. After `$9`, the curly brace notation must be used: `${10}, ${24}`, and so on, otherwise the shell would think `$10` was `$1` with a 0 (zero) appended to it. `$0` will *always* hold the program or command name.

The only disadvantage to developing a program that accepts command-line arguments is that the users must know the proper syntax and what the command-line arguments represent. For example, how do you know that the **cp** command can copy one file to another file or several files to a directory? What happens when you type the command in and provide three file names as arguments: **cp f1 f2 f3**? You have a UNIX system reference manual that provides you with the proper syntax, and the UNIX system will supply a usage message if you have not typed the command in properly (try entering **cp** Return). You will need to supply similar usage aids to any other users that you will expect to utilize the programs that you develop.

© 2003 Hewlett-Packard Development Company, L.P.

16–5. SLIDE: Arguments to Shell Programs (Continued)

Arguments to Shell Programs (Continued)

This shell program will install a program, specified as a command-line argument to your bin directory: First create the program my_install. Note that `$HOME/bin` should exist!

```
$ cat > my_install
echo $0 will install $1 to your bin directory
chmod +x $1
mv $1 $HOME/bin
echo Installation of $1 is complete
[Ctrl] + [d]
$ chmod +x my_install

$ my_install color3
my_install will install color3 to your bin directory
Installation of color3 is complete
$
```

Student Notes

This example demonstrates a program that has designated the first command-line argument to be the name of a file, which will be made executable and then moved to the bin directory under your current directory.

Remember the UNIX system convention to store programs under a directory called **`bin`**. You may want to create a **`bin`** directory under your *HOME* directory where your shell programs can be stored. Remember to append your **`bin`** directory to the *PATH* variable so that the shell can find your programs.

© 2003 Hewlett-Packard Development Company, L.P.

16–6. SLIDE: Some Special Shell Variables — # and *

Some Special Shell Variables — # and *

#	The number of command line arguments
*	The entire argument string

Example:

```
$ cat color4
echo There are $# command line arguments
echo They are $*
echo The first command line argument is $1

$ chmod +x color4

$ color4 red green yellow blue
There are 4 command line arguments
They are red green yellow blue
The first command line argument is red
$
```

Student Notes

The shell programs we've seen so far have not been very flexible. `color3` expected exactly two arguments, and `my_install` expected only one argument. Often when you create a shell program that accepts command-line arguments, you would like to allow the user to type in a variable number of arguments. You would like the program to execute successfully if the user types in 1 argument or 20 arguments.

The special shell variables **#** and ***** will provide you with a lot of flexibility when dealing with a variable argument list. You will always know how many arguments have been entered through **$#**, and you can always access the entire argument list through **$***, regardless of the number of arguments. Notice that the command (**$0**) is never included in the argument list variable **$***.

Each command-line argument will still maintain its individual identity as well. So you can reference them collectively through **$*** or individually through **$1, $2, $3**, and so on.

© 2003 Hewlett-Packard Development Company, L.P.

16–7. SLIDE: Some Special Shell Variables — # and * (Continued)

Some Special Shell Variables — # and * (Continued)

This enhanced example of the install program accepts multiple command-line arguments:

```
$ cat > my_install2
echo $0 will install $# files to your bin directory
echo The files to be installed are: $*
chmod +x $*
mv $* $HOME/bin
echo Installation is complete
Ctrl + d
$ chmod +x my_install2

$ my_install2 color1 color2
my_install2 will install 2 files to your bin directory
The files to be installed are: color1 color2
Installation is complete
```

Student Notes

The installation program is now more flexible. If you have several scripts that need to be installed, you have to execute the program only once and supply all of the names on the command line. It is important to note that if you plan to pass the entire argument string to a command, it must be able to accept multiple arguments.

Consider the following script, in which the user provides a directory name as a command line argument. The program will change to the designated directory, display its current position, and then list the contents:

```
$ cat list_dir
cd $*
echo You are in the $(pwd) directory
echo The contents of this directory are:
ls -F
$ list_dir dir1 dir2 dir3
sh: cd: bad argument count
```

Since the cd command cannot change into more than one directory, the program will incur an error.

© 2003 Hewlett-Packard Development Company, L.P.

16–8. SLIDE: The `shift` Command

The `shift` Command

Shifts all strings in * left n positions
Decrements # by n (default value of n is 1)

Syntax: `shift [n]`

Example:

```
$ cat color5
orig_args=$*
echo There are $# command line arguments
echo They are $*
echo Shifting two arguments
shift 2
echo There are $# command line arguments
echo They are $*
echo Shifting two arguments
shift 2; final_args=$*
echo Original arguments are: $orig_args
echo Final arguments are: $final_args
```

Student Notes

The **`shift`** command will reassign the command-line arguments to the positional parameters, allowing you to increment through the command-line arguments. After a **`shift`** ***n***, all parameters in * are moved to the left *n* positions, and # is decremented by *n*. The default for *n* is 1. The shift command does not affect the positional parameter `0`.

Once you have completed a shift, the arguments that have been shifted off of the command line are lost. If you will need to reference them later in your program, you will need to save them *before* you execute the **`shift`**.

The **`shift`** command is useful for:

- accessing positional parameters in groups, such as a series of *x* and *y* coordinates
- discarding command options from a command line, assuming that the options precede the arguments

© 2003 Hewlett-Packard Development Company, L.P.

Example

The following shows the output that would be generated if the shell program illustrated in the slide were executed:

```
$ color5 red green yellow blue orange black

There are 6 command line arguments
They are red green yellow blue orange black
Shifting two arguments
There are 4 command line arguments
They are yellow blue orange black
Shifting two arguments
Original arguments are: red green yellow blue orange black
Final arguments are: orange black
$
```


© 2003 Hewlett-Packard Development Company, L.P.

16–9. SLIDE: The `read` Command

The `read` Command

Syntax:

```
read variable [ variable ... ]
```

Example:

```
$ cat color6
echo This program prompts for user input
echo "Please enter your favorite two colors -> \c"
read color_a color_b
echo The colors you entered are: $color_b $color_a
$ chmod +x color6
$ color6
This program prompts for user input
Please enter your favorite two colors ->red blue
The colors you entered are: blue red
$ color6
This program prompts for user input
Please enter your favorite two colors ->red blue tan
The colors you entered are: blue tan red
```

Student Notes

Command-line arguments allow a user to pass information into a program when the program is invoked, and the user must know the correct syntax *before* the command is executed. There are situations, though, in which you would rather have the user execute just the program and then prompt him or her to provide input *during* the program execution. The **read** command is used to gather information typed at the terminal during the program execution.

You will usually want to provide a prompt to the user with the **echo** command so that he or she knows that the program is waiting for some input, and inform the user about what type of input is expected. Therefore, an **echo** statement should precede each **read** statement.

The **read** command will specify a list of variable names, whose values will be assigned to the words (delimited by white space) that the user supplies at the prompt. If there are more variables specified by the **read** command than there are words of input, the leftover variables are assigned to NULL. If the user provides more words than there are variables, all leftover data is assigned to the last variable in the list.

Once assigned, you can access these variables just as you can access any other shell variables.

© 2003 Hewlett-Packard Development Company, L.P.

NOTE:	Do not confuse positional parameters with variables read. Positional parameters are specified on the command line when you invoke a program. The `read` command assigns variable values through input provided during program execution in response to a programmed prompt.

`echo` and Escape Characters

There are several special escape characters that the `echo` command interprets that provide line control. Each escape character must be preceded by a backslash (\) and enclosed in quotes ("). These escape characters are interpreted by `echo`, *not* by the shell.

Character	Prints
`\a`	Alert character (equivalent to [Ctrl] + [g])
`\b`	Backspace
`\c`	Suppresses the terminating newline
`\f`	Formfeed.
`\n`	Newline
`\r`	Carriage return
`\t`	Tab character
`\\`	Backslash
nnn	The character whose ASCII value is *nnn*, where nnn is a one – to three-digit octal number that starts with zero.

© 2003 Hewlett-Packard Development Company, L.P.

16–10. SLIDE: The `read` Command (Continued)

The `read` Command (Continued)

This enhanced example of the install program prompts the user to input the file names to be installed:

```
$ cat > my_install3
echo $0 will install files into your bin directory
echo "Enter the names of the files -> \c"
read filenames
chmod +x $filenames
mv $filenames $HOME/bin
echo Installation is complete
Ctrl + d
$ chmod +x my_install3

$ my_install3
my_install3 will install files into your bin directory
Enter the names of the files -> f1 f2
Installation is complete
```

Student Notes

This version of the install routine will prompt the user for the file names to **`chmod`** and move to the **`$HOME/bin`** directory. This program gives the user a little more direction regarding what input is expected compared to **`install2`** in which the user must supply the file names on the command line. There is no special syntax the user must know to invoke this program. The program lets the user know exactly what it expects. All entered file names will be assigned to the variable *filenames*.

© 2003 Hewlett-Packard Development Company, L.P.

16–11. SLIDE: Additional Techniques

Additional Techniques

- Document shell programs by preceding a comment with number sign (#).
- `sh shell_program arguments`
  shell_program does not have to be executable.
  shell_program does have to be readable.
- `sh -x shell_program arguments`
  Each line of shell_program is printed before being executed. Useful for debugging your program.

Student Notes

The # character is used to provide comments in your shell program. The shell will ignore anything that follows the #, up to the Return. Comments inform others (and maybe you too) who are reading your program file of the purpose of the commands that are contained within the program.

An alternative way to execute a shell program is to use the following:

```
sh shell_program arguments
```

This invokes a subshell and designates that subshell as the command interpreter to use while executing the program. The program file *does not* have to be executable. This is useful if you are running under one shell and wish to execute a shell program written in another shell's command language.

You can also specify the command interpreter that should be used during the execution of a shell program by providing **#!/bin/*shell_name*** as the very first line of your shell program. Therefore, if you are currently running under the POSIX shell interactively, but have a C shell script that you would like to execute, the first line of the C shell program should be: **#!/bin/csh**.

© 2003 Hewlett-Packard Development Company, L.P.

Although there is no debugger for a shell program, the command

```
sh -x shell_program arguments
```

Will display each command in the shell program *before* executing it. This allows you to see how the shell is performing file name generation, variable substitution, and command substitution. This option is especially helpful for discovering typing errors.

© 2003 Hewlett-Packard Development Company, L.P.

16–12. LAB: Introduction to Shell Programming

Directions

Complete the following exercises and answer the associated questions.

1. Create a shell program called **whoson1** that will:
   - Display a greeting to the user with the **echo** command.
   - Define a variable *MYNAME* to your name.
   - Display the value of the *MYNAME* variable defined above.
   - Display the time and date.
   - Display all of the users who are logged in to the system.

2. Change to the **/tmp** directory. Invoke the program **color1**. Does the shell find the **color1** program?

3. Change to the **$HOME** directory. Create a directory named **bin** under your **HOME** directory. Move the **color1** program to your **bin** directory. Append your **bin** directory to the *PATH* variable so that the shell can find your **color1** program. Confirm that your *PATH* variable works by changing to the **/tmp** directory and invoking the **color1** program. Remember to define the color variable before invoking the **color1** program.

4. Change to **$HOME** directory. Interactively assign the output of the **date** command to a variable *date_var*. Create a shell program called **date_tst** that will display the value of this variable. Install **date_tst** under your **bin** directory.

© 2003 Hewlett-Packard Development Company, L.P.

5. Modify **date_tst** so that the value of the variable *date_var* is assigned when the program is executed. Does *date_var* need to be exported in this exercise? Do you need to change the permissions on **date_tst**?

6. Create a shell program called **whoson2** that will

   - Display a personalized greeting to the user with the **echo** command, such as *welcome username*, so that if user3 was logged in it would echo *welcome user3* or if user2 was logged in it would echo *welcome user2*. (Hint: this can be accomplished with an environment variable or command substitution.)
   - Display the system time and date.
   - Display all of the users who are logged into the system.
   - Display a message to the user displaying his or her ID and terminal connection.
   - Display a closing message before the program concludes.

   Place this program under your **bin** directory so that you can invoke it no matter where you are in your hierarchy.

7. If the command line for a shell program is:

```
$ myshellprog abc def -d -4 +900 xyz
```

   What will be printed out from the shell program if it contains the following?

```
echo $#
echo $3
echo $7
echo $*
echo $0
```


© 2003 Hewlett-Packard Development Company, L.P.

8. If the shell program invoked by the command line in the previous exercise contained a **shift 2** command as the first line, write the results of the following:

```
echo $#
echo $3
echo $7
echo $*
echo $0
```

9. What would be the output of the following shell program if, when prompted, the user typed in the following input?

```
James A.  Smith, Jr.
```

Shell program:

```
echo "Please type in your first, middle, and last names"
read first middle last
echo "$last, $first $middle"
```

10. Write a shell program named **search1** that prompts the user for a string to search for in all of the files in his or her current directory. Print the file names of all of the files that contain the string.

11. Write a shell program called **backwards** that will receive up to ten arguments and list the arguments in reverse order.

© 2003 Hewlett-Packard Development Company, L.P.

12. Write a shell program called **myecho** that will do the following:

    - Print the number of arguments passed to it.
    - Print the first three arguments on separate lines.
    - Print the remaining arguments on one line.

    Execute the program with 12 arguments.

    What argument list will produce the following output from this shell program?

```
I cannot
seem to
find my KEYS.
```

13. Create a program **my_vi** that will accept a command-line argument that designates a file to edit. **my_vi** should make a backup copy of the specified file and then start a **vi** session on the file. Use an extension like **.bak** when creating the backup file. At this point, only use file names of ten characters or less.

14. Create a companion program to **my_vi** called **my_recover** that will restore a file designated as a command-line argument from its backup file. Specify the file name without the **.bak** extension. For example if you want to restore the file **tst1** from **tst1.bak** you would execute **my_recover tst1**.

© 2003 Hewlett-Packard Development Company, L.P.

15. Write a shell program called `info` that will prompt the user for the following:

    - name
    - street address
    - city, state, and zip code

    The program should then store the replies in variables and display what the user entered with an informative format.

16. Write a shell program called **`home`** that prompts for any user's *login_id* and displays that user's **`HOME`** directory. Recall that the **`HOME`** directory is the sixth field in the **`/etc/passwd`** file. You should display the *login_id*'s from the **`/etc/passwd`** file in four columns so that the user knows what the available login IDs are.

17. Write a shell program called **`alpha`** that will display the first and last command line arguments. Hint: use the `cut` command.

18. Create a shell program called **`copy`** that will provide a user interface to the **`cp`** command. Your program should prompt the user for the names of the files that he or she wants copied, and then prompt the user for the destination of the copy. The destination should be a directory when copying multiple files, and the destination can be a file when copying only one file. Ring the bell when the program is completed.

© 2003 Hewlett-Packard Development Company, L.P.

© 2003 Hewlett-Packard Development Company, L.P.

Module 17 — Shell Programming — Branches

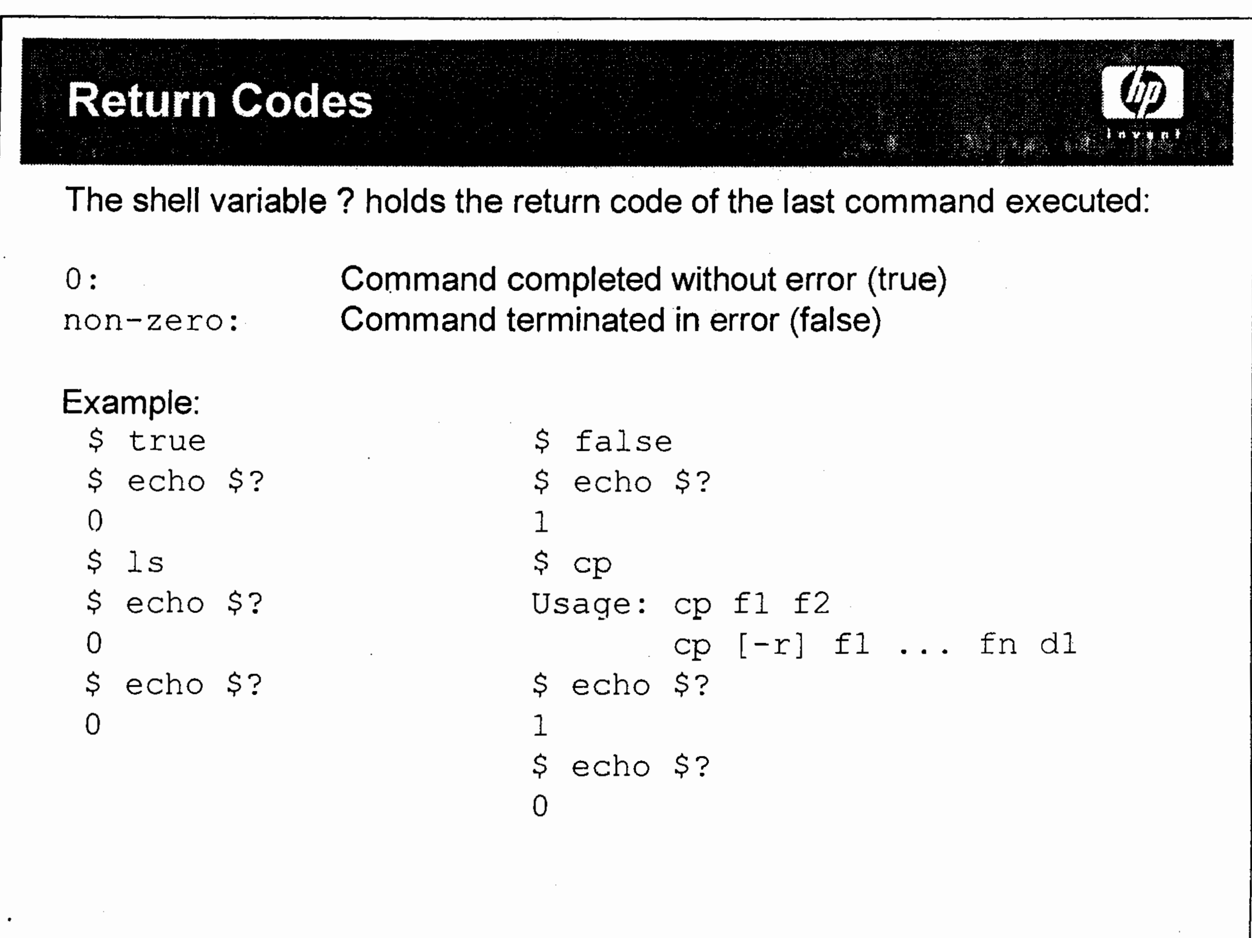
Objectives

Upon completion of this module, you will be able to do the following:

- Describe the use of return codes for conditional branching.
- Use the `test` command to analyze the return code of a command.
- Use the `if` and `case` constructs for branching in a shell program.

© 2003 Hewlett-Packard Development Company, L.P.

17–1. SLIDE: Return Codes



Student Notes

All UNIX operating system commands will generate a return code upon completion of the command. This return code is commonly used to determine whether a command completed normally (returning `0`) or encountered some error (returning non-zero). The non-zero return code often reflects the error that was generated. For example, syntax errors will commonly set the return code to `1`. The command **`true`** will always return `0` and the command **`false`** will always return `1`.

Most programming decisions will be controlled by analyzing the value of return codes. The shell defines a special variable `?` that will hold the value of the previous return code.

You can always display the return code of the *previous* command with

```
echo $?
```

When executing conditional tests (that is, less than, greater than, equality), the return code will denote whether the condition was true (return `0`) or false (returning non-zero). Conditional tests will be presented on the next several slides.

© 2003 Hewlett-Packard Development Company, L.P.

17–2. SLIDE: The `test` Command

The `test` Command

Syntax:
`test` *expression* or `[` *expression* `]`

The `test` command evaluates the expression, and sets the return code.

Expression Value	Return Code
true	0
false	non-zero (commonly 1)

The `test` command can evaluate the condition of

- Integers
- Strings
- Files

Student Notes

The **`test`** command is used to evaluate expressions and generate a return code. It takes arguments that form logical expressions and evaluates the expressions. *The* **`test`** *command writes nothing to standard output.* You must display the value of the return code to determine the result of the **`test`** command. The return code will be set to `0` if the expression evaluates to *true*, and the return code will be set to `1` if the expression evaluates to *false*.

The **`test`** command is initially presented alone so that you can display the return codes. But it is most commonly used with the **`if`** and **`while`** constructs to provide conditional flow control.

The test command can also be invoked as `[` ***`expression`*** `]`. This is intended to assist readability, especially when implementing numerical or string tests.

NOTE: There must be white space around `[` and `]`.

© 2003 Hewlett-Packard Development Company, L.P.

17–3. SLIDE: The `test` Command — Numeric Tests

The `test` Command — Numeric Tests

Syntax:

`[ number relation number ]` Compares numbers according to relation

Relations:

`-lt`	Less than
`-le`	Less than or equal to
`-gt`	Greater than
`-ge`	Greater than or equal to
`-eq`	Equal to
`-ne`	Not equal to

Example: (Assume X=3)

```
$ [ "$X" -lt 7 ]        $ [ "$X" -gt 7 ]
$ echo $?               $ echo $?
0                       1
```

Student Notes

The **`test`** command can be used to evaluate the numerical relationship between two integers. It is commonly invoked with the `[ ... ]` syntax. The return code of the **`test`** command will denote whether the condition was true (returning `0`) or false (returning `1`).

The numeric operators include

`-lt`	Is less than
`-le`	Is less than or equal to
`-gt`	Is greater than
`-ge`	Is greater than or equal to
`-eq`	Is equal to
`-ne`	Is not equal to

When testing the value of a shell variable, you should protect against the possibility that the variable may contain nothing. For example, look at the following test statement:

```
$ [ $XX -eq 3 ]
sh: test: argument expected
```


© 2003 Hewlett-Packard Development Company, L.P.

If *XX* has not been previously assigned a value, *XX* will be NULL. When the shell performs the variable substitution, the command that the shell will attempt to execute will be

[-eq 3] which is not a complete test statement and is guaranteed to cause a syntax error. A simple way around this is to quote the variable being tested.

```
[ "$XX" -eq 3 ]
```

When the shell performs the variable substitution, the command that the shell will attempt to execute will be

```
[ "" -eq 3 ]
```

This will ensure that the variable will contain at least a NULL *value* and will provide a satisfactory argument for the **`test`** command.

NOTE:	As a rule, you should surround all **`$variable`** expressions with double quotation marks to avoid improper variable substitution by the shell.

© 2003 Hewlett-Packard Development Company, L.P.

17–4. SLIDE: The `test` Command — String Tests

The `test` Command — String Tests

Syntax:

`[ string1 = string2 ]`	Determines string equivalence
`[ string1 != string2 ]`	Determines string nonequivalence

Example:

```
$ X=abc
$ [ "$X" = "abc" ]
$ echo $?
0
```

```
$ X=abc
$ [ "$X" != "abc" ]
$ echo $?
1
```

Student Notes

The **`test`** command can also be used to compare the equality or inequality of two strings. The `[ ... ]` syntax is commonly used for string comparisons. You have already seen that there must be white space surrounding the `[ ]`, and there must also be white space provided around the equivalence operator.

The string operators include the following:

string1* = *string2	True if ***string1*** and ***string2*** are identical.
string1* != *string2	True if ***string1*** and ***string2*** are not identical.
-z *string*	True if the length of ***string*** is zero.
-n *string*	True if the length of ***string*** is non-zero.
string	True if the length of ***string*** is non-zero.

© 2003 Hewlett-Packard Development Company, L.P.

Quotation marks will also protect the string evaluation if the value of the variable contains blanks. For example:

```
$ X="Yes we will"
$ [ $X = yes ]                    causes a syntax error
```

Interpreted by the shell as: **`[ Yes we will = yes ]`**

```
$ [ "$X" = yes ]                  proper syntax
```

Interpreted by the shell as: **`[ "Yes we will" = yes ]`** This will be evaluated correctly since the quotation marks surround the string.

Numerical versus String Comparison

The shell will treat all arguments as numbers when performing numerical tests, and all arguments as strings when performing string tests. This is best illustrated by the following example:

```
$ X=03
$ Y=3
$ [ "$X" -eq "$Y" ]               compares numeral 03 with numeral 3
$ echo $?
0                                 true — they are equivalent numerically
$ [ "$X" = "$Y" ]                 compares the string "03" with the string "3"
$ echo $?
1                                 false — they are not equivalent strings
```


© 2003 Hewlett-Packard Development Company, L.P.

17–5. SLIDE: SLIDE: The `test` Command — File Tests

The `test` Command — File Tests

Syntax:

`test -option filename` Evaluates filename according to option

Example:

```
$ test -f funfile
$ echo $?
0
$ test -d funfile
$ echo $?
1
```

Student Notes

A useful testing feature provided by the shell is the capability to test file characteristics such as file type and permissions. For example:

`test -f filename`

will return true (0) if the file exists and is a regular file (not directory or device).

`test -s filename`

will return true (0) if the file exists and has a size greater than 0.

There are many other file tests available. A partial list includes:

-r *file* True if the ***file*** exists and is readable.

-w *file* True if the ***file*** exists and is writable.

-x *file* True if the ***file*** exists and is executable.

© 2003 Hewlett-Packard Development Company, L.P.

`-d` *`directory`* True if *`directory`* exists and is a directory.

The tests on the slide could also be entered:

```
$ [ -f funfile ]

$ [ -d funfile ]
```

Refer to your HP-UX Reference Manual for additional options.

© 2003 Hewlett-Packard Development Company, L.P.

17–6. SLIDE: The `test` Command — Other Operators

The `test` Command — Other Operators

Syntax:

```
-o        OR
-a        AND
!         NOT
\(  \)    GROUPING*
```

Example:

```
$ [ "$ANS" = y -o "$ANS" = Y ]
$ [ "$NUM" -gt 10 -a "$NUM" -lt 20 ]
$ test -s file -a -r file
```

*Note: the ( ) must be escaped with a backslash.

Student Notes

You can use Boolean operators to test for multiple conditions.

Expr 1	Operator	Expr 2	Outcome
true	-o	true	true (0)
true	-o	false	true (0)
false	-o	true	true (0)
false	-o	false	false (1)
true	-a	true	true (0)
true	-a	false	false (1)
false	-a	true	false (1)
false	-a	false	false (1)

© 2003 Hewlett-Packard Development Company, L.P.

Examples

```
$ [ "$ANS" = y -o "$ANS" = Y ]
$ [ "$NUM" -gt 10 -a "$NUM" -lt 20 ]
$ test -s file -a -r file -a -x file
```

The NOT operator (`!`) is used in conjunction with the other operators and is most commonly used for file testing. There *must* be a space between the not operator and any other operators or arguments. For example,

```
test ! -d file
```

can be used instead of

```
test -f file -o -c file -o -b file ...
```

Parentheses can be used to group operators, but parentheses have another special meaning to the shell, which is interpreted first. Therefore, the parentheses must be escaped to delay their interpretation.

The following example is verifying that there are 2 command line arguments, AND that the first command line argument is a `-m`, AND that the last command line arguments is a *directory* OR a *file* whose size is greater than zero:

```
[ \( $# = 2 \) -a \( "$1" = "-m" \) -a \( -d "$2" -o -s "$2" \) ]
```


© 2003 Hewlett-Packard Development Company, L.P.

17–7. SLIDE: The `exit` Command

The `exit` Command

Syntax:
```
exit [arg]
```

Example:

```
$ cat exit_test
echo exiting program now
exit 99

$ exit_test
exiting program now

$ echo $?
99
```

Student Notes

The `exit` command will terminate the execution of a shell program and set the return code. It is normally set to zero to denote normal termination and to a non-zero value to denote an error condition. If no argument is provided, the return code is set to the return code of the last command executed prior to the `exit` command.

© 2003 Hewlett-Packard Development Company, L.P.

17–8. SLIDE: The `if` Construct

The `if` Construct

Syntax: (used for single-decision branch)

```
if
    list A
then
    list B
fi
```

Example:

```
if
   test -s funfile
then
   echo funfile exists
fi
echo hello
```

Student Notes

The **`if`** construct provides for program flow control based on the *return code* of a command. If the return code of a designated command is `0`, a specified command list will be executed. If the return code of the designated command is non-zero, the command list will be disregarded.

The slide shows the general format of the **`if`** construct including a flow chart and a simple example. Each command list is commonly one or more UNIX system shell commands separated by [Return] or semicolons. The decision for the **`if`** statement will be based on the *last* command executed in the *list A*, prior to the **`then`**.

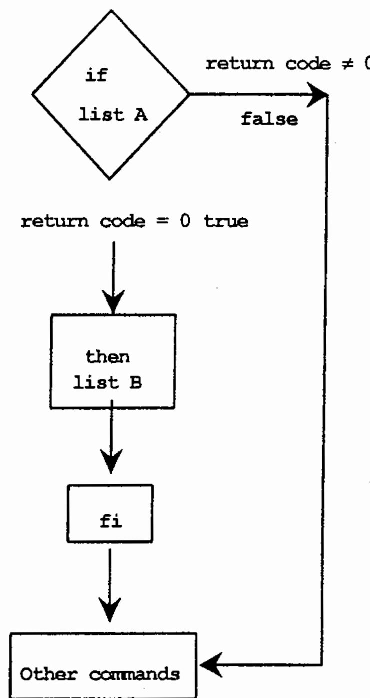
A summary of the execution of the **`if`** construct is as follows:

1. Command *list* A is executed.

2. If the return code of the *last* command in command *list A* is a `0` (TRUE), execute command *list B*, then continue with the first statement following the **`fi`**.

3. If the return code of the *last* command in command *list A* is *not* **`zero`** (FALSE), jump to **`fi`** and continue with the first statement following the **`fi`**.

© 2003 Hewlett-Packard Development Company, L.P.



The `if` Construct Flowchart

The **`test`** command is commonly used to control the flow of control, but *any* command can be used, since all UNIX system commands generate a return code, as demonstrated by the following example:

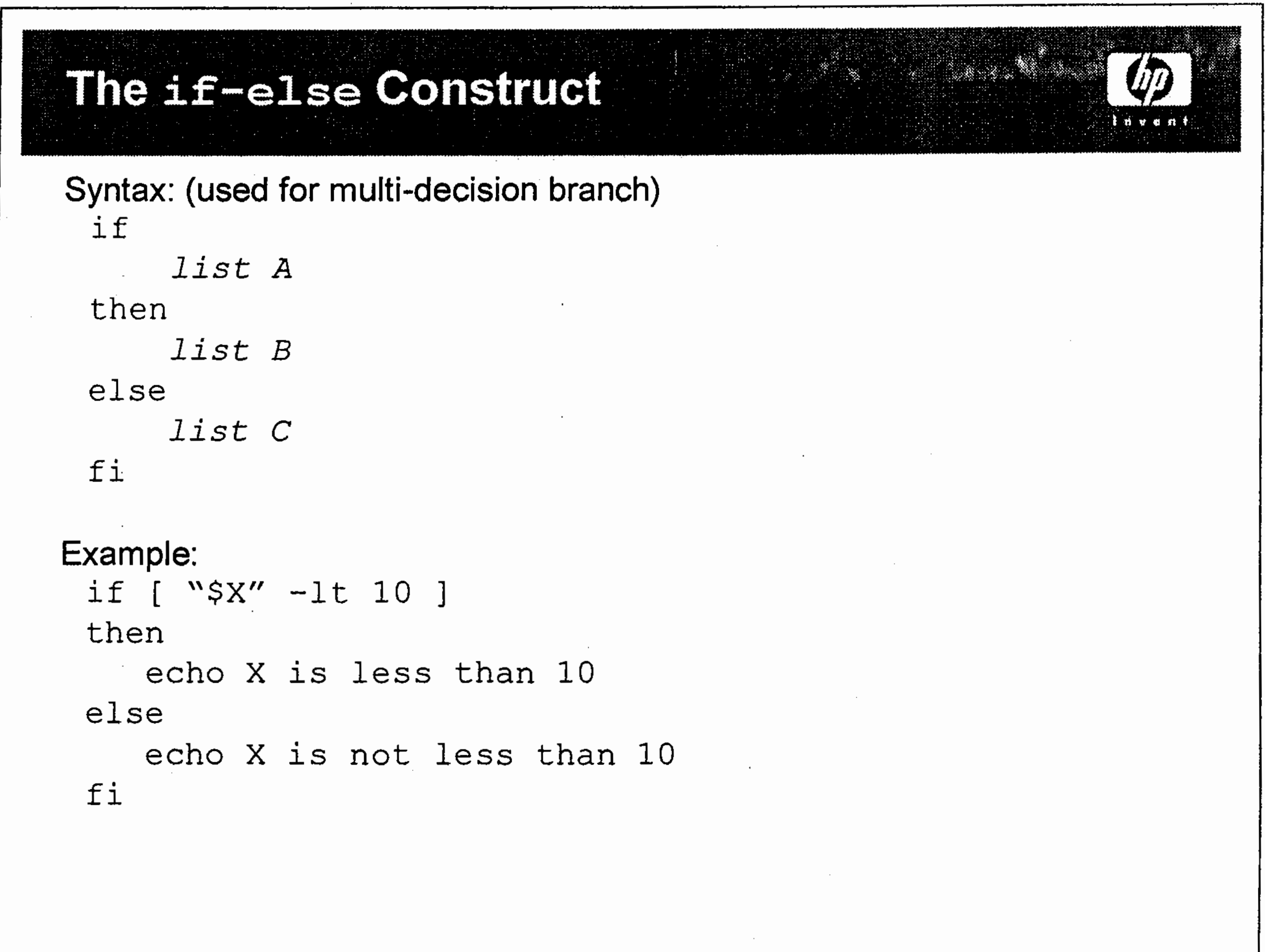
```
if
    grep kingkong /etc/passwd > /dev/null
then
    echo found kingkong
fi
```

The **`if`** construct also provides for program control when errors are encountered as in the following example:

```
if
    [ $# -ne 3 ]
then
    echo Incorrect syntax
    echo Usage: cmd arg1 arg2 arg3
    exit 99
fi
```


© 2003 Hewlett-Packard Development Company, L.P.

17–9. SLIDE: The `if-else` Construct



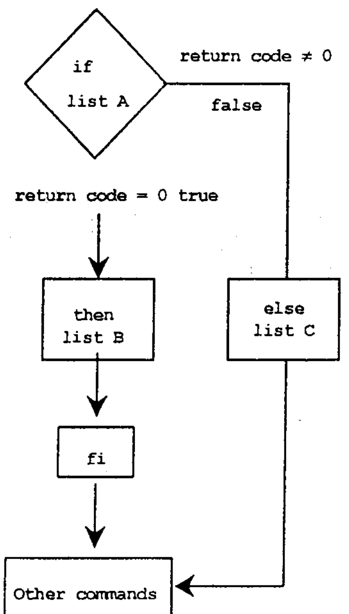
Student Notes

The **`if-else`** construct allows you to execute one set of commands if the return code of the controlling command is `0` (true) or another set of commands if the return code of the controlling command is non-zero (false).

The execution of the **`if`** construct in this case would be

1. Command *list A* is executed.
2. If the return code of the *last* command in command *list A* is a `0` (TRUE), execute command *list B*, then continue with the first statement following the **`fi`**.
3. If the return code of the *last* command in command *list A* is not `0` (FALSE), execute command *list C*, then continue with the first statement following the **`fi`**.

© 2003 Hewlett-Packard Development Company, L.P.



The `if-else` Construct Flowchart

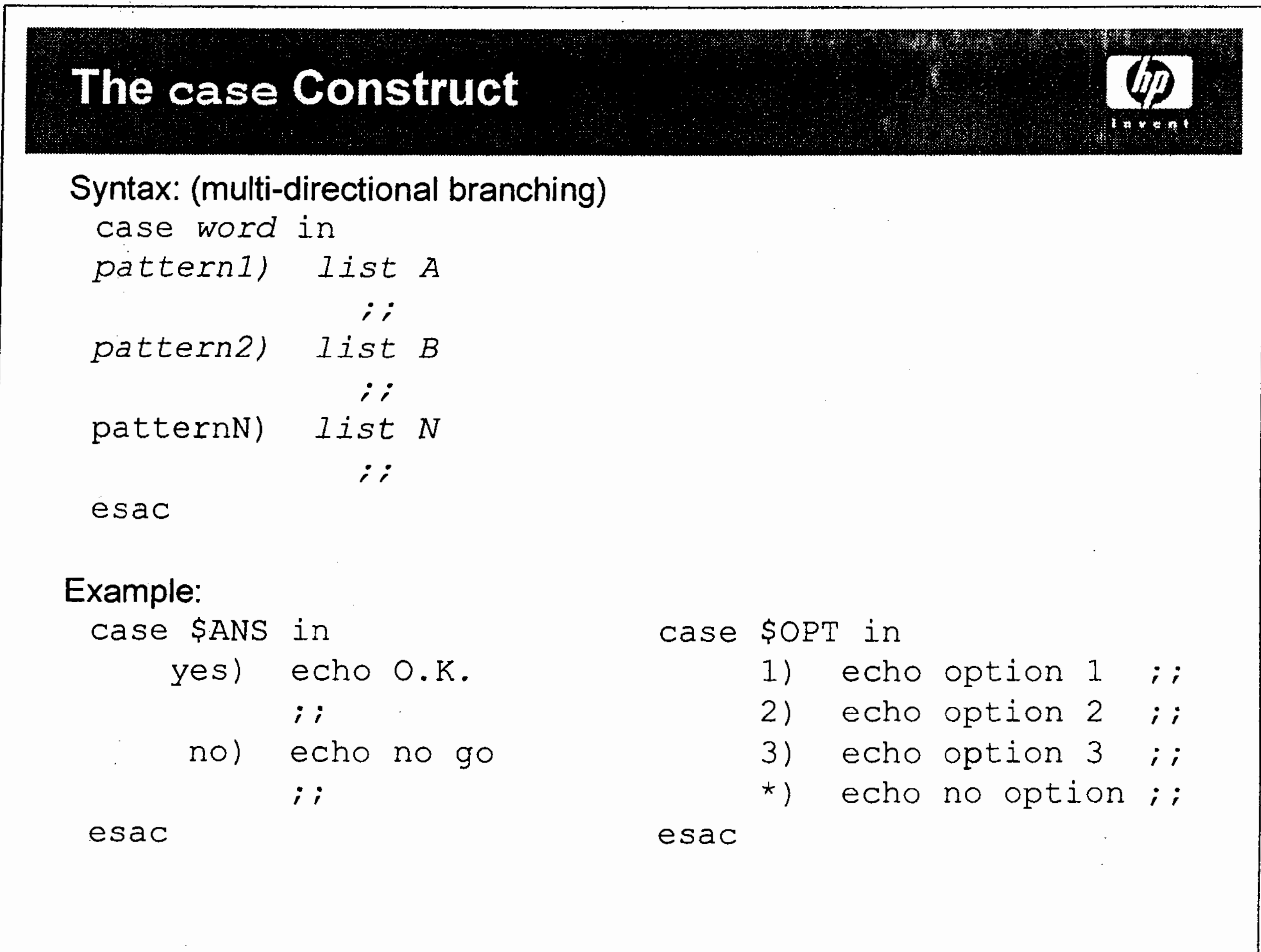
Note that list C can contain any UNIX system command including **`if`**. For example, extend the example on the slide to determine if the value of the variable X is less than 10, greater than 10 or equal to 10. This decision could be determined with

```
if
    [ "$X" -lt 10 ]
then
    echo X is less than 10
else
    if
        [ "$X" -gt 10 ]
    then
        echo X is greater than 10
    else
        echo X is equal to 10
    fi
fi
```

Notice how the indenting style enhances the readability of the code section. It is readily apparent which **`if`** goes with which **`fi`**. Notice also that *every* **`if`** requires **`fi`**.

© 2003 Hewlett-Packard Development Company, L.P.

17–10. SLIDE: The case Construct



Student Notes

The **if-else** construct can be used to support multidirectional branching, but it becomes cumbersome when more than two or three branches are required. The **case** construct provides a convenient syntax for multi-way branching. The branch selected is based on the sequential comparison of a word and supplied patterns. These comparisons are strictly string-based. When a match is found, the corresponding list of commands will be executed. Each list of commands is terminated by a double semicolon (`;;`). After finishing the related list of commands, program control will continue at the **esac**.

The *word* typically refers to the value of a shell variable.

The *patterns* are formed with the same format as generating filenames, even though we are not matching filenames.

The following special characters are allowed:

*	Matches any string of characters including the null string.
?	Matches any single character.

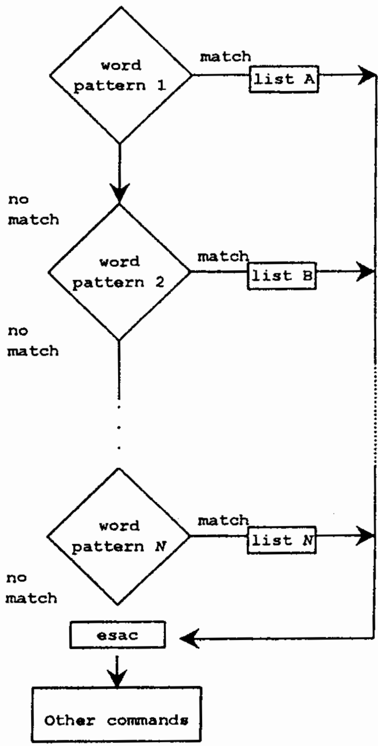
© 2003 Hewlett-Packard Development Company, L.P.

`[...]` Matches any one of the characters enclosed in the brackets. A pair of characters separated by a minus will match any character between the pair (lexically).

There is also the addition of the | character which means *OR*.

Please note that the right parenthesis and the semicolons are mandatory.

The **case** construct is commonly used to support menu interfaces or interfaces that will make some decision based on several user input options.



The case Construct Flowchart

© 2003 Hewlett-Packard Development Company, L.P.

17–11. SLIDE: The case Construct — Pattern Examples

The case Construct — Pattern Examples

The `case` construct patterns use the same special characters that are used to generate file names.

```
$ cat menu_with_case
 echo                  COMMAND MENU
 echo       d to display time and date
 echo       w to display logged-in users
 echo       l to list contents of current directory
 echo         Please enter your choice :
 read choice
 case $choice in
     [dD]*)    date ;;
     [wW]*)    who  ;;
     l*|L*)    ls   ;;
         *)    echo Invalid selection ;;
 esac
$
```

Student Notes

This slide shows an example of the case construct, with patterns that are less strict than the previous slide. Using patterns you can support user responses that are not case sensitive, or search for a response that contains a certain string pattern *or* another.

It is common to conclude all **case** patterns with a `*)` in order to generate a message to the user to inform him or her that he or she did not provide an acceptable response.

© 2003 Hewlett-Packard Development Company, L.P.

17–12. SLIDE: Shell Programming — Branches — Summary

Shell Programming — Branches — Summary

Return Code	Return value from each program – `echo $?`
Numeric test	`[ "$num1" -lt "$num2" ]`
String test	`[ "$string1" = "$string2" ]`
File test	`test -f` *filename*
exit *n*	Terminates program and sets the return code

```
if                                 case word in
      command listA                pattern1) command list
then                               ;;
         command listB             pattern2) command list
else                               ;;
                                   *)        command list

      command listC                ;;
fi                                 esac
```

Decision based on *return code* of last command in *listA*

The string *word* is compared to each string pattern

Student Notes

© 2003 Hewlett-Packard Development Company, L.P.

17–13. LAB: Shell Programming Branches

Directions

Complete the following exercises and answer the associated questions.

1. Define a variable called *X* equal to some string. Use the **`test`** command to determine if the value of *X* is the string *xyz*. (Hint: you must display the return value of the **`test`** command.)

2. Define a variable called *Y* and assign it to some number. Use the **`test`** command to determine if the value of Y is greater than 0. (Hint: you must display the return value of the **`test`** command.)

3. In a shell program, create an **`if`** statement that will echo **`yes`** if the argument passed is equal to *abc* and **`no`** if it is not.

4. Create a short shell program that will prompt the user to enter a number. Store the user's input in a variable called *Y*. Use an **`if`** construct that will echo **`Y is positive`** if *Y* is greater than zero and **`Y is not positive`** if it is not. Also display the value of *Y* to the user. (Hint: the **`read`** command will retrieve the user's input.)

© 2003 Hewlett-Packard Development Company, L.P.

5. Write a shell program that checks the number of command line arguments and echoes an error message if there are not exactly three arguments or echoes the arguments themselves if there are three. (Hint: The number of command line arguments is available through the special shell variable **$#**. What special shell variable stores all of the command line arguments?)

6. Write a shell program that prompts the user for input and takes one of three possible actions:

   - If the input is A, the program should echo "good morning".
   - If the input is B or b, the program should echo "good afternoon".
   - If the input is C or quit, the program should terminate.
   - If any other input is provided, issue an error message and exit the program setting the return code to 99.

7. Create a shell program that will prompt for a user-ID name. Verify that the user ID entered corresponds to an account on your system. If a legal user-id is provided, display the pathname of the user's home directory. If a user-id is entered that is not recognized, display an error message.

8. Use the **date** command to determine if it is morning (before 12:00 noon), afternoon (between 12:00 and 6:00 p.m.) or evening (after 6:00 p.m.). Depending on the time, create a shell program called **greeting** that will echo out the appropriate message: good morning, good afternoon or good evening. (Hint: The **date** command uses a 24-hour clock.)

© 2003 Hewlett-Packard Development Company, L.P.

9. Create a shell program that will ask the user if he or she would like to see the contents of the current directory. Inform the user that you are looking for a **yes** or **no** answer. Issue an error message if the user does not enter **yes** or **no**. If the user enters **yes** display the contents of the current directory. If the user enters **no**, ask what directory he or she would like to see the contents of. Get the user's input and display the contents of that directory. Remember to verify that the requested directory exists prior to displaying its contents.

10. Create a program **mycp** that will copy one file to another. The program will accept two command line arguments, a source and a destination. Check for the following situations:
    - It should make sure that the source and destination do not reference the same file.
    - The program should verify that the destination is a file.
    - The program should verify that the source file exists.
    - The program should check to see if the destination exists. If it does, ask the user if he or she wants to overwrite it.

11. Write a shell program called **options** that responds to command line arguments as follows:
    - If the first argument on the command line is -d, the program will run the date command.
    - If the first argument on the command line is -w, the program will list all of the users who are on the system.
    - If the first argument on the command line is -l, the program will list the contents of the directory provided as the second command line argument.
    - If no arguments or more than two arguments are on the command line, issue a usage message, and set the return code to 10.
    - If an option is provided that is not recognized, issue a usage message, and set the return code to 20.

© 2003 Hewlett-Packard Development Company, L.P.

© 2003 Hewlett-Packard Development Company, L.P.

Module 18 — Shell Programming — Loops

Objectives

Upon completion of this module, you will be able to do the following:

- Use the `while` construct to repeat a section of code while some condition remains true.
- Use the `until` construct to repeat a section of code until some condition is true.
- Use the iterative `for` construct to walk through a string of white space delimited items.

© 2003 Hewlett-Packard Development Company, L.P.

18–1. SLIDE: Loops — An Introduction

Purpose:	Repeat execution of a list of commands.
Control:	Based on the *return code* of a key command.

Three forms:	`while ... do ... done`
	`until ... do ... done`
	`for ... do ... done`

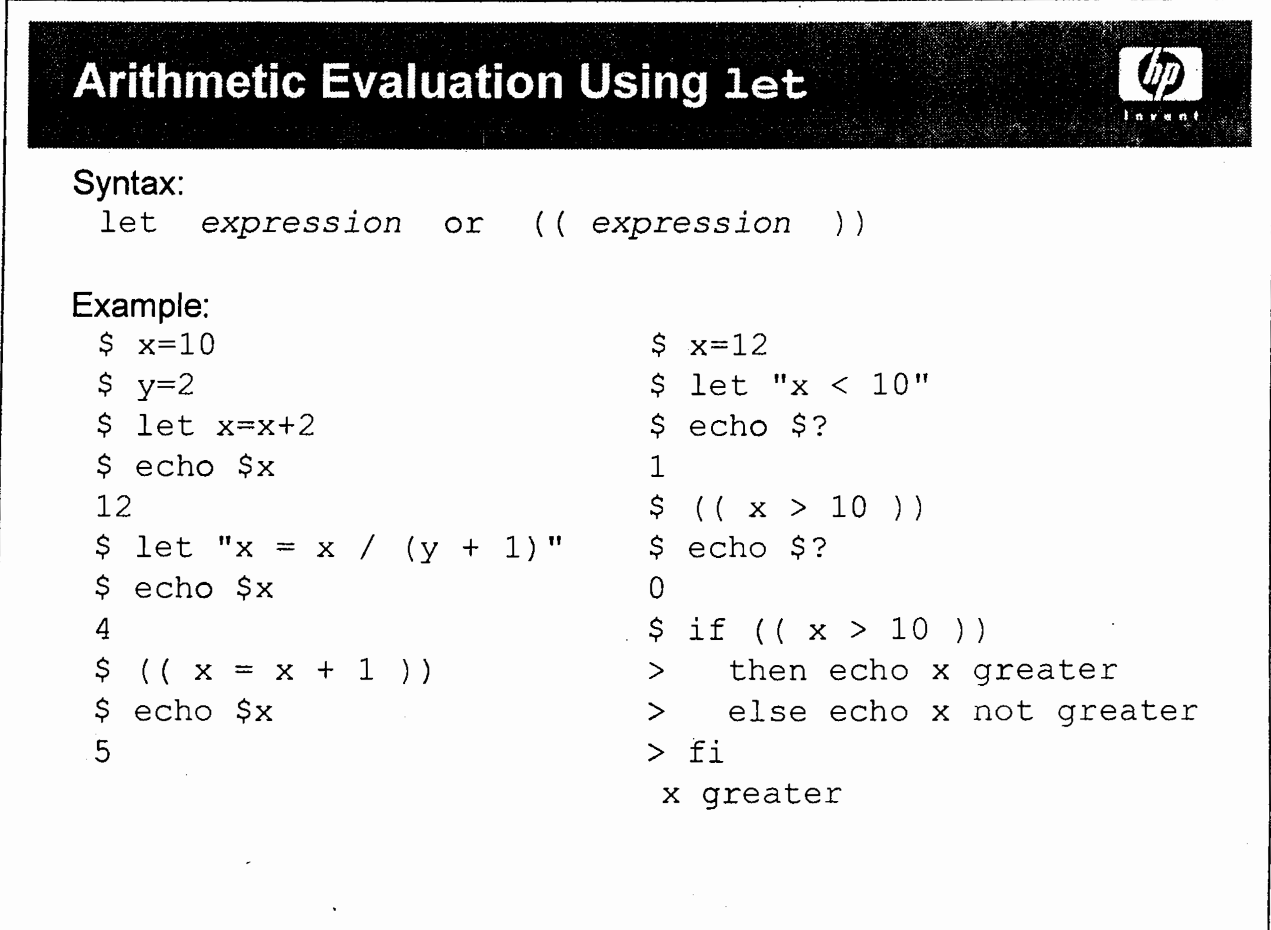
Student Notes

The looping constructs allow you to repeat a list of commands, and as in the branching constructs, the decision to continue or cease looping will be based on the return code of a key command. The **`test`** command is frequently used to control the continuance of a loop.

Unlike branches, which start with a keyword and end with the keyword in reverse (**`if/fi`** and **`case/esac`**), loops will start with a keyword and some condition, and the body of the loop will be surrounded by **`do/done`**.

© 2003 Hewlett-Packard Development Company, L.P.

18–2. SLIDE: Arithmetic Evaluation Using `let`



Student Notes

Loops are commonly controlled by incrementing a numerical variable. The `let` command enables shell scripts to use arithmetic expressions. This command allows long integer arithmetic. The syntax is shown on the slide, where *expression* represents an arithmetic expression of shell variables and operators to be evaluated by the shell. Using `(( ))` around the expression replaces using the `let`. The operators recognized by the shell are listed below, in decreasing order of precedence.

Operator	Description
`-`	Unary minus
`!`	Logical negation
`* / %`	Multiplication, division, remainder
`+ -`	Addition, subtraction
`<= >= < >`	Relational comparison

© 2003 Hewlett-Packard Development Company, L.P.

`== !=`	Equals, does not equal
`=`	Assignment

Parentheses can be used to change the order of evaluation of an expression, as in

```
let "x=x/(y+1)"
```

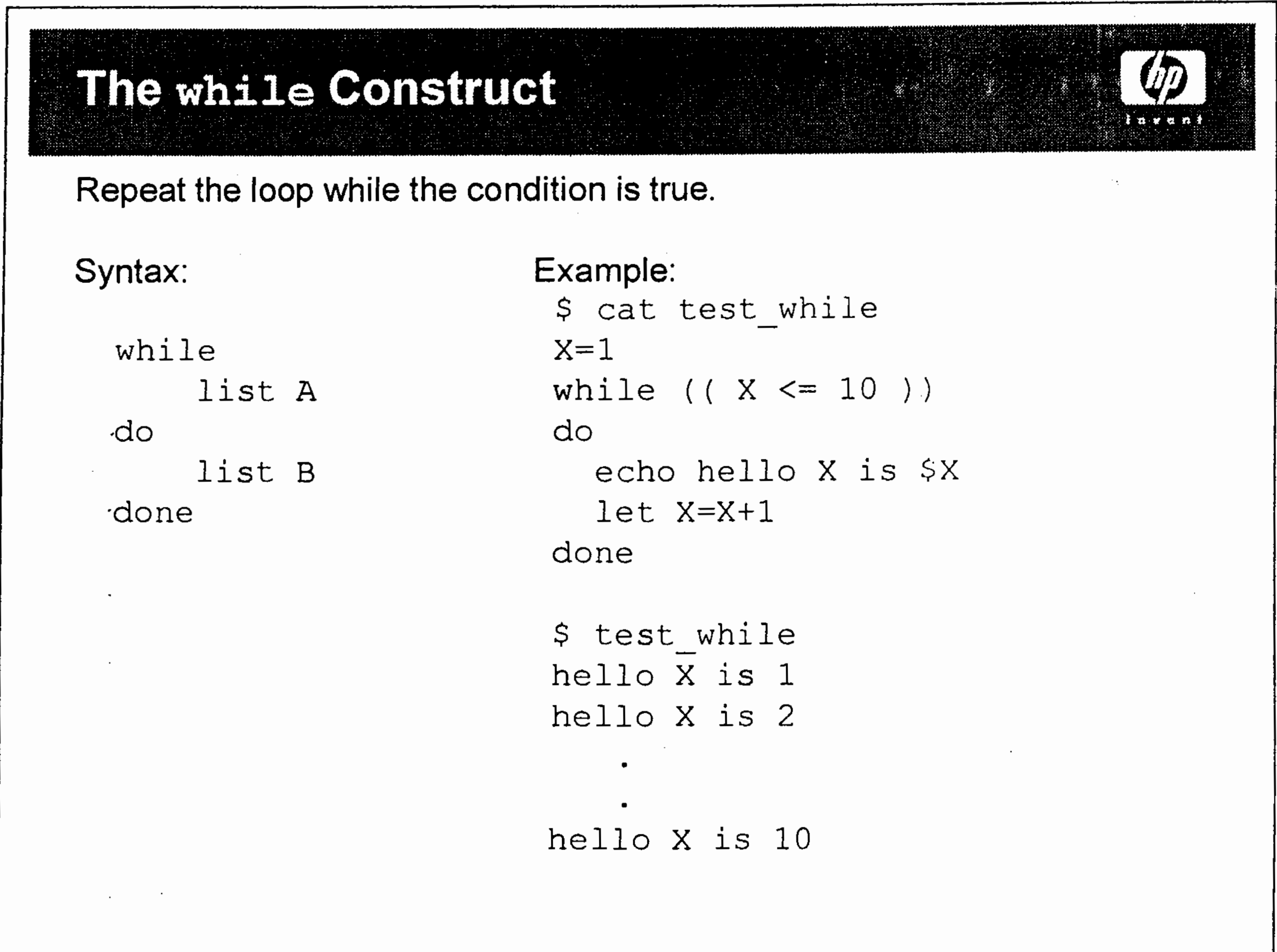
Note the double quotes are necessary to escape the special meaning of the parentheses. Also, if you wish to use spaces to separate operands and operators within the expression, double quotes must be used with **`let`**, or the `(( ))` syntax must be used:

```
let "x = x + (y / 2)" OR (( x = x + (y / 2) ))
```

When using the logical and relational operators, (`!, <=, >=, <, >, ==, !=`), the shell return code variable, `?` will reflect the *true* or *false* value of the result (`0` for *true*, `1` for *false*). Again, the double quotes must be used to prevent the shell from interpreting the less than and greater than signs as I/O redirection.

© 2003 Hewlett-Packard Development Company, L.P.

18–3. SLIDE: The `while` Construct



Student Notes

The **`while`** construct is a looping mechanism provided by the shell that will continue looping through the body of commands (*list B*) **`while`** a condition is true. The condition will be determined by the return code of the last command in *list A*. Often a **`test`** or **`let`** command is used to control the continuance of the loop, but any command can be used that generates a return code.

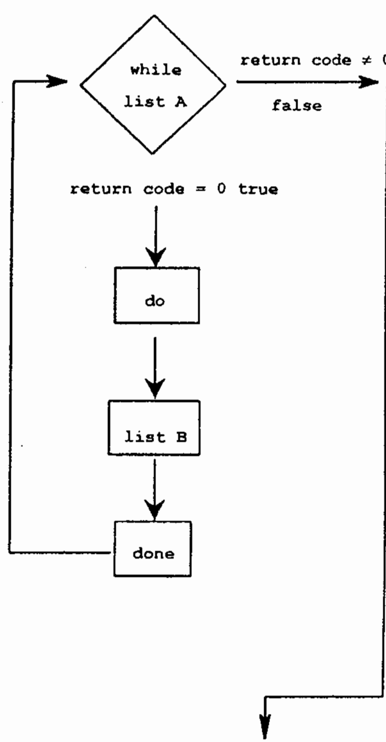
The example on the slide could have been written using a **`test`** command instead of the **`let`** command, as follows:

```
$ X=1
$ while [ $X -le 10 ]
> do
>       echo hello X is $X
>       let X=X+1
> done
```


© 2003 Hewlett-Packard Development Company, L.P.

The execution is as follows:

1. Commands in *list A* are executed.
2. If the return code of the *last* command in *list A* is 0 (*true*), execute *list B*.
3. Return to step 1.
4. If the return code of the *last* command in *list A* is *not* 0 (*false*), skip to the first command following the **done** keyword.



The `while` Construct Flowchart

WARNING:	**Be careful of infinite `while` loops. These are loops whose controlling command *always* returns true.**

© 2003 Hewlett-Packard Development Company, L.P.

```
$ cat while_infinite
while
     true
do
     echo hello
done

$ while_infinite
hello
hello
   .
   .
   .
```

Ctrl + c

© 2003 Hewlett-Packard Development Company, L.P.

18–4. SLIDE: The `while` Construct — Examples

The `while` Construct — Examples

Example A:

Repeat while ans is yes.

```
ans=yes
while
 [ "$ans" = yes ]
do
 echo Enter a name
 read name
 echo $name >> file.names
 echo "Continue?"
 echo Enter yes or no
 read ans
done
```

Example B:

Repeat while there are cmd line arg.

```
while (( $# != 0 ))
do
 if test -d $1
 then
   echo contents of $1:
   ls -F $1
 fi
 shift
 echo There are $# items
 echo left on the cmd line.
done
```

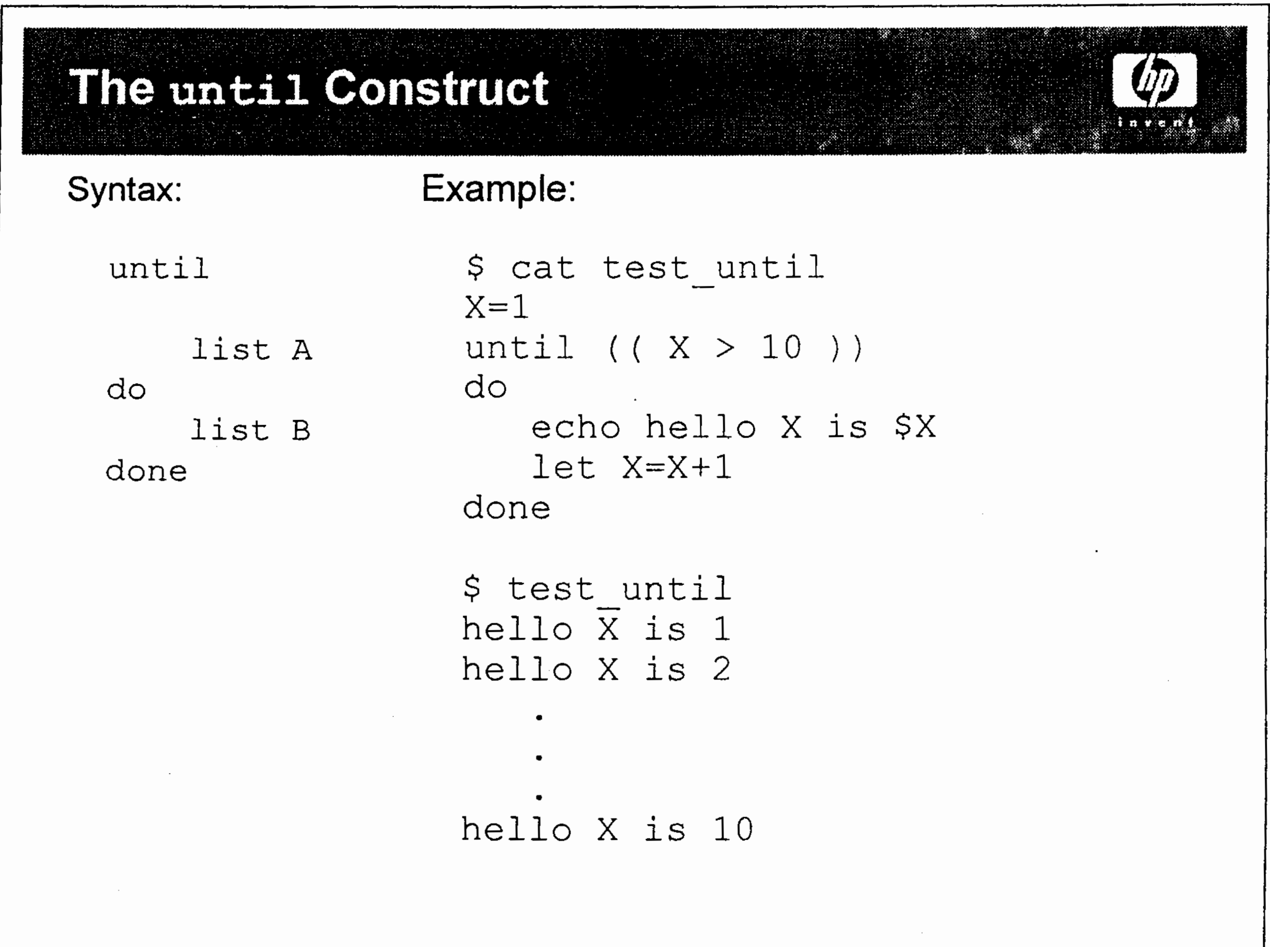
Student Notes

The slide shows two additional examples of the **while** construct. Example A is prompting the user for input, and determining whether the loop should be continued based on the user's response. Example B is looping through each of the arguments on the command line. If an argument is a directory, the contents of the directory will be displayed. If the argument is not a directory, it will simply be skipped over. Note the use of the **shift** command to allow access to each of the arguments one by one. When combined with the **while** command, this makes the loop very flexible. It does not matter if there is one argument or 100 arguments, the loop will continue until all of the arguments have been accessed.

Note that a **while** loop may need to be set up if you want to execute the loop at least once. Example A will execute the body of the loop at least once because *ans* has been set equal to **yes**. In Example B, if the program has been executed with no command line arguments (**$#** equals **0**), then the loop will not execute at all.

© 2003 Hewlett-Packard Development Company, L.P.

18–5. SLIDE: The until Construct



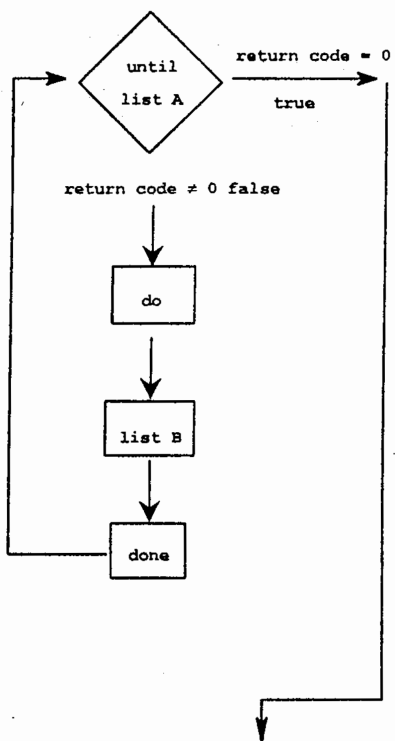
Student Notes

The `until` construct is another looping mechanism provided by the shell that will continue looping through the body of commands (*list B*) `until` a condition is true. Similar to the `while` loop, the condition will be determined by the return code of the last command *in list A*.

The execution is as follows:

1. Command list A is executed.
2. If the return code of the *last* command in list A is *not* 0 (*false*), execute list B.
3. Return to step 1.
4. If the return code of the *last* command in list A is 0 (*TRUE*), skip to the first command following the `done` keyword.

© 2003 Hewlett-Packard Development Company, L.P.



The `until` Construct Flowchart

CAUTION: Be careful of infinite `until` loops. These are loops whose controlling command *always* returns false.

```
$ X=1
$ until
>       [ $X -eq 0 ]
> do
>       echo hello
> done
hello
hello
   .
   .
   .
Ctrl + c
```


© 2003 Hewlett-Packard Development Company, L.P.

18–6. SLIDE: The `until` Construct — Examples

The `until` Construct — Examples

Example A:
Repeat until ans is no

```
ans=yes
until
 [ "$ans" = no ]
do
 echo Enter a name
 read name
 echo $name >> file.names
 echo "Continue?"
 echo Enter yes or no
 read ans
done
```

Example B:
Repeat until there are no cmd line arg.

```
until (( $# == 0 ))
do
 if test -d $1
 then
  echo contents of $1:
  ls -F $1
 fi
 shift
 echo There are $# items
 echo left on the cmd line.
done
```

Student Notes

The slide shows the same examples that were presented for the `while` construct, but now they are implemented with the `until` construct. Notice that the logic associated with the test conditions must be reversed to match the logic of the `until` construct.

Notice also that the sensitivity of the user input has changed slightly. Using the `while` construct, the loop will continue *only* if the user inputs the string `yes`. It is very strict in its condition for continuing the loop. Using the `until` construct the loop will continue as long as the user enters anything other than no. It is not as strict in its condition for continuing the loop. You may want to consider these issues when deciding which construct is most applicable to your interface.

Predefining the *ans* variable is not necessary either because it would be initialized to NULL. Since NULL is not equivalent to `no` the test would return false, and the loop would be executed. You just want to make sure that *$ans* is enclosed in quotes in the test expression to provide legal `test` syntax.

© 2003 Hewlett-Packard Development Company, L.P.

18–7. SLIDE: The `for` Construct

The `for` Construct

For each item in *`list`*, repeat the loop, assigning *`var`* to the next item in list until the *`list`* is exhausted.

Syntax:

```
for var in list
do
    list A

done
```

Example:

```
$ cat test_for
for X in 1 2 3 4 5
do
echo "2 * $X is \c"
let X=X*2
echo $X
done
$ test_for
2 * 1 is 2
2 * 2 is 4
2 * 3 is 6
2 * 4 is 8
2 * 5 is 10
```

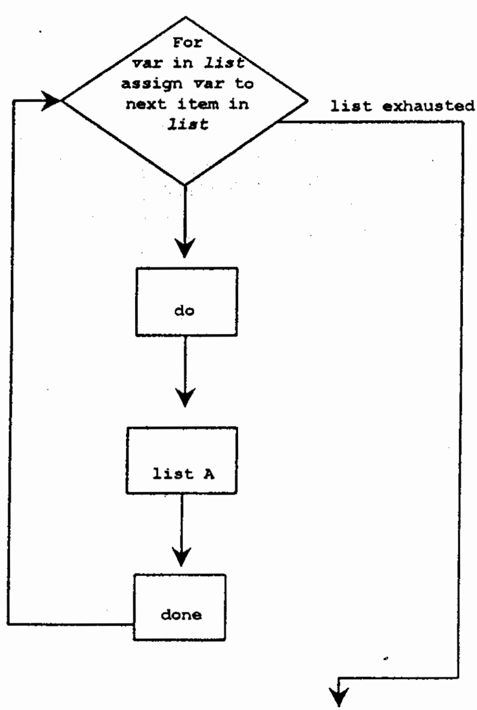
Student Notes

On the slide, the keywords are **`for`**, **`in`**, **`do`**, and **`done`**. *var* represents the name of a shell variable that will be assigned through the execution of the **`for`** *loop*. **`list`** is a sequence of strings separated by blanks or tabs that *var* will be assigned to during each iteration of the loop.

The construct works as follows:

1. The shell variable *var* is set equal to the first string in *list*.
2. Command list A is executed.
3. The shell variable *var* is set equal to the next string in *list*.
4. Command list A is executed.
5. Continue until all items from **`list`** have been processed.

© 2003 Hewlett-Packard Development Company, L.P.



The for Construct Flowchart

© 2003 Hewlett-Packard Development Company, L.P.

18–8. SLIDE: The `for` Construct — Examples

The `for` Construct — Examples

Example A:

```
$ cat example_A
for NAME in $(grep home /etc/passwd | cut -f1 -d:)
do
  mail $NAME < mtg.minutes
  echo mailed mtg.minutes to $NAME
done
```

Example B:

```
$ cat example_B
for FILE in *
do
  if
     test -d $FILE
  then
     ls -F $FILE
  fi
done
```

Student Notes

The `for` construct is a very flexible looping construct. It is able to loop through any list that can be generated. Lists can easily be created through command substitution, as seen in the first example. With the availability of pipes and filters, a list can be generated from almost anything.

If you require access to the same list many times, you might want to save it in a file. You can then use the `cat` command to generate the list for your `for` loop, as in the following example:

```
$ cat students
user1
user2
user3
user4
```


© 2003 Hewlett-Packard Development Company, L.P.

```
$ cat for_students_file_copy
for NAME in $(cat students)
do
    cp test.file /home/$NAME
    chown $NAME /home/$NAME/test.file
    chmod g-w,o-w /home/$NAME/test.file
    echo done $NAME
done
$
```

Accessing Command Line Arguments

You can generate the list from *command line arguments* with

```
for i in $*                    or           for i
do                                          do
    cp $i $HOME/backups                         cp $i $HOME/backups
done                                        done
```


© 2003 Hewlett-Packard Development Company, L.P.

18–9. SLIDE: The `break`, `continue`, and `exit` Commands

The `break`, `continue`, and `exit` Commands

`break [n]`	Terminates the iteration of the loop and skips to the next command after [the *n*th] done.
`continue [n]`	Stops the current iteration of the loop and skips to *the beginning* of the next iteration [of the *n*th] enclosing loop.
`exit [n]`	Stops the execution of the shell program, and sets the return code to *n*.

Student Notes

There may be situations where you need to discontinue a loop prior to the loop's normal terminating condition. The **`break`** and **`continue`** provide unconditional flow control. They are commonly used when an error condition is encountered to terminate the current iteration of the loop. The **`exit`** command is used when a situation cannot be recovered from, and the entire program must be terminated.

The **`break`** command will cause the loop to terminate and control to be passed to the command immediately following the **`done`** keyword. You will completely break out of the designated loops, and continue with the following commands.

The **`continue`** command is slightly different. When encountered, the continue command will skip the remaining commands in the body of the loop and transfer control to the top of the loop. Thus the **`continue`** command allows you to just terminate one iteration of the loop but continue execution at the top of the loop just interrupted.

In the **`while`** and **`until`** loops, the process will continue at the beginning of the initialization list. In the **`for`** loop the process will set the variable to the next item in the list, and then continue.

© 2003 Hewlett-Packard Development Company, L.P.

The **`exit`** command will stop the execution of a shell program and set the return value for the shell program to the argument, if specified. If no argument is supplied, the return value of the shell program is set to the return value of the command that executed immediately prior to the **`exit`**. The **`return`** command will behave just as the exit within a shell function.

NOTE:	The flow of control of a loop should normally be terminated through the condition at the top of the loop (**`while`**, **`until`**) or by exhausting the list (for). These should be used only when an irregular or error condition occurs in the loop.

Example

```
while
     cmd1
do
     cmdA
     cmdB
     while
          cmdC
     do
          cmdE
          break 2
          cmdF
     done
     cmdJ
     cmdK
done
cmdX
```

1. What command will be executed following the **`break 2`**?
2. What if the **`break 2`** is replaced simply with a **`break`**?
3. What about a **`continue 2`**?
4. What about a simple **`continue`**?

© 2003 Hewlett-Packard Development Company, L.P.

18–10. SLIDE: `break` and `continue` — Example

The `break` and `continue` — Example

```
while
    true
do
    echo "Enter file to remove: \c"
    read FILE
    if test ! -f $FILE
    then
        echo $FILE is not a regular file
        continue
    fi
    echo removing $FILE
    rm $FILE
    break
done
```

Student Notes

This example shows an effective use of the **`break`** and **`continue`** commands. The command executed as the test condition of the **`while`** loop is the **`true`** command which will always generate a *true* result; this means that this loop is an *infinite* loop which will loop forever unless some command inside the loop terminates it (which the **`break`** command does). If the file entered is not a regular file, an error message is printed and the **`continue`** command causes the user to be prompted for the file name again. If the file *is* a regular file, it is removed, and the **`break`** command is used to break out of the infinite loop.

© 2003 Hewlett-Packard Development Company, L.P.

18–11. SLIDE: Shell Programming — Loops — Summary

Shell Programming — Loops — Summary

`let` *`expression`*	Evaluate an arithmetic expression
`((`*`expression`*`))`	Evaluate an arithmetic expression
`while` *`condition is true`* `do ... done`	While
`until` *`condition is true`* `do ... done`	Until
`for var in` *`list`* `do ... done`	For
`break [`*`n`*`]`	Break out of loop
`continue[`*`n`*`]`	Terminate current iteration of loop
`exit [`*`n`*`]`	Terminate the program

Student Notes

© 2003 Hewlett-Packard Development Company, L.P.

18–12. LAB: Shell Programming — Loops

Directions

Complete the following exercises and answer the associated questions.

1. Create a program called `double_it` that will prompt the user for a number and then display two times the number.

2. Create a program called `sum_them` that will prompt the user to input 10 numbers. The program will add all of the numbers that the user has entered, and display the final sum. (Hint: accumulate the sum each time a new number is entered.)
   Optional: Modify `sum_them` so that the number of numbers that the user would like to add together is provided through a command line argument. For example, `sum_them 6` would prompt the user for six numbers and add them together.

3. Create a program called `words_in` that will continue to prompt the user to input a single word until the user enters `quit`. Save each word that is entered. After the user types `quit` echo back all of the words that have been entered. Can you complete this exercise with a `while` loop? With an `until` loop? Select the one you prefer. (Optional: display all of the words entered in alphabetical order.)

4. In a shell program create a `for` loop that will:
   - create the directories `Adir`, `Bdir`, `Cdir`, `Ddir`, `Edir`
   - copy `funfile` to each directory
   - list the contents of each directory to verify the copy
   - echo a message when each iteration of the loop is complete

© 2003 Hewlett-Packard Development Company, L.P.

5. Write a shell program called `new_files` that will accept a variable number of command line arguments. The shell program will create a new file associated with each command line argument (use the `touch` command), and echo a message that notifies the user as each file is created.

6. Use `vi` to create a file called `mailtest`. At your shell prompt, create an interactive `for` loop to mail `mailtest` to everyone who is logged on. (Hint: use `who` and `cut` with command line substitution to generate the list for the `for` loop.)

7. Create a shell program called `my_menu` that will display a simple menu that has three options.
   a. The first option will run `double_it` (Exercise 1).
   b. The second option will `run sum_them` (Exercise 2).
   c. Quit.

   The menu should be redisplayed after each selection is completed, until the user enters 3.

8. Create a program called `msg_me` that will display a message to your screen once every 5 seconds, for a minute. (Hint: look up the `sleep` command.) You might want to store the message in a separate text file so that it can be easily changed.

© 2003 Hewlett-Packard Development Company, L.P.

9. Write a shell program called `ison` that will *run in the background* and check every 60 seconds whether a particular user has logged into the system. The user name should be passed into `ison` as a command line argument. When the user logs in, print a message on your terminal informing you of the login, and report what terminal the user logged into. (Hint: Use the `sleep` command.)

   If you are on a standalone system in a network, you might want to try the `rwho` command.

10. Create a directory called `.waste` in your home directory. Write a shell program called `myrm` that will move all of the files you delete into the `.waste` directory, your wastebasket. This is a useful tool which will allow restoration of files after they have been removed. Remember, the UNIX system has no *undelete* capability.

    Have `myrm` also include the options:

`-l`	List contents of the wastebasket
`-d`	Dump the contents of the wastebasket

© 2003 Hewlett-Packard Development Company, L.P.

Module 19 — Offline File Storage

Objectives

Upon completion of this module, you will be able to do the following:

- Use the `tar` command for storing files to tape.
- Use the `find` and `cpio` commands for storing files to tape.
- Retrieve files that were stored using `tar` or `cpio`.

© 2003 Hewlett-Packard Development Company, L.P.

19–1. SLIDE: Storing Files to Tape

Storing Files to Tape

- To store files to a tape you must know the device file name for your tape device.
- Typical names might be
  `/dev/rmt/0m` 9-track tape or DDS tape(old name)
  `/dev/rmt/c0t3d0BEST` 9-track tape or DDS tape
- Ask your system administrator which device file accesses the tape drive.
- Commands to perform file backups include
  `tar`
  `cpio`

Student Notes

There are many times when the average user of a UNIX system will want to save copies of files to some removable media. Popular media used for backups include 9-track tape (1/2-inch reel), or DDS format DAT tape. This module is designed to give you the basics of storing and retrieving copies of files to and from tape. Keep in mind that your system administrator is usually responsible for backing up the entire system; you should coordinate all tape backups through your system administrator.

NOTE: The *only* way to recover a file that has been deleted is to restore it from a tape backup.

Regarding DDS Tapes

Since the introduction of HP DAT products in 1990, HP has only supported DDS cartridges as the storage medium. DAT audio tapes are not supported, and the use of DAT tapes will invalidate the drive warranty.

© 2003 Hewlett-Packard Development Company, L.P.

DDS cartridges are built to a much higher standard than ordinary DAT tapes. DDS cartridge designs and tape formulations are rigorously tested before they are permitted to carry the DDS logo. Non-DDS tapes can appear to work in a DAT drive but can cause data loss, tape jams, head clogging, and permanent damage to the drive.

Be sure you are using DDS format DAT tapes. Some tapes that are marked *Data Grade* are not necessarily DDS format tapes.

DDS format tapes carry this logo:

DDS Format Tape Logo

© 2003 Hewlett-Packard Development Company, L.P.

19–2. SLIDE: The `tar` Command

The `tar` Command

Syntax:

```
tar -key [f  device_file][file. . .]
```

Examples:

Create an archive:

```
$ tar -cvf /dev/rmt/0m myfile
```

Get a table of contents from the archive:

```
$ tar -tvf /dev/rmt/0m
```

Extract a file from the archive:

```
$ tar -xvf /dev/rmt/0m myfile
```

Student Notes

The `tar` command archives the tape file. It saves and restores files onto magnetic tape. Its function is controlled by its first argument called the **key argument**.

Examples of valid key arguments are

`c`	A new archive is **created**.
`x`	Files are **extracted** from the archive.
`t`	A **table of contents** of the archive is printed.
`r`	Files are added to the end of the archive.
`u`	Files are added to the archive if they are new or modified.

© 2003 Hewlett-Packard Development Company, L.P.

Modifiers can be added to these keys.

`v`	Echoes filenames to screen as they are archived or restored - verbose.
`f file`	Designates the ***`file`*** where the archive will be written. Note: The ***`file`*** does not have to be a device file. You can create an archive file under your directory on your disk. The default is **`/dev/rmt/0m`**.

The Linux version of **`tar`** has more, varied command options. It works in the same manner as in HP-UX. For details, refer to the appropriate man pages.

© 2003 Hewlett-Packard Development Company, L.P.

19–3. SLIDE: The `cpio` Command

The `cpio` Command

Two modes:

`cpio -o[cvx]`	Generate an archive. Read list of files from *stdin*. Archive is written to *stdout*.
`cpio -i[cdmtuvx]`	Restore from an archive. Archive is read from *stdin*.

Examples:

Create an archive of all files under current directory:

```
$ find . | cpio -ocv > /dev/rmt/0m
```

Restore all files from an archive:

```
$ cpio -icdmv < /dev/rmt/0m
```

Student Notes

This command makes archive copies of files and directories in HP-UX. `cpio` stands *for copy input to output.* `cpio` has two modes:

-o *Make a backup.* Read standard input, and copy each file to standard output.

-i *Restore a backup.* Read standard input for the backup data, and recreate it on the disk.

When creating backups, the `cpio -o` command uses standard input as its source of file names and standard output as the archive output. Since the defaults are standard input for a file list and standard output for the archive, you have to specify the tape as a device, and you must provide a list of files to store. This is usually accomplished by piping the output of **`find`** into `cpio`.

When restoring an archive, the `cpio -i` command will read the archive from standard input (the tape special device file) and restore the file contents to your disk. The file names created will depend on whether the archive was created with relative or absolute pathnames.

© 2003 Hewlett-Packard Development Company, L.P.

There are several options that we will use with the major options `-o` and `-i`.

-o	-i	Option Function
-c	-c	Writes header in ASCII format. (If used with `-o`, it must be used with `-i`)
—	-d	Recreates directory structure as needed.
—	-m	Retains current modification date. (Important for version control.)
—	-t	Display table of contents of archive.
—	-u	Unconditionally restores. (If the file already exists, this option overwrites the file.)
-v	-v	Displays a list of files copied.
-x	-x	Handles special (device) files.

Additional Examples

- Get table of contents:

  ```
  $ cpio -ict < /dev/rmt/0m
  ```

- Restore a single file:

  ```
  $ cpio -icudm "filename" < /dev/rmt/0m
  ```

- Restore all file names matching pattern:

  ```
  $ cpio -icudm '*filename*' < /dev/rmt/0m
  ```

Notes on the `find` Command

- The **`find`** command is commonly used with the backup commands to generate the list of file names that will be backed up. Notice that find can generate a list of relative path names (**`find .`**) or a list of absolute path names (**`find /home/user3`**). The method used to generate the list of file names will define how the file names will be saved on the tape.

Syntax:

```
find path-list [expression]
```

The ***`expression`*** supports many keywords, which can specify search criteria. For details, see the manual page **`find(1)`**.

The Linux version of **`cpio`** has more and varied command options, although it works in the same manner as in HP-UX. For details, refer to the appropriate man pages.

© 2003 Hewlett-Packard Development Company, L.P.

19–4. LAB: Offline File Storage

Directions

Complete the following exercises. Write the commands you would use to perform the following tasks with a device file name of **/dev/rmt/0m**, or if there is a tape drive available, your instructor may have you actually perform some of these commands. You may replace **/dev/rmt/0m** in your command's syntax with a file name if you want to try the exercises without accessing a tape drive.

```
$ tar cf /tmp/archive_file mydir
```

1. Using **tar**, create an archive of all files in your **HOME** directory that start with **abc**.

2. Obtain a table of contents listing of this tape archive.

3. Using **find** and **cpio**, make a backup of your whole directory structure from your *HOME* directory on down.

4. Remove the file **backup** from your current directory. Then restore the file from tape using the **cpio** command.

5. Create the directory **$HOME/tree.cp**. Look up the pass mode of the **cpio** command in **cpio(1)**. Using the **cpio** command in the pass mode, recreate the directory structure **$HOME/tree** under the directory **$HOME/tree.cp**.

© 2003 Hewlett-Packard Development Company, L.P.

Appendix A — Commands Quick Reference Guide

Objectives

To provide a list of frequently used commands along with an explanation of proper use.

© 2003 Hewlett-Packard Development Company, L.P.

A–1. Commands Quick Reference Guide

General Commands

exit	terminate terminal session and log out
man *cmd*	display manual page for *cmd*
laserROM	initiate an HP LaserROM documentation reference session
absolute path	complete designation of a file's or directory's location in the UNIX hierarchy. *ALWAYS* starts with /
relative path	designation of a file's or directory's location from your current position in the UNIX hierarchy
.	current directory
..	parent directory
pwd	display current directory location in hierarchy
cd *dir*	change to designated directory
cd	change to HOME directory
mkdir *dir*	create directory
rmdir *dir*	remove directory
ls *file* or *dir*	list the file or contents of directory
ls -a	list all of the files, including hidden files
ls -F	list files with format flag / — denotes directory * — denotes executable — denotes regular file | — denotes FIFO file
ls -l	display files in long format including permissions, ownership and size **rwx rwx rwx** user group others **r** — read access (mode value = 4) **w** — write access (mode value = 2) **x** — execute access (mode value = 1)

© 2003 Hewlett-Packard Development Company, L.P.

`ll`	HP-UX shorthand for `ls -l`
`lsf`	HP-UX shorthand for `ls -F`
`lsr`	HP-UX shorthand for `ls -R`
`lsx`	HP-UX shorthand for `ls -x`
`cat` [***file***]	display contents of ***file***
`more` [***file***]	display contents of file one screen at a time [space] — next screen [Return] — next line `q` — quit `more`
`tail - `***n file***	display the last ***n*** lines of a file
`pr` ***file***	format file for printing
`lp` ***file***	queue file to be printed (HP-UX)
`lpr` ***file***	queue file to be printed (Linux)
`pr` ***file*** | `lp`	format and print file (HP-UX)
`pr` ***file*** | `lpr`	format and print file (Linux)
`lpstat -t`	display status of the printer(s) and print system (HP-UX)
`lpq`	display status of the printer(s) and print system (Linux)
`cancel` ***jobnumber***	cancel print job (HP-UX)
`lprm` ***jobnumber***	cancel print job (Linux)
`touch` *file*	create empty file or update timestamp on existing file
`cp [-i]` ***f1 f2***	copy ***f1*** to ***f2***
`cp [-i]` ***f1 f2 ... dir***	copy file(s) to another directory
`ln [-i]` ***f1 f2***	link ***f1*** to ***f2*** ***f1*** and ***f2*** access same data space on disk
`ln -s` ***dir1 dir2***	symbolically link ***dir1*** to ***dir2***
`mv [-i]` ***f1 f2***	rename ***f1*** to ***f2***

© 2003 Hewlett-Packard Development Company, L.P.

mv [-i] ***f1 f2... dir***	move file(s) to another directory
mv [-i] ***dir1 dir2***	rename ***dir1*** to ***dir2***
rm ***f1 f2...***	remove files
rm -i ***f2 f2...***	remove files interactively
rm -r ***dir***	remove directory and EVERYTHING below directory
who	display users logged in to your system
who am I	display your user id and terminal location
whoami	display your user id
news	(HP-UX) display system news (updates file **$HOME/.news_time**)
write ***username***	start interactive communication with ***username***
mesg y	allow your terminal to receive messages
mesg n	disables receipt of messages by your terminal
mail ***username***	send mail message to ***username***
mail	read mail messages **?** — mail help **d** — delete previous message **s** ***file*** — save message to ***file*** **q** — quit mail
mailx ***username***	send mail message to ***username***
mailx	read mail messages
elm	HP utility to send and read mail messages
echo ***string***	display ***string***
banner ***string***	display ***string*** in large letters
date	display the system time and date
id	display current user id and group status
chmod ***mode file***	change permissions for file to ***mode*** **chmod +x** ***file*** **chmod 777** ***file***

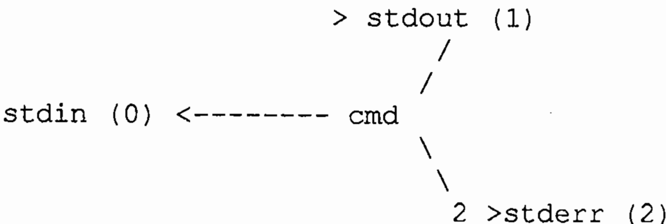
© 2003 Hewlett-Packard Development Company, L.P.

umask ***mode***	remove ***mode*** from default permissions
chown ***username file***	change ownership of file to ***username*** refer to **/etc/passwd** (Not available to non-root users, by default, in Linux)
chgrp ***groupname file***	change group access of file to ***groupname*** refer to **/etc/group** (Not available to non-root users, by default, in Linux)
su ***username***	switch user id to ***username***
newgrp ***groupname***	switch group id to ***groupname***
passwd	change the password for your account
vi ***filename***	Start a **vi** edit session on a file

Filename Generation

*	Match zero or more characters
?	Match any single character
[amqp]	Match specific characters, in this case **a**, **m**, **q**, **p**
[a-z]	Match a range of characters, in this case **a** through **z**
[!a-z]	Do NOT match a character in the range

File Input/Output Redirection: cmd <> file

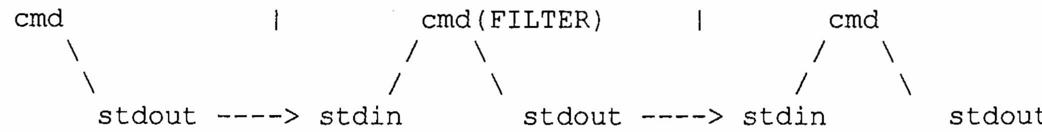


cmd **<** ***file***	get input for **cmd** from a file
cmd **>** ***file***	send **stdout** of **cmd** to a file
cmd **2>** ***file.err***	send **stderr** of **cmd** to a file
cmd **>** ***file*** **2>** ***file.err***	send stdout and **stderr** to files

© 2003 Hewlett-Packard Development Company, L.P.

`cmd >&2`	send **stdout** to **stderr** Useful when generating error messages with echo **echo error message text >&2**

Piping: cmd <> cmd



`cmd1 | cmd2`	Take output of **cmd1** and send it in to **cmd2**

Shell Variables

name=lisa	assign a value to the variable ***name***
export name	transport the variable ***name*** to the environment
set	display all variables defined in the current shell
env	display just the environment variables
echo ***enter a name***	prompt for user input
read name	read the user input and assign to variable ***name***
echo $name	display the value (**$**) of the variable ***name***
grep $name /etc/passwd	search for value of ***name*** in **/etc/passwd**
cmd arg1 arg2 arg3 ... arg9 **$0 $1 $2 $3 ... $9**	command line containing arguments variables for command line arguments
shift ***n***	shift through command line arguments
echo $#	display number of command line arguments
echo $*	display all (current) command line arguments
exit #	terminate program and set return value to **#**
echo $?	display return value of last command

Quoting

****	escapes special meaning of next character
'string'	escapes special meaning of all characters between quotes

© 2003 Hewlett-Packard Development Company, L.P.

`"string"`	escapes special meaning of all characters between quotes except **$**, \, and ` (grave accent)

Command Substitution

`cmd1 `cmd2``	Executes a command within a command line

```
banner $(date)
dirs=$(ls -F | grep /)
X=$(expr $X + 1)
for name in $(who | cut -f1 -d" ")
```

Filters

cut -c ***list [file]***	cut and display specified columns
cut -f ***list*** **-d** ***char [file]***	cut and display specified fields **-d** *char* — *char* represents the delimiting character between fields

Example:
```
who | cut -c12-18
cut -f1,6 -d: /etc/passwd
```

grep [-inv] *pattern* [*file*]	search for *pattern* in files **-i** — ignore case of letters in pattern **-n** — display line number where pattern found **-v** — display lines that DO NOT contain pattern

Example:
```
grep user /etc/passwd
who | grep user3
```

more [*file*]	display file one screen at a time

Example:
```
ps -ef | more
sort funfile | more
```

pr [- #] [-o #] [-h "*title info*"] [*file*]	format output to screen **-#** — provide # columns of output **-o#** — offset output # columns from left margin **-h** "*text*" — replaces default header with *text*

Example:
```
pr funfile | more
```

sort [-ndt *X*] [+***field***] [***file***]	**-n** — numeric sort **-d** — dictionary sort **-t** *X* — use *X* as the delimiter between fields

© 2003 Hewlett-Packard Development Company, L.P.

	`+field` — field to base sort on (field numbers start with 0)

Example:
```
sort names
sort -nt: +2 /etc/passwd
```

`tee [-a] file`	send output to `stdout` and `file` `-a` — append output to `file`

Example:
```
ls | tee ls.out
```

`wc [-cwl] [file]`	count characters, words or lines in a file `-c` — count characters `-w` — count words `-l` — count lines

Multi-tasking

`cmd > cmd.out &`	Run **cmd** in background stdin is disconnected for jobs running in background
`nohup cmd > cmd.out &`	Protect background **cmd** from log out
`nice cmd`	Run **cmd** at a lower priority
`jobs`	Display jobs running under current session
`ps -ef`	Display all processes running on the system
`echo$$`	Displays process id number of current shell process
Ctrl + z	Suspend a foreground job
`bg %#`	Put job number # in background
`fg %#`	Put job number # in foreground
`kill PID`	Terminate job with process identifier `PID`
`kill -s SIGNAME PID`	Send signal `SIGNAME` to `PID`
`trap cmd #`	Trap signal # and execute **cmd**, when signal occurs
`stty -a`	Display terminal settings and key mappings
Ctrl + c	Send interrupt to foreground process (signal 2)
Ctrl + \	Send quit to foreground process (signal 3)

© 2003 Hewlett-Packard Development Company, L.P.

Branching

`if` `    cmd(s)`	*if RETURN VALUE of LAST cmd is true do cmds following then* *if RETURN VALUE of LAST cmd is false do cmds following else*

`then`
 `    cmdtrue(s)`
 `else`
 `    cmdfalse(s)`
 `fi`	

`case $vara in`
 `    pat1) cmdsa`
 `          ;;`
 `    pat2) cmdsb`
 `          ;;`
 `       *) cmds default`
 `          ;;`
 `esac`
	compare value of vara to patterns
	 execute commands that follow matching pattern

Looping

`while cmd(s)`
 `do`
 `    cmdtrue(s)`
 `done`
	while RETURN VALUE of LAST cmd is true do cmds following do

`until cmd(s)`
 `do`
 `    cmdfalse(s)`
 `done`
	until RETURN VALUE of LAST cmd is true do cmds following do

`for vara in a b c d e`
 `do`
 `    cmd(s)`
 `done`
	assign vara to each item in list, do cmds for each item in list

Common POSIX Shell Environment Variables

#	The number of arguments supplied to a shell script
*****	All of the arguments supplied to a shell script
?	The return code of the last executed command
$	The PID of the last invoked shell

© 2003 Hewlett-Packard Development Company, L.P.

COLUMNS	Defines the width of the edit window for shell edit modes
EDITOR	Defines the edit mode to be used for command stack. Associated with `set -o vi`
ENV	script executed when a new Korn shell is invoked. Usually set to `.kshrc`
FCEDIT	Defines the editor that will be invoked from command stack
IFS	Internal Field Separators, usually a space, tab and newline, which separate commands and input for `read`
HISTFILE	The path of the file used to store the command history. The default is `.sh_history`
HISTSIZE	The number of saved commands accessible by the shell. The default is 128
HOME	Your login directory. The default for the cd command
LINES	Defines the column length of the edit window for printing lists
PATH	The directories to search to find executable programs
PS1	The primary prompt. The default is $
PS2	The secondary prompt. The default is >
PWD	The present working directory, set by the last `cd` command
OLDPWD	The previous working directory, set before the last `cd` command. Accessed with `cd -`
SHELL	The path of the program for the current shell
TERM	The model of the terminal being used
TMOUT	If this variable has a value greater than 0, the shell will terminate if this amount of time elapses before a command or Return is entered
TZ	Defines the time zone to be used for displaying the time and date

© 2003 Hewlett-Packard Development Company, L.P.

VISUAL	Defines the edit mode to be used for command stack. Associated with `set -o vi`

The variables, listed above, can also be used in the Linux bash shell and Linux/HP-UX Korn shell environments. Some of the variables listed may not be set by default, however, in a user's login shell environment unless the appropriate login controls have been applied by a system administrator.

© 2003 Hewlett-Packard Development Company, L.P.

© 2003 Hewlett-Packard Development Company, L.P.