HEWLETT® PACKARD

HP Computer Systems
Training Course

Fundamentals of the UNIX System

Instructor Guide

Version G.02
51434S Instructor
Printed in USA 7/99

Notice

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD PROVIDES THIS MATERIAL “AS IS” AND MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HEWLETT-PACKARD SHALL NOT BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS IN CONNECTION WITH THE FURNISHING, PERFORMANCE OR USE OF THIS MATERIAL WHETHER BASED ON WARRANTY, CONTRACT, OR OTHER LEGAL THEORY).

Some states do not allow the exclusion of implied warranties or the limitations or exclusion of liability for incidental or consequential damages, so the above limitations and exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

OSF, OSF/1, OSF/Motif, Motif, and Open Software Foundation are trademarks of the Open Software Foundation in the U.S. and other countries.

UNIX® is a registered trademark of The Open Group.

X/Open is a trademark of X/Open Company Limited in the UK and other Countries.

HP Education
100 Mayfield Avenue
Mountain View, CA 94043 U.S.A.

Contents

Module 2 — Logging In and General Orientation

Module 3 — Using CDE

Module 4 — Navigating the File System

Module 5 — Managing Files

Module 6 — File Permissions and Access

Module 9 — File Name Generation

Module 10 — Quoting

Module 11 — Input and Output Redirection

Module 12 — Pipes

Module 16 — Introduction to Shell Programming

Module 17 — Shell Programming — Branches

Module 18 — Shell Programming — Loops

Figures

Tables

Overview

Course Description

This course is designed to be the first course in the UNIX® curriculum presented by Hewlett-Packard. It is intended to give anyone (system administrators, programmers, and general users) a general introduction 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) or any other UNIX-based operating system.

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.
- 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.

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.

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.
- 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.

Reference Documentation

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

Notes to the Instructor

Reporting Errors in This Course

The Worldwide HP Education Development Team wants to provide you with accurate and up-to-date course materials. To do this better, we need your help. Please identify and report errors in the course materials (slides, student workbook, instructor guide, and labs) and submit them for correction through our on-line defect tracking system, SETI.

SETI (the Search for Errors, Typos, and Inconsistencies) is available via the World Wide Web and has been created to improve the quality of course materials by collecting defect information from instructors. All information you enter into SETI is given a status and can be viewed by all instructors around the world.

Bug Fixes

Some of the most irritating errors are among the easiest to fix:

- spelling
- grammar
- formatting
- technical inaccuracies
- source files that do not match output
- source files with tag errors
- missing source files

We will fix such reported errors on a tight turnaround time, triggered by the priorities of our team management. Other changes are more complex to implement:

Enhancements

- reorganization of content
- change in course design, audience, and course length
- addition of content (labs, examples, notes, topics, and so forth)

Requested design changes or course enhancements like these will be documented and considered for the next update of the course.

How to Report Errors

Errors should be submitted to the SETI web page:

http://hppsda.mayfield.hp.com/cgi-bin/seti

You can also locate the SETI web page from the WW HP Education web site at **http://hppsda.mayfield.hp.com.** Click on the link to Course Material and scroll down to the bottom of the Course Material page. Click on Course Defect Tracking System—SETI.

From this WWW location, you can:

- List the courses currently being tracked by SETI.
- Retrieve a report of all defects submitted against a particular version and release of any course being tracked by SETI.
- Submit a defect report for any course. (If the course is not currently being tracked, your defect report will initiate such tracking automatically.)
- Submit a request to modify the SETI tracking system .
- Obtain on-line help and guidelines on the use of SETI.

Please be prepared to identify and characterize the error in detail:

- source material
- course number
- version letter
- release number
- module number
- page number
- paragraph number or slide number
- instructions for the correction (for example, change "....." to ".....")
- characterize severity of defect

For source file errors, identify the file name and line number where the error occurs.

Orientation and Philosophy

This course is designed to give a general overview of the UNIX system. It covers basic concepts such as command syntax, commonly used commands, and basic shell programming techniques.

HP 51434S is a 5-day lecture/lab course targeting all users of the UNIX operating system, and provides preparation for the entire curriculum of system administration and software development courses. It provides an in-depth coverage of the UNIX operating system and shell programming, based upon HP-UX Releases 10.xx and 11.00.

Operating system features and functions are referred to as UNIX, unless they are specific

to HP-UX.

Additional Materials Provided

At the end of this overview you will find some supplementary documents that you may wish to copy and hand out as needed, or use for your own reference:

HP-UX to VMS Command Mapping	A simple mapping of some common VMS commands to their HP-UX counterparts. No claim is made as to the accuracy of this material. It was provided by a student in one of our classes. You may have someone in your class who comes from a VMS background, and you may find this material helpful. You should use it at your discretion.
Lab Notes for MPE Users	These are additional notes and explanations for students who have worked with the HP 3000. The goal of this information is to relate HP-UX to the MPE user's past experience by highlighting similarities and differences between HP-UX and MPE. This should allow MPE users to draw on their prior knowledge to develop HP-UX skills more quickly and competently, and reduce areas of confusion.
Phonebook Project	An exercise that can be assigned or recommended to advanced students who are looking for a more challenging exercise. Copy the six pages and hand out (perhaps only to select students) after the appropriate modules. More detailed directions accompany the project itself.

A Note About the Labs

In many modules there are more lab exercises than can be completed in the allotted time. The instructor notes for each lab indicate which exercises are appropriate for beginning, intermediate, and advanced user levels. You should assign exercises based on the students' knowledge and ability. It is not expected that every student complete every exercise.

A continuing lab (the phonebook project) is provided as part of these Instructor Notes. You may choose whether or not to use it. It follows the sequence of the topics as they are presented in the course. If you teach the modules in a different sequence or omit some modules, this lab may not function properly. Alternatively, you can provide students with the solutions to portions of the lab they may not have received lecture and notes for.

Appendices and Optional Modules

There is one appendix to the course:

Commands Quick Reference Guide	A categorized listing of the commands and structures presented in this class.

Sample Schedule

The following is a sample daily schedule. Times may vary depending on the experience level and interests of the students. You should vary this schedule as necessary and as students' needs and interests dictate.

You may wish to survey your students to decide which if any of the appendixes you will cover, and at what point in the class.

Table 1.

Module	Approximate Times	
	Lecture	Lab
Day One		
Module 1 — Introduction to UNIX	30 min	0min
Module 2— Logging In and General Orientation	45 min	45 min
Module 3— Using CDE	45 min	45min
Module 4— Navigating the File System	45 min	45min
Module 5— Managing Files	45 min	30min
Module 6— File Permissions and Access	45 min	0 min
Day Two		
Module 6 — File Permissions and Access (continued)	0 min	30min
Module 7 — Shell Basics	45 min	30min
Module 8 — Shell Advanced Features	90 min	45min
Module 9 — File Name Generation	30 min	30min
Module 10 — Quoting	60 min	0 min
Day Three		
Module 10 — Quoting (continued)	0 min	45min
Module 11 — Input and Output Redirection	30 min	30min
Module 12 — Pipes	60 min	45min
Module13 — Using Network Services	30 min	30min
Module 14 — Introduction to the `vi` Editor	60 min	60min

Table 2.

Module	Approximate Times	
	Lecture	Lab
Day Four		
Module 15 — Process Control	30 min	30min
Module 16— Introduction to Shell Programming	60 min	75min
Module 17— Shell Programming - Branches	60 min	45min
Day Five		
Module 18 — Shell Programming — Loops	60 min	90min
Module 19 — Offline File Storage	45 min	30min
Appendix A — Commands Quick Reference Guide		

Instructor Profile and Prerequisites

The instructor should have completed the student prerequisites and this course. Ideally, the instructor should team teach this course with an experienced instructor before teaching it alone.

Classroom Setup

There are many places in the course where it would be to the instructor's advantage to be able to demonstrate certain tasks. If it is possible, we recommend that you set up a demonstration system.

Equipment List

Each student or pair of students should have a terminal to log in on an HP-UX system. An available tape drive is desirable but not absolutely required.

Software

Release 10.xx or 11.00 of HP-UX operating system.

Preparation Tasks

Lab Setup

Log in as **root** and create a directory called **labs**. Restore the lab tape to the **labs** directory. The lab tape is in **tar** format. Use the command

```
tar xvf /dev/devicename
```

Read the README file and follow the instructions to set up the student accounts.

```
                README
FUNDAMENTALS OF THE UNIX SYSTEM (51434S)
      UNIX SYSTEM BASICS I (51489S)
      UNIX SYSTEM BASICS II (H2572S)
```

The lab tape for these courses contains the following:

README	This file. A text file explaining the installation procedure
lab_setup*	a shell script to install user accounts
students	a text file containing user account names
lab_files/	a master directory containing all required files for this course
restore_dirs*	a shell script to delete lab files from user accounts
infinite.c	a C Program that will be compiled and loaded into the lab_files directory. This program will be required for the exercises in the Multi-tasking module.

Create Student Accounts

Before you install the lab files for these courses, you should have created entries for your student accounts in **/etc/passwd** and **/etc/group**.

Each student account must be assigned to at least two groups. The default primary group is "class", and the secondary group is "class2". The installation script will verify that there is an entry for each student in **/etc/passwd**, and two entries in **/etc/group**. You can modify the default group designation by modifying the variable *PRIM_GROUP* in **lab_setup**. All student accounts should use the Posix shell (**/usr/bin/sh**).

Default student names are stored in the file **students**. This allows you to easily customize the installation procedure for the names that you select for the accounts at your local site.

To create your user accounts:

1. Create an entry for each user in **/etc/passwd**.
2. Each user's primary group should be **class**.
3. Each user's login shell should be **/usr/bin/sh**.
4. Create an entry in **/etc/group** for each user in group **class** and group **class2**.
5. Modify **students** if your user account names are not **user1, user2, ...**.
6. Modify the *PARENT_DIR* variable in **lab_setup** if your user accounts will not be under **/users**.

Modify lab_files.rhosts

Edit the file **.rhosts** in the directory **lab_files** to contain the names of other systems on your network from which you will allow users to log on remotely.

Install Student Files

The directory **lab_files** contains all of the files that will be used throughout the presentation of the Fundamentals of the UNIX System class, and the UNIX System Basics I and UNIX System Basics II classes. They are set up as a directory in case you would like to add any local files to your student accounts. All you need to do is copy any customized files to the **lab_files** directory, and they will be automatically installed for you the next time you run **lab_setup**.

Each student account will be installed under the **/home** directory. If you would like your student accounts under some other directory, modify the *PARENT_DIR* variable in **lab_setup**.

To install student accounts:

Run **lab_setup**. This script will verify that there is an entry in **/etc/passwd** and **/etc/group** for each of your user accounts. It will create the *HOME* directory for each student account, if it does not already exist, and then copy all of the **lab_files** to each student directory. The default permissions for the files do NOT include write access for group

and others. This script will call **restore_dirs** if the directories have not been cleared out from your last class.

Setting the TERM type

The **.profile** file installed under each student account uses:

```
eval `tset -s -Q -h`
```

to determine the *TERM* type. You will need to set up your **/etc/ttytype** file, or modify the master **lab_files/.profile** according to your local configuration to guarantee that *TERM* is properly defined when your students log in.

In order to run the **rwho** and **ruptime** commands in the networking appendix you will need to configure the rwho daemon (rwhod). The rwho daemon is initiated from **/etc/rc.config.d.netdaemons**.

Post-Class Cleanup

The script "restore_dirs" can be executed when the class is over to clear out the directories associated with your student accounts. It will delete all files, and then install each student account with only the DOT FILES.

Materials List

Classroom Materials

- Overhead projector
- White board
- Flip chart

Library List

- *HP-UX Reference* (3 vols), P/N B2355-90033
- *Shells: User's Guide*, P/N B2355-90046

Supplementary Information

The following information can be copied and used as handouts in class.

UNIX to VMS Command Mapping

Here are some basic UNIX commands and their corresponding VMS commands:

UNIX	VMS
ls	dir
cp	copy
cd	set default
rm	delete
mv	rename
cat	type
more	type/page
pwd	show default
ps	show process or show system
man	help
mkdir, **rmdir**	create/dir, delete __.dir
kill	stop or delete/entry
tail/head, **grep**	search
chmod	set protection
wc **diff**	difference
file **f77,cc**	fortran, cc
logout/exit/^d	logout
df /bdf	show device
ln	assign or define (sort of)
du	dir/size/total
who/w	show users, show sys (sort of)
set/unset	local symbol assignment/delete/sum
stty **echo**	write sys$output

passwd	set password
date	show time

Phonebook Project

The following exercise is a phonebook project. It can be assigned/recommended to advanced students who are looking for a more challenging exercise. Copy the following six pages and hand out (perhaps only to select students) after the appropriate modules, as follows:

After Module: *Shell Advanced Features*	Phonebook Project — Part 1
After Module: *Pipes*	Phonebook Project — Part 2
After Module: *Introduction to Shell Programming*	Phonebook Project — Part 3
After Module: *Shell Programming—Branches*	Phonebook Project — Part 4
After Module: *Shell Programming—Loops*	Phonebook Project — Part 5

Phonebook Project — Part 1

The Phonebook Project is intended to allow you to develop a series of shell programs that will implement an online phone book. As you continue through the course, you can work on this project as time permits. As new programming techniques are presented, there will be additional assignments to enhance the functionality of your online phone book.

The list of names and phone numbers will be stored in a text file. Each line will be a single entry consisting of:

```
Lastname:Firstname:area code:phone number:address
```

The : will be the separator between the different fields in a line. You can go into more detail if you like by splitting the address into street, city, state, and zip fields. It all depends on how much information and flexibility you want.

The following summary provides the minimal functionality that your program should perform. It is recommended that each function be supported by a corresponding shell program. This modular approach will make it easier to implement the separate features and add functionality.

Minimal Functions:

1. Add—Add a name, address, and phone number to the phonebook file.
2. Lookup—Prompt the user for a name, and then print out the phone number and address associated with the name.
3. List—List the contents of the phonebook in alphabetical order.
4. Delete—Prompt the user for a name and then remove that entry from the phonebook file.
5. Change—Prompt the user for a name and new phone number or address. Update the phonebook file with the new information.
6. Report—Print a report with headings that lists the information requested by the user.
7. User friendly interface—In order for your users to not have to remember all the names of the phonebook commands, you should prompt them with a menu from which they can select the appropriate option.

What to Do Now?

1. Use an editor to create a sample phonebook file.
2. Create a shell script that will add an entry to your phonebook file. HINT: assign the new entry to an environment variable, and append the value of this variable to your phonebook file.
3. Create a shell script that will search the phonebook file for an entry associated with a specific name. HINT: assign the name that you are interested in to an environment variable, and look for the entry that contains the value of this variable.
4. Create a shell script that will list the contents of the book in alphabetical order.

5. Create a shell script that will delete an entry from the phonebook file. HINT: assign the name that you are interested in deleting to an environment variable.

6. Create a shell script that will allow an entry to be changed. At this point, you will need to replace the entire line.

7. Create a shell script that will provide a report capability. For example, all listings whose phone number is in a specific area code, or a specific city. Format the output and include headers.

Phonebook Project — Part 2

Update the filters you implemented in your shell programs to use pipelines instead of file redirection, where appropriate.

Phonebook Project — Part 3

Modify the following shell programs developed for your phonebook project.

1. *Add a name:* The shell program should prompt the user for the name, area code, phone, and address information. After this information is input, it should be appended to the phonebook file.

2. *Lookup a name:* The shell program should prompt the user for the name of the party on which he or she wants information. Once the name has been entered, the associated data should be displayed.

3. *Delete a name:* The shell program should prompt the user for the name of the entry that should be deleted from the phonebook file. After the name is entered, the corresponding data should be deleted.

4. *Change a name:* The shell program should prompt the user for the name of the entry that needs to be modified. The old entry should be deleted, and then the shell program should prompt the user for the new information. This could combine the Delete and Add functions.

Phonebook Project — Part 4

Enhance your phonebook project to

1. Create a menu interface that prompts the user for which option he or she would like to execute. If you have been developing each of the phonebook functions in a separate shell program, all you will need to do is create the menu interface, prompt the user for input, and set up each menu option to invoke the shell programs that you have been developing.

2. Include error checking. For example, verify that a requested name exists in the phonebook. If it does not, provide an informative error message to the user.

Phonebook Project — Part 5

Enhance your phonebook project so that the menu you created in the last module is repeatedly displayed until the user selects the option to quit.

Lab Notes for MPE Users

These are additional notes and explanations for students who have worked with the HP 3000. The goal of this information is to relate HP-UX to the MPE user's past experience by highlighting similarities and differences between HP-UX and MPE. This should allow MPE users to draw on their prior knowledge to develop HP-UX skills more quickly and competently, and reduce areas of confusion. This information is broken up into several sections:

- Logging In
- Commands
- File System
- Permissions
- Text Editing
- Shell Programming
- Batch Processing

These notes are designed as a self study to be read independently by the student either before or after appropriate labs.

Logging In

On MPE, a login ID consists of two required pieces of information: an MPE user name and an MPE account name. In addition, a third identifier, the MPE group name, can optionally be used to specify which MPE group the user logs into. The group name is required if the user has no home group. On HP-UX, a login ID consists of only a user name. In MPE, the user, account, and group name information is stored in the MPE System Directory. This information is accessible through **LISTDIR5** on MPE V or, on MPE/iX,**:LISTUSER**, **:LISTACCT**; **:LISTGROUP**; and is maintained through **:NEWUSER**, **:ALTUSER**, **:PURGEUSER**, **:NEWACCT**, **:ALTACCT**, **:PURGEACCT**, **:NEWGROUP**, **:ALTGROUP**, **:PURGEGROUP**. In HP-UX, all information related to a user name is kept in the file**/etc/passwd** and can be maintained by editing this ASCII file. HP-UX also provides **sam**, a menu-driven system administrator utility which edits this file for you. The user names and associated information are accessible simply by looking at the contents of this file. That is **cat /etc/passwd** . Section four of the HP-UX reference manual documents the layout of **/etc/passwd**. Each line contains a separate user name. Several fields, separated by colons, are stored as follows:

```
sally:tekXeRorZ0qXQ:201:20::/home/planets/sally:/usr/bin/sh
```

- login name (**sally** from example above)

  User name. All user names as well as file names and commands are case sensitive in HP-UX.

- encrypted password (**tekXeRorZ0qXQ** from example)

  In MPE, a user could be prompted for as many as three different passwords at login time, for the account password, user password, and group password. At most one password is

required for HP-UX. In HP-UX, all passwords are stored in an encrypted format; therefore, no one, not even super-user, can read these passwords. The super-user can, however, edit **/etc/passwd** to remove an encrypted password. On HP-UX, users maintain their own passwords through the **passwd** command.

- numerical user ID (**201** from example)

  Each user name has associated with it a user ID number. A 0 signifies super-user privilege. Super-user privilege can be assigned to one or more HP-UX users. Super-user privilege allows access to special administrative tools and overrides many operating system controls. MPE capabilities can be assigned selectively to MPE users. The MPE user **MANAGER.SYS** has full MPE capabilities. So too, the HP-UX user **root** has super-user privilege. Having super-user privilege on HP-UX is similar to having a combination of SM, PM, and OP capabilities on MPE.

- group ID number (**20** from example)

  This is the ID number of the group this user initially belongs to. MPE and HP-UX use the term group. In MPE, a group is a collection of files. We will see that on HP-UX a collection of files is called a directory.

  In HP-UX a group is a collection of users. Groups are used for security purposes. Each file is owned by a group. A group, that is all its members, can be assigned permission to access the files the group owns. And access can be denied to other users of the system who are not members of the group.

  Each user is permitted to be a member of one or more groups (one group at a time).

  Let's say we have a group named **payroll** with the user names **sally**, **hank**, and **sue** permitted to be members. The file **/employee/pay** is owned by the **payroll** group. I allow only members of the **payroll** group read access to **/employee/pay**.

  The user **sally** can also be permitted to be a member of a group named **benefits**. Only members of the **benefits** group are permitted read access to the file **/employee/ben**. **/employee/ben** is owned by the **benefits** group.

  Let us say **sally** is initially assigned to the **payroll** group. That is, the fourth field in **/etc/passwd** for **sally** has the **payroll** group's ID number.

  So when sally issues the command **id** she would see her current user ID is **sally** and her current group ID is **payroll**. At this point, **sally** is permitted access to the file **/employee/pay** but is denied access to the file **/employee/ben**. To switch to the **benefits** group and gain access to the file **/employee/ben**, **sally** would issue the command **newgrp benefits**.

  Now the **id** command will reflect that her user ID is still **sally** but her group ID is now **benefits**, thus now allowing her access to the file called **/employee/ben**.

  The list of user-group relations is contained in the file, **/etc/group**. Specifically, each line contains a group name, an optional password, a group ID number, and a list of user names permitted as members of the group. This information is accessible by looking at the contents of this ASCII file and is maintained by editing it with any text editor such as **vi** . The section of the file we described would look like

```
payroll::20:sally,hank,sue
benefits::21:sally
```

- reserved field (empty in the example)

  Can be used for personal identification such as user's full name, office location, extension, home phone.

- initial working directory (**/home/planets/sally** in the example)

  This is similar in concept to a home group in MPE. When a user logs on, he is initially placed in this home directory. Relative path names will be evaluated relative to this directory. On MPE/iX, the **:CHGROUP** could be used to switch to other groups of files without logging off. On HP-UX, a user can issue a **cd** (change directory) command to switch directories. We will see that the user does need appropriate permissions (file security) to a directory in order to switch to it.

- shell program (**/usr/bin/ksh** in example above)

  This is the shell program or command interpreter that will be run for this user upon login. A command interpreter is the part of the operating system that reads the user's commands and initiates execution of these commands. MPE provides only one command interpreter, but on HP-UX there are several command interpreters (or shells) to choose from. The primary shells available are

  — Bourne shell (**/bin/sh**) oldest shell

  — C shell (**/bin/csh**) Berkeley shell

  — Korn shell or K shell (**/bin/ksh**) newer shell, incorporates best of C shell extensions and maintains compatibility with Bourne shell.

  — POSIX shell (**/usr/bin/sh**), default shell, compliant with POSIX.2 standard, features are most like the Korn shell.

The HP 3000 system manager must specify in the configuration the type of terminal each user is working with. That means MPE knows what type of terminal you are using before you log in.

HP-UX, on the other hand, determines what type of terminal you are using at the time you log in. UNIX was originally designed to run on virtually any hardware, using a variety of terminals. Some older terminals supported on HP-UX don't have backspace keys and have only uppercase. For these reasons, terminals work in a special way at login time on HP-UX. In order to allow users to correct errors during login in a consistent manner regardless of what terminal they are working with, the backspace key is not used. Instead of backspace, a single character is erased with # and the entire line is erased with @. Also, if uppercase is entered at login time, HP-UX assumes you are using a terminal that is strictly uppercase and all characters are shifted to lowercase before being executed. To avoid this, user names should always be entered in lowercase when logging in.

Commands

Many HP-UX commands have MPE commands that are similar.

HP-UX	MPE
touch	build
cat	fcopy (to screen)
more	print (iX)
ls	listf
cp	fcopy, copy (iX)
mv	rename
rm	purge
mkdir	newgroup
rmdir	purgegroup
who	showjob job=@s (sessions only)
whoami	showme
ps	showjob, showproc (iX)
mail	HPDESKMANAGER (the product)
pwd	showme (to see the group name)
id	showme (to see the user name)
write	tell
cd	chgroup (iX)
chmod (on a file)	altsec
chmod (on a directory)	altgroup access=, altacct access=
ln	file equation
chown	no equivalent
vi	editor
sort	sort
nice	job ;pri=es
kill	abortjob
if	if
tar	store/restore
man	help
fc -l (Korn shell)	listredo (iX)
r (Korn shell)	do (iX, redo without editing)
read	input (iX)
nohup (with &as suffix)	stream
at (no &required)	stream ;at=

The **man** command is similar to MPE **:HELP**. It allows online access to the HP-UX reference manual. Many MPE users are familiar with UDCs (or command files) that are abbreviations for commands. Many MPE users create UDCs for longer command lines in an effort to save keystrokes. For example:

```
SJ = SHOWJOB
SJJ = SHOWJOB JOB=@J
SJS = SHOWJOB JOB=@S
```

HP-UX was designed with similar time savers already built in, that is, ID = copy, mv = move, cd = change directory, pwd = present working directory, man = manual, and so forth. On HP-UX you must use these abbreviations.

Many UNIX commands can operate in a slightly different way by providing options. The man command contains a -k option which allows an online reference manual search based on a key word.

The MPE **:LISTREDO** and **:REDO** commands provide access to commands the MPE user has previously executed. In the HP-UX Korn shell, (**/usr/bin/sh**), the escape key followed by the k key brings up the last command the HP-UX user just executed. At this point the user can use **vi** commands to edit the line displayed before re-executing this line. By entering the k key over and over the user can retrieve earlier commands that had been executed and can edit them with **vi**.

File System

On MPE, all files reside within a MPE group. Each group resides within a MPE account. So all files can be identified by **filename.groupname.accountname**. The lists of valid accounts, groups and files are all located in the MPE system directory.

On HP-UX, the term *directory* is used to mean something quite different from the MPE system directory. A directory on HP-UX is simply a collection of files; very much like a MPE group is a collection of files. There are, however, some differences. MPE groups contain only files. I cannot have one MPE group contained within second MPE group. On HP-UX, however, I can have one directory stored within another directory. That is, a directory can contain files or directories. Looking back, MPE could be thought of as a two-level directory file system where the *upper-level* directory is the account name and the *lower-level* directory (or subdirectory) is the group name.

HP-UX has a parent directory for all files and directories on the system called root, written as **/**; therefore, all files and directories are contained within **/**.

On MPE a fully qualified file name is **filename.groupname.accountname**.

On HP-UX the absolute path name is similar to this fully qualified file name. All directories that the file is located within are specified. On MPE, the file name is listed first, followed by the subdirectory (group), followed by the parent directory (account). On HP-UX the sequence is reversed. The absolute path name always starts with the highest level directory containing the file which will always be **/**. The **/** is followed by all other directories containing the file with the higher level directories preceding the lower-level directories. All intermediary directory names are suffixed with a **/**. The last piece of information is the file name.

So **/etc/passwd** refers to the absolute path name for the file **passwd** which is located within the directory **etc**. **etc** is located within the directory **/** (root).

The file **/home/planets/earth/funfile** refers to the file **funfile** (notice the HP-UX file names allow special characters like a . and can be longer than eight characters) which is located within the directory **earth**. Remember HP-UX is case sensitive; therefore, **Earth** is not the same directory as **earth**. **earth** is located within the directory **planets**; **planets** is located within the directory **home**; **home** is located within the directory **/** .

On MPE, if the file we want to access resides within our logon group and account, we do not need to fully qualify the file name to access it. So if we were logged into the group **DB** within the account **MFG** and we wanted to access the file **TRS** (full file name is **TRS.DB.MFG**), we could specify simply **TRS**.

Similarly, on HP-UX, we can specify a relative path name as opposed to an absolute path name. We do this by specifying only the portion of the path name that is "below" our present working directory.

So if our present working directory were **/home/planets/earth** and we wanted to view the contents of the file **/home/planets/earth/funfile** with the cat command, we could simply enter

```
cat funfile
```

as opposed to writing out

```
cat /home/planets/earth/funfile
```

which would also work.

Permissions

On MPE, file security includes provisions to differentiate between five types of file access: read, write, lock, append and execute. Also, the ability to create new files is controlled by assigning save access to a group.

Additionally, on MPE, in order to be able to access the file the user must have the appropriate access at the account, group and file level.

Finally, on MPE, users are segregated into categories, Account Member, Group User, Any, Account Librarian, Group Librarian, and Creator.

On HP-UX, read, write, and execute permissions may be assigned to both files and directories. HP-UX segregates users into three mutually exclusive categories:

- user (owner of the file or directory). Every file and directory has an owner. Ownership can be changed with the **chown** command.
- group (every file and directory is owned by a group). All logons who are currently members of the group that owns the file fall in this category. Group ownership of a file can be changed with the **chgrp** command.
- other (anyone else on the system).

The HP-UX user is similar to MPE creator. HP-UX group membership is initially based on the group ID entry in the **/etc/passwd** file and can be changed with the newgrp command.

Read, write, and execute mean something special when applied to directories.

Read access to a directory means that we can use the ls command to get a list of files in that directory. On MPE the LISTF command is always available to get a list of files from any group/account and cannot be disabled through file security.

Write access to a directory means that we can add or remove files from the directory. On MPE, save access to a group allows us to add files to the group. On MPE, the ability to purge files is strictly limited to creator and SM capability and cannot be further controlled with file security.

On MPE/iX we need to know the group password to issue the **CHGROUP** command to switch to a different group. In order to use the HP-UX **cd** command to switch to another directory we need execute permission to the directory we are switching to.

Similar to MPE, having read access to a file at the file level is not sufficient access for us to read the file. We also need appropriate access to the directory in which the file resides and to all additional directories included in the path name of the file.

So, in order to access a file in any way (read, write, or execute), not only do we need appropriate access to the file; we also need search access (execute access) to that file's parent directory and to any additional directories included in that file's path name.

For example, let's say our present working directory is **/home** and we are trying to read the file **/home/planets/earth/funfile**. In order for our command

```
cat planets/earth/funfile
```

to succeed, we need read access to the file **funfile** and search access (that is, execute access) to the directories **planets** and **earth**.

Text Editing

Many editors are available on both MPE and UNIX. **EDITOR** is available on every MPE system and for that reason is taught in introductory MPE classes. **vi** is taught in this class because it is so widely available on UNIX systems.

MPE **EDITOR** is a line editor. **vi** works differently. **vi** relies heavily on command mode and input mode. In **EDITOR** the **add** command allows lines to be added to the text. Everything that is typed after entering the add command is included as contents of the file until you key **//** . In **vi** this is called input mode. In **EDITOR** you terminate *input mode* for the add command by typing **//**. In **vi** several commands put you into input mode. For example, a, i, o, and O all initiate input mode. In each case, everything typed from that point until the escape key is hit is added to the file. So the escape key in **vi** serves a similar purpose as the **//** serves for the add command in **EDITOR**.

Shell Programming

An MPE UDC catalog is a file that can contain multiple command definitions. For example,

```
purgeudc
file a=listout
purge l
run purgprog
****
buildfil
.
.
.
****
anotherudc
.
```

```
.
.
****
```

UNIX provides the ability to create our own commands through shell programs. Both the body of a UDC and the contents of a shell program are made up of operating system commands. One MPE UDC file can contain multiple command definitions as in the example above. An HP-UX shell program contains the definition of a single command. The command is executed by typing the name of the HP-UX file. Shell programs can use parameters just as UDCs can. The following UDC and shell programs both purge four files.

```
purgeudc f1,f2,f3,f4
purge !f1
purge !f2
purge !f3
purge !f4
*
purgeshellprogram f1 f2 f3 f4
rm $1
rm $2
rm $3
rm $4
```

In MPE several commands, including **:LISTF**, **:STORE**, and **:RESTORE**, allow for wildcards in file names. Here @ matches any pattern, ? matches any single character, and # matches any digit. HP-UX has a similar file name matching facility which works on all commands, where * matches any pattern (* = @), ? matches any single character (? = ?), and [ ]can be used to identify a character class to match a single position. So [0-9] matches any single digit.

MPE HP-UX	
`?### ?[0-9][0-9][0-9]`	*any character followed by three digits*
`@a@ *a*`	*a anywhere in filename*
`@t *t`	*ends with t*

Many features of UNIX such as redirection of command input and output, command variables, command files (shell programs), implied run, the PATH variable (called HPPATH), and character classes are part of MPE/iX.

The logon option of MPE UDCs allows a UDC to be executed automatically when the user logs into MPE. The **.profile** file in the user's home directory is executed automatically when a HP-UX user first logs in and provides similar functionality.

The nobreak option of an MPE UDC disables the break key. The HP-UX **trap** command can be used to provide the same functionality in a shell program.

Batch Processing

The **:STREAM** command in MPE can be used to initiate a batch job. The batch job logs on independent of the online user who issued the **STREAM** command. An HP-UX command that has an ampersand (**&**) at the end will run as a background process. This background process is similar to a batch job in that both allow the user to continue working interactively and both share the CPU with any online users.

The HP-UX background process is not independent of the online user. The MPE batch job sends all its default output (**$STDLIST**) to the printer. The HP-UX background process sends all its default output (**stdout**) to the user's screen. If the MPE user who streamed the job logs off, the job continues to execute. If the HP-UX user who initiated the background process logs off, the background process will abort.

The HP-UX **nohup** command allows the HP-UX background process to work more like an MPE job in that **nohup** processes will continue to execute after the online user logs off.

For example, in MPE

STREAM MYJOB

initiates a batch job.

In HP-UX

nohup myjob &

initiates the background process.

The HP-UX **at** command provides the ability to schedule jobs to run at a future time similar to the MPE command **STREAM ;AT=**. The background process will execute at the future time even if the user logs out.

In order to remove a job or session on MPE we would first issue a **:SHOWJOB** to determine the job or session number and then issue an **:ABORTJOB** to abort the job or session. The HP-UX user would first issue a **ps** command to determine the process id of the background process or online user and then issue a **kill** command to send the process a signal causing the process to abort. Signal number 9 is a *sure kill* and cannot be disabled by the **trap** command. For example in MPE,

```
SHOWJOB                    determine which job or session number to abort
ABORTJOB #J154             remove job number #J154 from the system
```

in HP-UX:

```
ps -ef                     determine which process id to abort
kill -9 16882              remove process id number 16882 by sending it signal 9
```

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.

Overview of Module 1

Audience

general user	General system users

Product Family Type

open sys	Open systems environment

Abstract

This module is designed to introduce the student to operating systems in general and to the UNIX system specifically. There is only general information in this module and there is no lab associated with it.

Time

Lab	0 minutes
Lecture	30 minutes

Instructor Profile

UX	General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

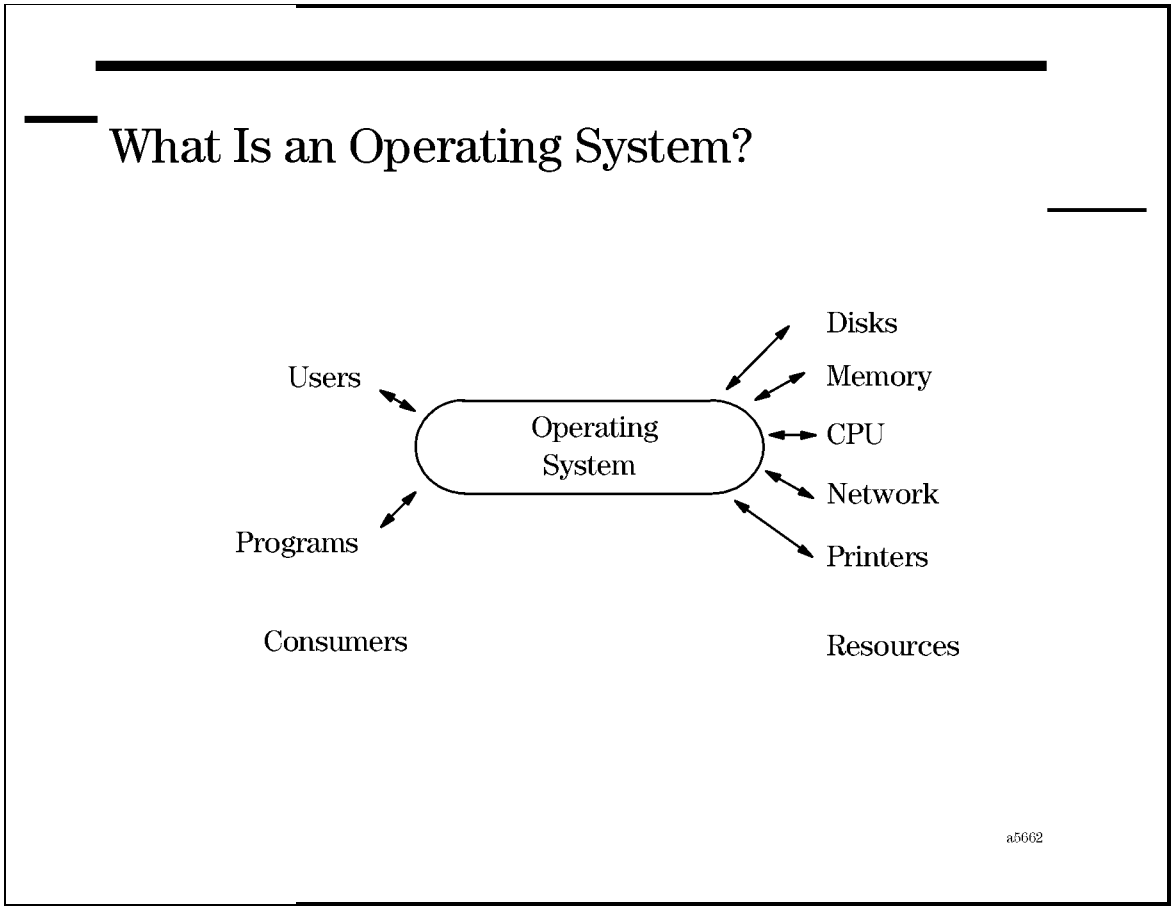
Language

USEnglish	US English

Trademarks

UNIX	UNIX® is a registered trademark of The Open Group in the U.S.A. and other countries.
OSF	OSF/Motif® is a registered trademark of the Open Software Foundation.
XWIN	X Windows™ is a trademark of the Massachusetts Institute of Technology.

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. 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

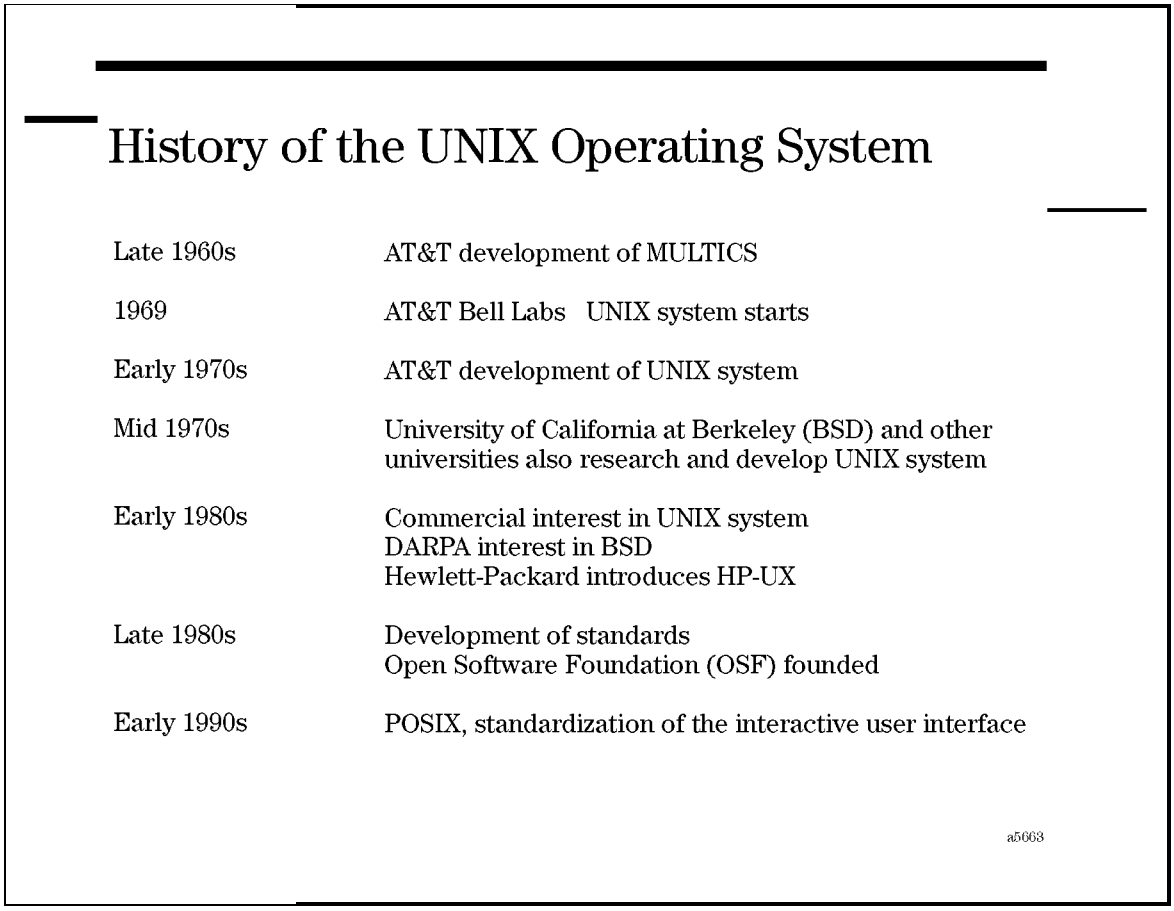
which road has access to the intersection, eventually giving all roads access through the intersection.

1-1. SLIDE: What Is an Operating System? Instructor Notes

Teaching Tips

Point out that the operating system acts as a traffic cop and resource allocator, regulating the flow of data and the use of resources to fulfill a variety of demands.

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



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 kilobytes for user programs, and 64 kilobytes 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:

- 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.

1-2. SLIDE: History of the UNIX Operating System — Instructor Notes

Teaching Tips

Briefly explain the history of the UNIX operating system.

Additional information is available in the student notes and the following text page for those students who are interested in reading more of the details related to the history of the UNIX operating system.

There are currently numerous implementations of the UNIX system, but most implementations originate from either AT&T's System V version or Berkeley's BSD version of the UNIX system. Since so many implementations have been developed, standards have become increasingly important. Standards will be briefly introduced later in this module.

The Open Software Foundation (OSF)

The goal of the Open Software Foundation (now called the Open Group) is to develop and license technologies to promote software portability, interoperability, and scalability. OSF develops their offerings through contributions of existing technologies and incorporating current standards (POSIX, XPG, ANSI, and so on).

The first offering, Motif, provided technology for user interface development, where consistent user interfaces could be developed across multiple hardware platforms. The second offering was the operating system, OSF/1 based on BSD 4.2 and System V Release 2 while incorporating the capabilities for advanced features such as multi-processors, the ability of a file system to span disks, dynamic kernel configuration, and more. Other development that OSF is currently researching include:

Distributed Computing Environment (DCE)	The capability to distribute the execution of a process among several networked processors.
Distributed Management Environment (DME)	The capability to centrally manage a heterogeneous network.
Architecture Neutral Distribution Format (ANDF)	The release of a common tape format that can be used across multiple hardware platforms.

For more information on OSF refer students to Hewlett-Packard's OSF seminar series:

- Introduction to Programming with OSF/Motif Widgets and the X Toolkit—H2595S
- The OSF Computing Environment Seminar—H2594S
- OSF DCE Application Programming—H5097S
- OSF DCE Internals—H5098S

- User Interface Design with OSF/Motif—H5095S

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 services. 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.

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.
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 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.

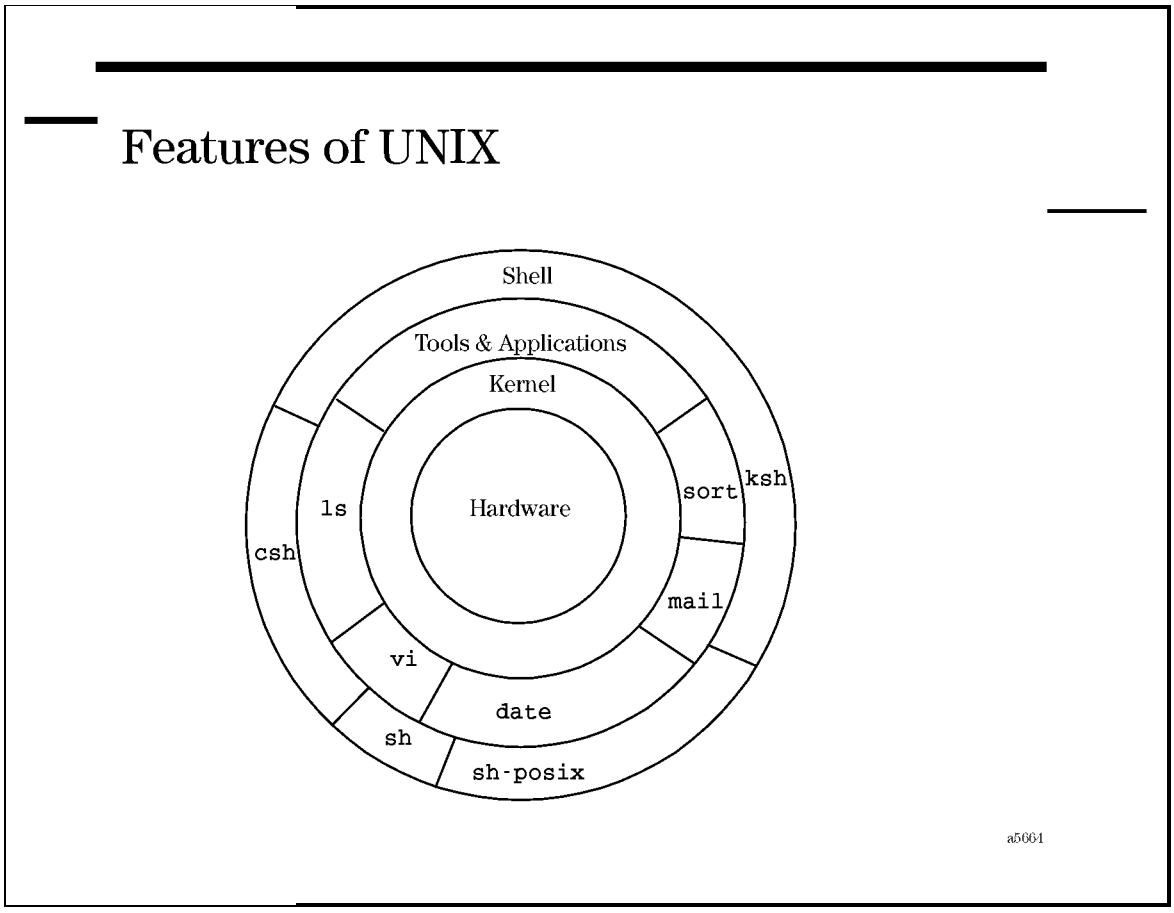
	XPG/3—support POSIX.1 and Common Application Environment, selects standards that will be incorporated for several aspects of the computing environment, not just the operating system interface to promote portability.
1990	SVID Issue 3—POSIX.1, FIPS 151-1 and C Standard.
1991	HP-UX 8.0—licensee of System V Release 3, SVID2 compliant, incorporating BSD4.2 and BSD4.3 extensions that have become defacto industry standards, incorporating POSIX-, FIPS-, XPG2-, and XPG3-compliant interfaces.
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.

1-3. TEXT PAGE: History of the UNIX Operating System — Instructor Notes

Purpose

This text page is provided for those students who are interested in reading a little more about the history of the UNIX operating system.

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.

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 the kernel can understand. Then the system will execute the command.

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.

- 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.

Table 1-1. Comparison of Shell Features

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

1-4. SLIDE: Features of UNIX — Instructor Notes

Key Points

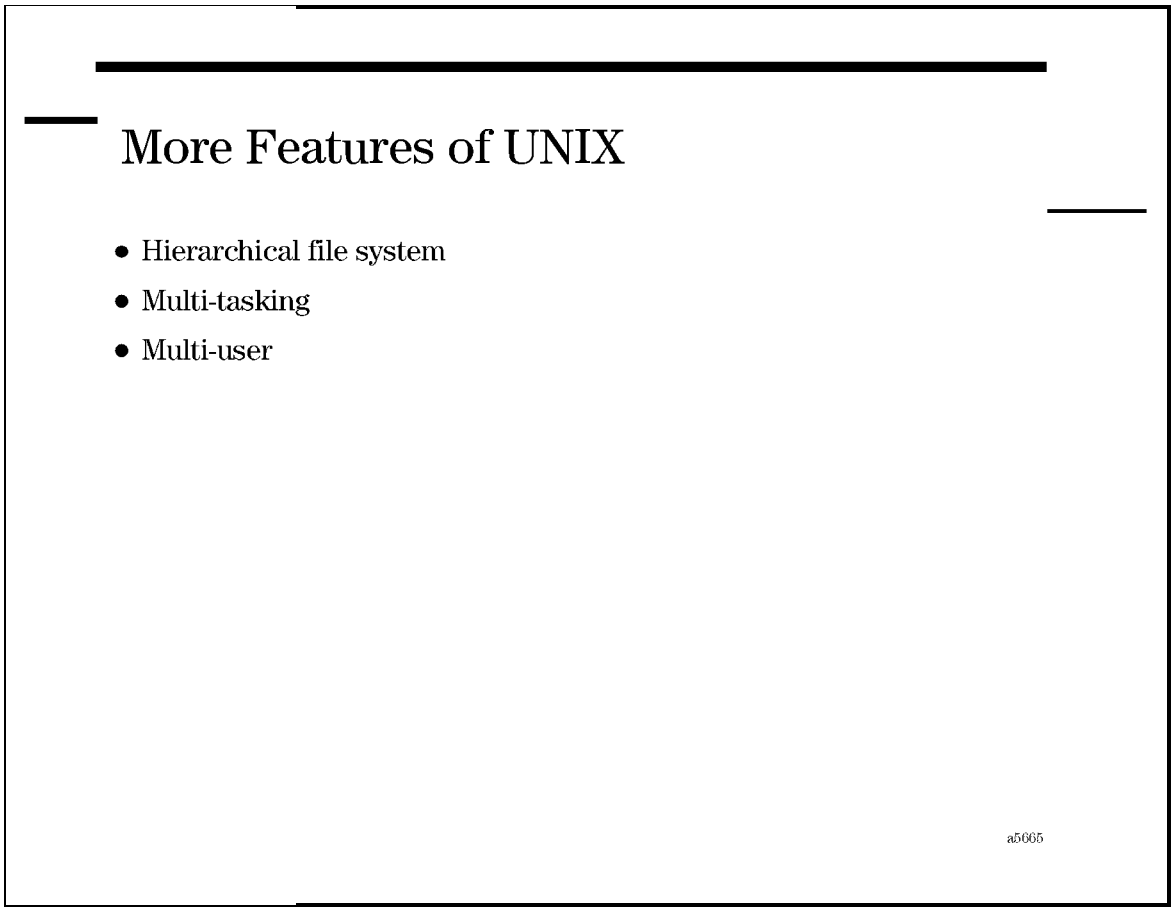
- The shell is a command interpreter.
- It also provides an extensive programmable interface.
- The shell is *not* built into the operating system.
- The shell could be replaced with another program.
- Many command interpreters are available.

Many students notice as they progress through the course that many commands do not share a common syntax or even style. The collection of the UNIX system utilities is largely evolutionary. Most early commands were terse with single letter options. But many of the early UNIX system enhancements and utilities were contributed, with little or no control over the consistency of syntax with other similar utilities. Therefore, you will find commands like

- **`find`** that has complete word options, instead of single letter options.
- **`cut`** and **`sort`** that have similar options with different identifiers such as -t and -d.
- Some commands use a hyphen (-) prior to the options, and some do not.

NOTE: The POSIX.2 standard requires that on a POSIX-compliant system, executing the command **`sh`** activates the POSIX shell (located in file **`/usr/bin/sh`** on HP-UX systems).

1-5. SLIDE: More Features of UNIX



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.

Multi-tasking

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 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.

Multi-user

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.

1-5. SLIDE: More Features of UNIX — Instructor Notes

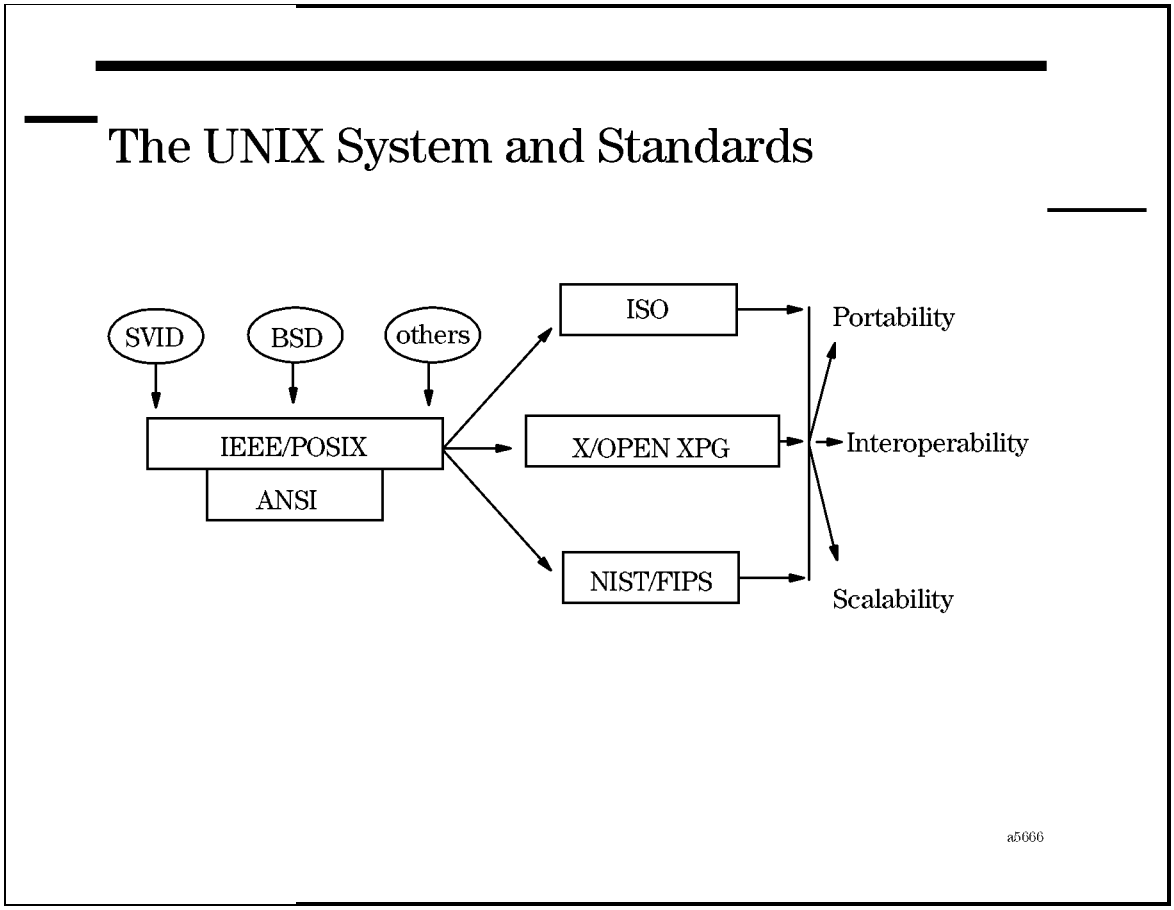
Key Points

- A file is a container for information and data.
- A directory is a container for files and other directories.
- The UNIX system provides a hierarchical file system.
- The UNIX system supports the execution of multiple processes simultaneously.
- The UNIX system supports multiple users to be logged in concurrently.

Teaching Tips

You might want to contrast the UNIX system's hierarchical file system against a flat file system that does not allow directories. Note that there is no limit to the number of subdirectories that can be created, and subdirectories are just containers for files; they are not tied into the user accounts structure of the system, though a hierarchy could be created that resembles the organization structure.

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.

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.

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

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

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.

1-6. SLIDE: The UNIX System and Standards Instructor Notes

Teaching Tips

This slide is intended to provide some insight into what organizations have what responsibilities with respect to standards development within the UNIX system environment. It should be presented from left to right.

The left side of the slide depicts that much of the standards definition is derived from the SVID and BSD interface definitions, but there are other implementations that are also making contributions, such as Carnegie Mellon's Mach for mutiprocessing extensions.

IEEE/POSIX and ANSI are the United States organizations that are attempting to define the standards for the programming interface, the user interface and the operating interface.

International and federal organizations adopt various standards, such as IEEE/POSIX and ANSI to promote the development of international standards in the case of ISO and X/OPEN, or extend the standard for governmental standards as in the case of NIST/FIPS.

The ultimate goals of the organizations are illustrated on the far right side of the slide. Different implementations of the UNIX system that allow:

- Portability of applications.
- Sharing of data and resources (interoperability).
- Progressive hardware solutions, which can be selected based on application requirements.

Other Information

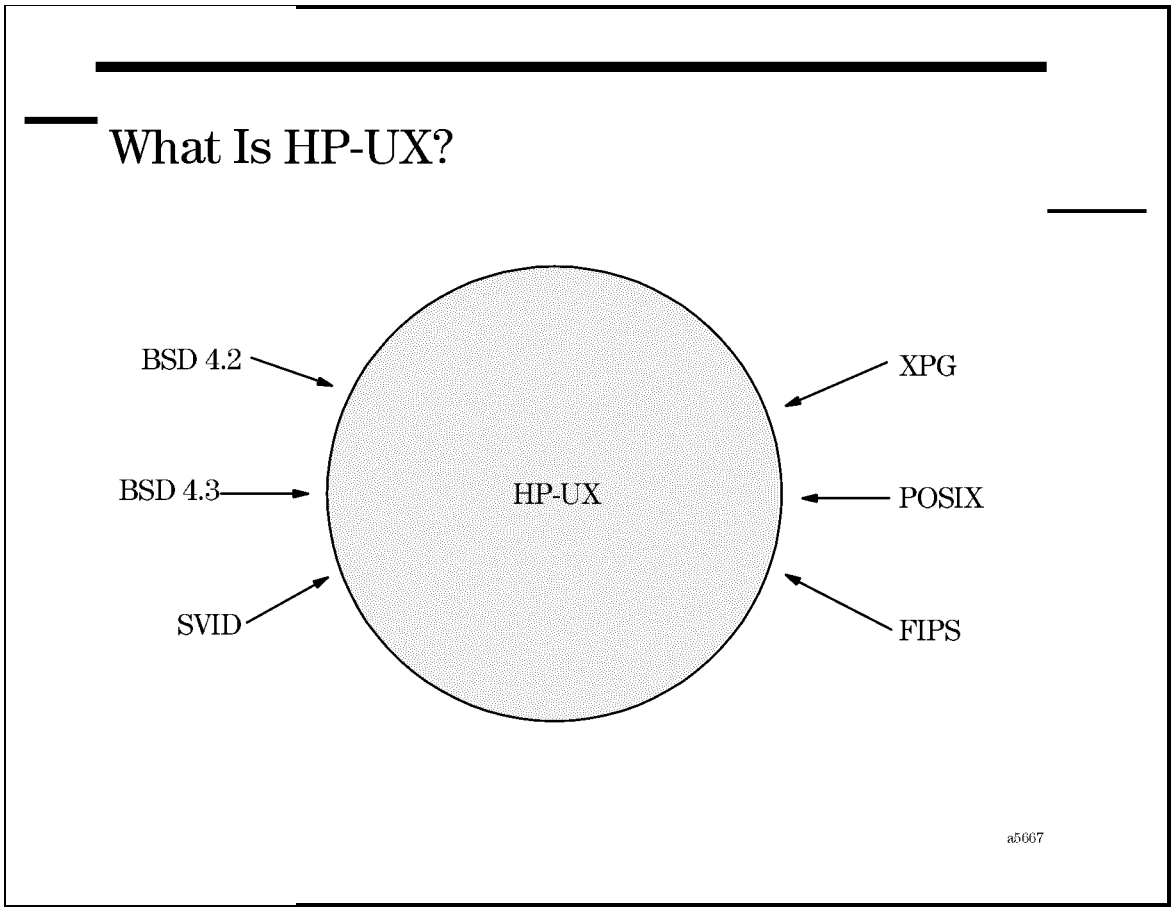
If you are interested in the published POSIX standards you can contact

Secretary, IEEE Standards Board
Institute of Electrical and Electronics Engineers
P.O. Box 1331
Piscataway, NJ 08855-1331
(908) 562-3809

There are also several publications that discuss the standards development that are published by UniForum including: *Your Guide to POSIX*, *POSIX Explored: System Interface*, *POSIX Update: Shell and Utilities*. These publications are informative on the position of the POSIX standards with relation to the System V, BSD, and FIPS requirements.

UniForum
2901 Tasman Drive
Suite 201
Santa Clara, CA 95054-1138
(408) 986-8840

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

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

- X Windows and the Motif graphical user interface
- Common Desktop Environment—a Motif-based user interface (CDENext)
- Visual Editor—a 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

1-7. SLIDE: What Is HP-UX? Instructor Notes

Teaching Tips

Explain the value that Hewlett-Packard has added to the operating system, but be sure to stress that HP-UX conforms to the SVID definition of compatibility, as well as the other emerging standards.

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.

Overview of Module 2

Audience

general user General System users

Product Family Type

open sys Open Systems environment

Abstract

This module is designed to acquaint the student with the process of logging in and out, and to introduce the shell and some basic commands. Since students will need to know how to use Section 1 of the *HP-UX Reference Manual* , the format of the manual pages and the **man** command will also be introduced.

Time

The given times are approximate.

Lab 45 minutes

Lecture 45 minutes

Prerequisites

m44m Introduction to HP-UX

Instructor Profile

UX General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000 Logon access to HP-UX system

Software Requirements

UX11 HP-UX release 11.0

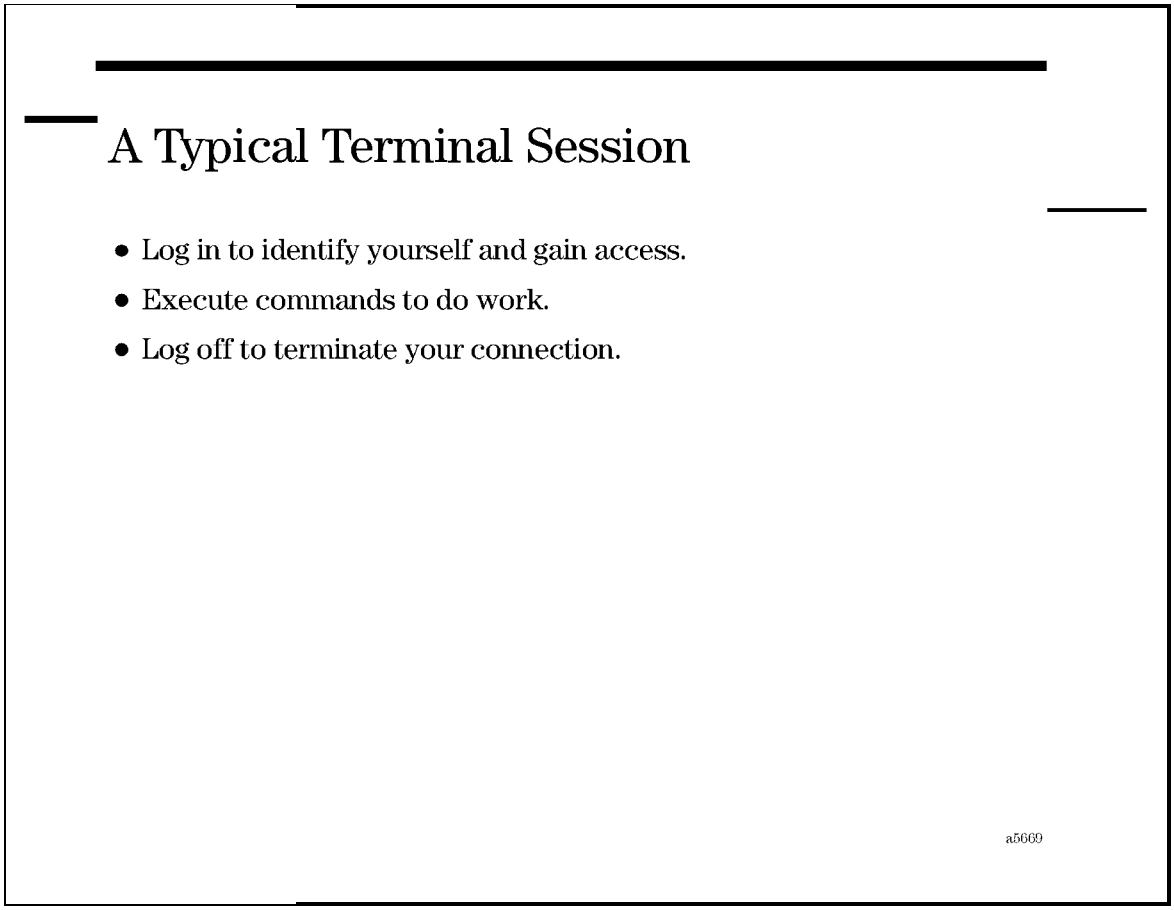
Material List

P/N B2355-90033(T)	*HP-UX Reference Manual* , one per terminal

Lab Instructions

setup1	Create user logon of **user1**, **user2**, ... **user***n*, where *n* is the number of students in the class. Set up one user per student.
printer	A printer should be installed in the classroom, and configured as the default printer.
catman	In order to use the **-k** option of the **man** command **catman -w** must be executed to create the **/usr/share/lib/whatis** database.

2-1. SLIDE: A Typical Terminal Session



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 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.

2-1. SLIDE: A Typical Terminal Session Instructor Notes

Key Points

The basics of a terminal session are: log in, do work, and log out.

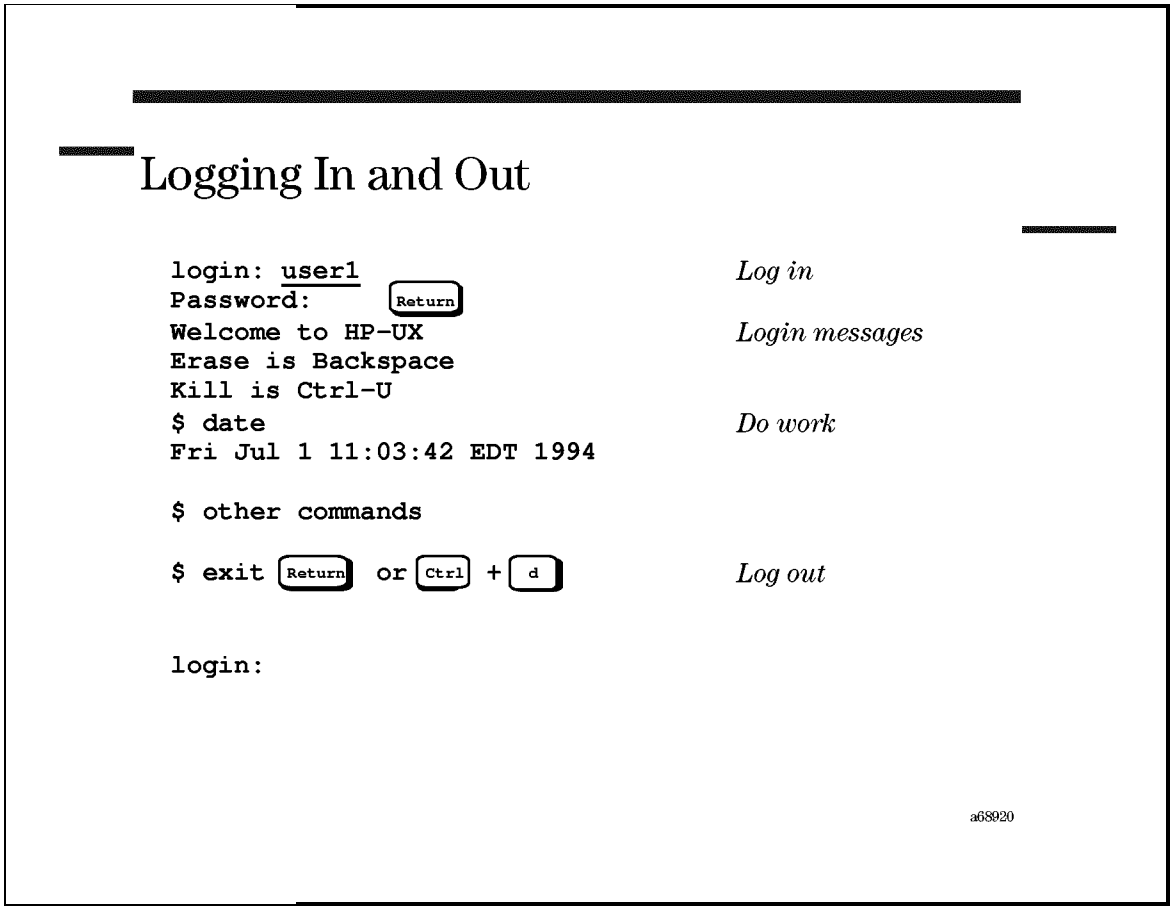
Users cannot access the system's programs or files until they log in.

If a user forgets his or her password, no one can find out what it is because it is encrypted in the file **`/etc/passwd`**. The only solution available is to have the system administrator remove the encrypted password from the **`/etc/passwd`** file and then reenter a new password.

You may want to inform the students of the dangers of leaving their terminals unattended while logged in. You might want to mention the **`lock`** command, which allows users to lock an unattended (alphanumerical) terminal. The lock command prompts the user to enter a special key that must be reentered to unlock the terminal.

In a networked environment, it is recommended that users have the same login ID and password on all systems.

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.

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 logname. 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 Back space key. It is already mapped by the CDE or HP-VUE login process.

2-2. SLIDE: Logging In and Out — Instructor Notes

Teaching Tips

Assign the students their login IDs. You might ask them to log in now and see what problems they encounter. The backspace key will not work. This is covered in the next slide.

If your students must provide a password when they log in for the first time, you should review the password restrictions. The **passwd** command is presented later in this module, which will also describe the password specifications.

When you are logging in, the UNIX system does not know what type of a terminal you are communicating through, so special erase and kill characters are defined for the login process. These allow you to make corrections as you are logging in:

The **#** character deletes the previous character, but it does not erase the character from the display.

The **@** character effectively deletes the current line, and gives you a new line with no new prompt. It will not erase your previous input from the terminal display.

Once you have logged in, the erase and kill characters may change. The user can customize the key sequences for erase and kill using the **stty** command. Common definitions for erase and kill during a UNIX system session are

- erase character — Backspace or CTRL + h, which deletes the previous character
- kill character — CTRL + u, which deletes the current line and starts you on a new line with a new prompt

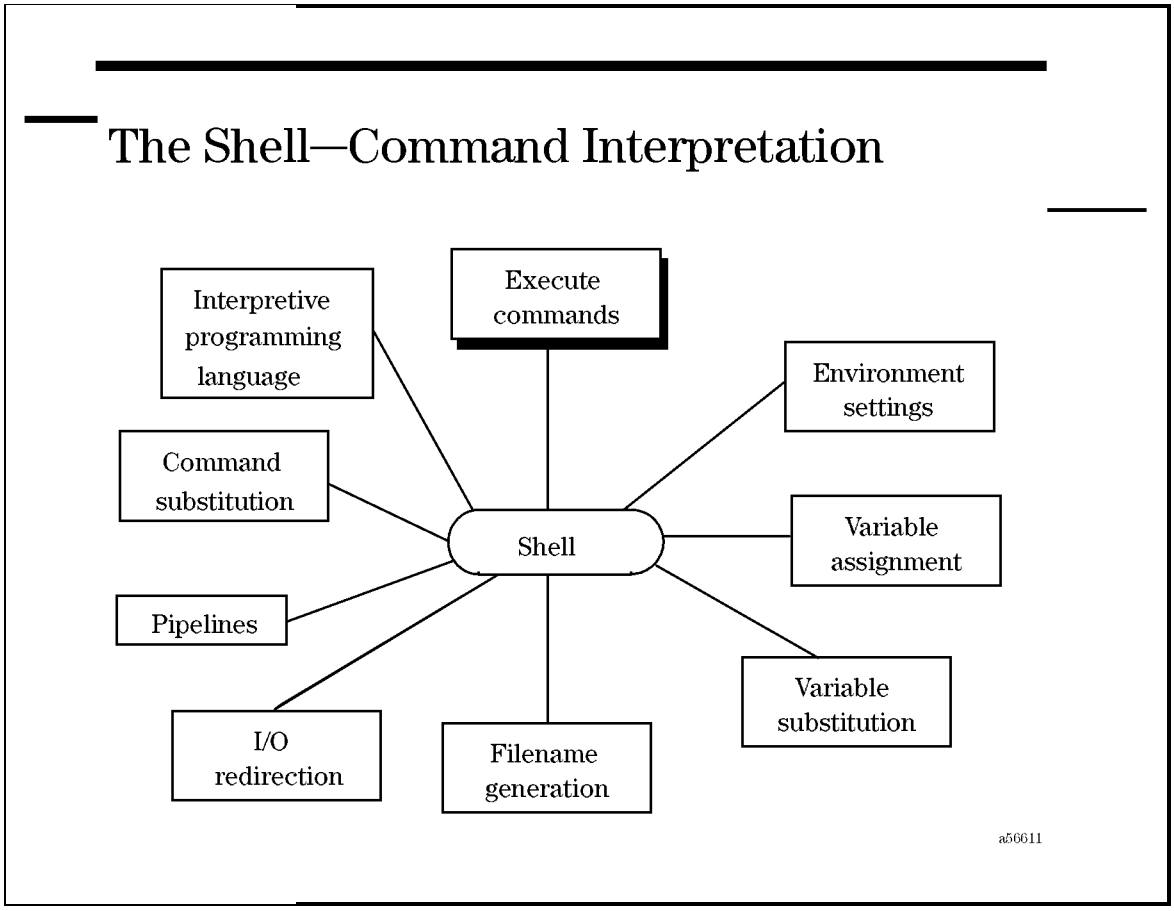
Point out that the students' own computers may have different keys mapped in for erase and kill. Tell them that it is the system administrator's job to map in the correct keys. You may point out the **stty -a** command to see the mapped-in key.

You should log in with lowercase letters. If you log in with the uppercase version of your login identifier, the system will assume you are communicating through a terminal that supports only uppercase characters. As a result, all commands and messages will be displayed in uppercase, but interpreted as lowercase, and the Shift key will be disabled. You may need to show the class the **stty -lcase** command, if anyone tries to log in using capital letters.

NOTE: Instead of remembering the **#** and **@**, you can just press Return when you have mistyped your login name. If your login name is not recognized, the UNIX system will issue the **password:** prompt. Press Return one more time, and the **login:** prompt will be reissued.

NOTE: You can talk about CDE or HP-VUE login right now or tell the students you will talk about the CDE and/or HP-VUE environment only once at the end of the class. In both cases, tell them that they will find some notes concerning CDE or HP-VUE in several chapters.

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.

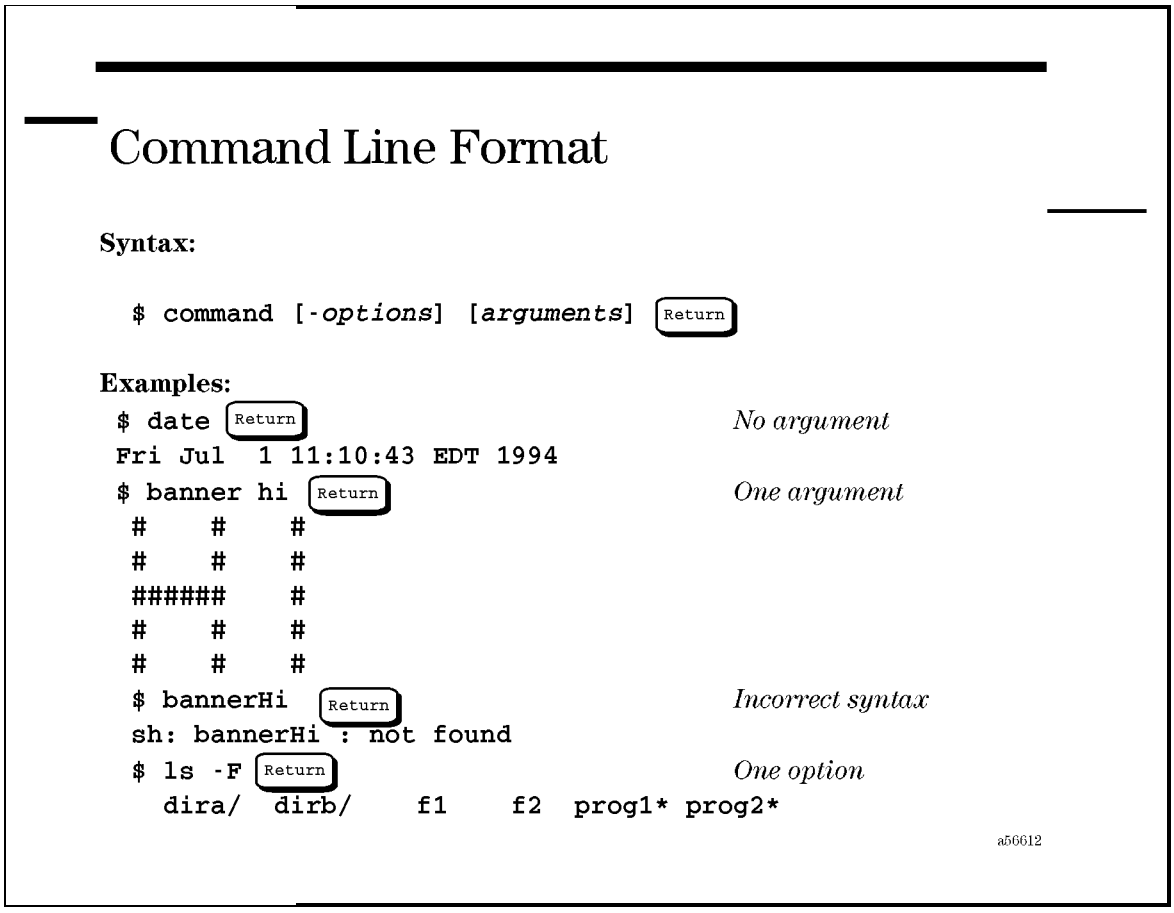
2-3. SLIDE: The Shell — Command Interpretation — Instructor Notes

Teaching Tips

The intention of this diagram is to provide the students with a "big picture" of what the shell has to offer, as well as a simple graphical summary of the shell's capabilities.

If you have decided to speak about CDE or HP-VUE every time there is a difference with alphanumerical stuff, you can tell the students that during CDE or HP-VUE login, a shell is started but they cannot access it directly. They have to open a window called terminal emulator to start another shell process and to gain a **$** prompt.

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.

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`**

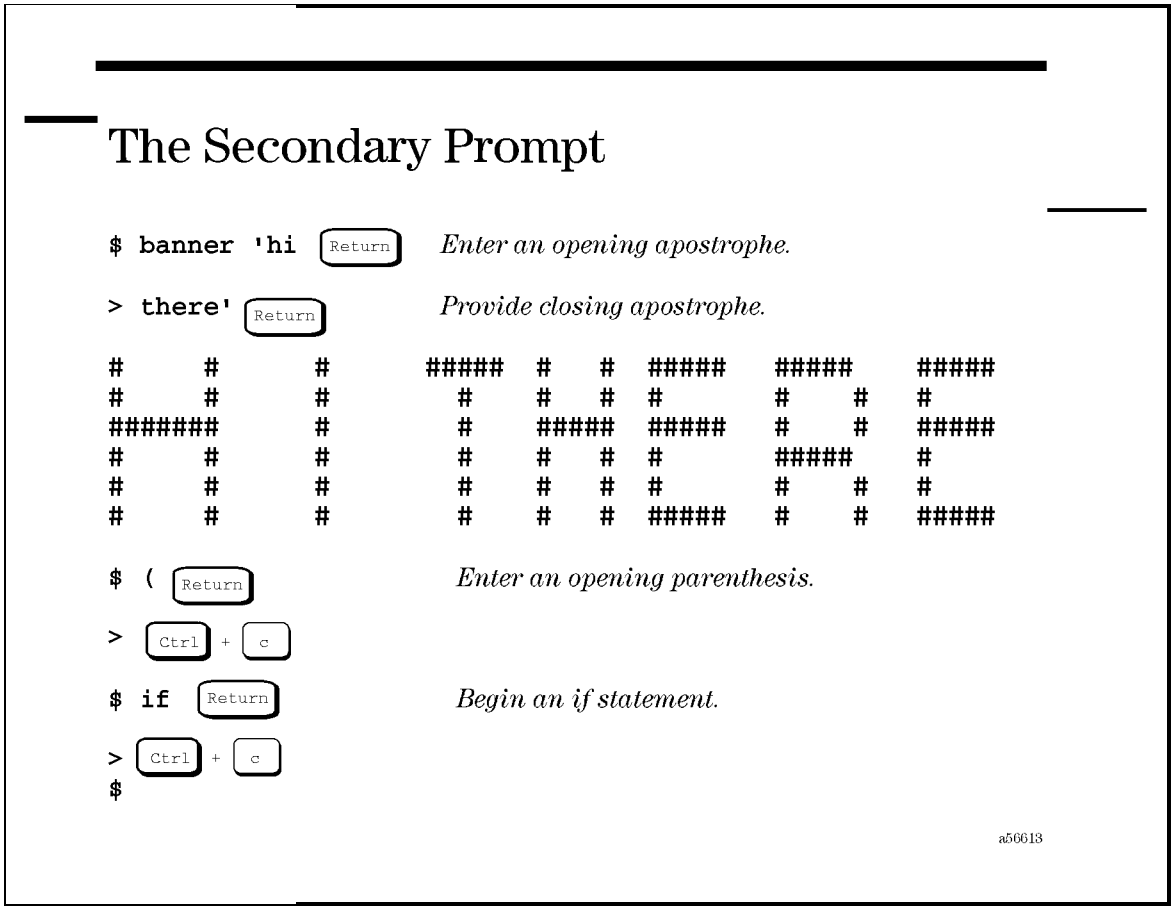
2-4. SLIDE: Command Line Format Instructor Notes

Teaching Tips

Point out the difference between **`banner Hi`** and **`bannerHi`** . You can also mention that commands can be separated by semicolons instead of Return.

Stress that the UNIX system is case sensitive and that most commands are defined as lowercase. This is especially important for users from environments that are not case-sensitive (such as DOS and MPE), or environments that are normally uppercase oriented.

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.

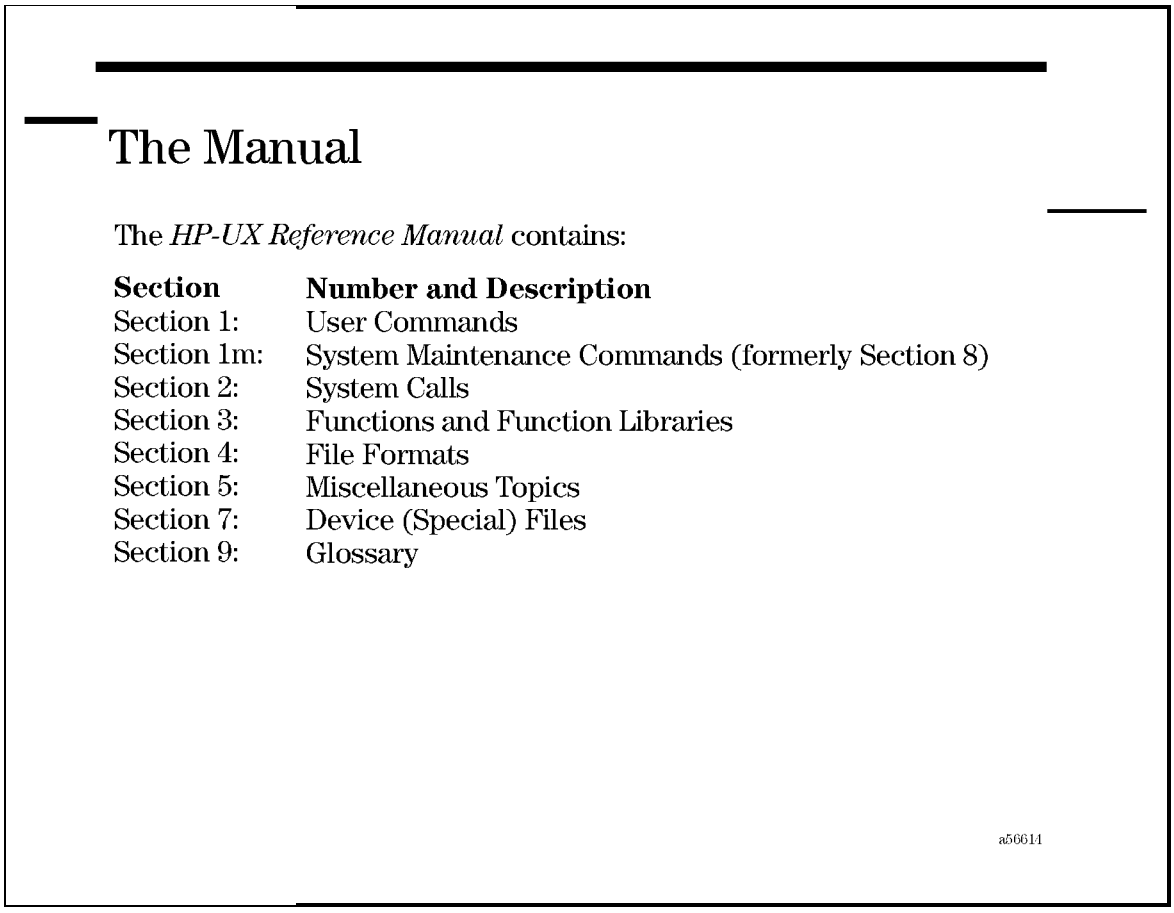
If you enter a command incorrectly, as illustrated on the slide, the shell will issue you a secondary prompt. A special key sequence should be defined to interrupt the currently executing program. Commonly Ctrl + c will terminate the currently running program and return the shell prompt. You can issue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

2-5. SLIDE: The Secondary Prompt Instructor Notes

Key Points

It is common for students to issue a command line that displays the secondary prompt early in the class. Therefore, make sure that students are aware of the technique to interrupt the command line and get back to the prompt. You may want to inform your students that they will be taking advantage of the secondary prompt later.

2-6. SLIDE: The Manual



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.

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.

2-6. SLIDE: The Manual — Instructor Notes

Teaching Tips

Have the students look up the **`date`** command in the paper version of the manual. Have them use the index. Point out the multiple volumes of the *HP-UX Reference Manual* in its current configuration.

The notes here contain the first real reference to the system administrator and the root account. You might want to take a moment here to define the super-user (root) account, and inform the students the responsibilities attached to logging in as *root*.

NOTE: Some of the section classifications have changed between HP-UX 7 and HP-UX 8. Specifically, Section 4: HP-UX 7.0 concerns special files; Section 5: HP-UX 7.0 concerns file formats; and Section 7: HP-UX 7.0 concerns miscellaneous facilities.

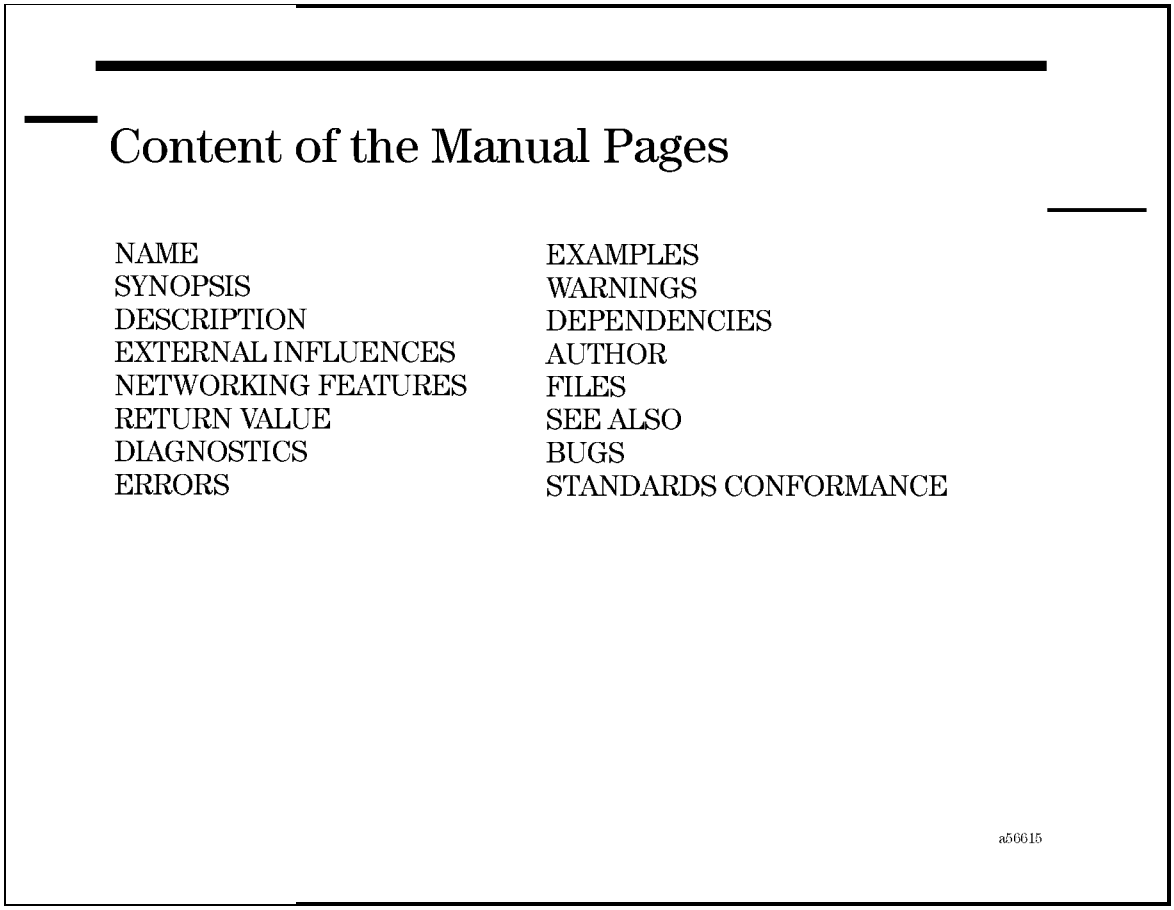
HP-UX Reference Manual Sections

In the *HP-UX Reference Manual* students may notice that sections 6 and 8 are missing. Section 6 listed games on the original UNIX system; no games are currently supported on HP-UX. Section 8 described administrative commands; this material has been moved to section 1M.

NOTE: You may want to describe the LaserROM product or, if it's set up, you can show it to the students. It is a CD-ROM application that contains manuals, software status bulletins, application notes, product catalogs, and system support information about HP computer systems.

NOTE: You may want to introduce the cde or vuehelp facility on HP-UX workstations.

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. 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.

2-7. SLIDE: Content of the Manual Pages Instructor Notes

Teaching Tips

Briefly present each of the sections. The next page will illustrate an example of the man page for the **banner** command.

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

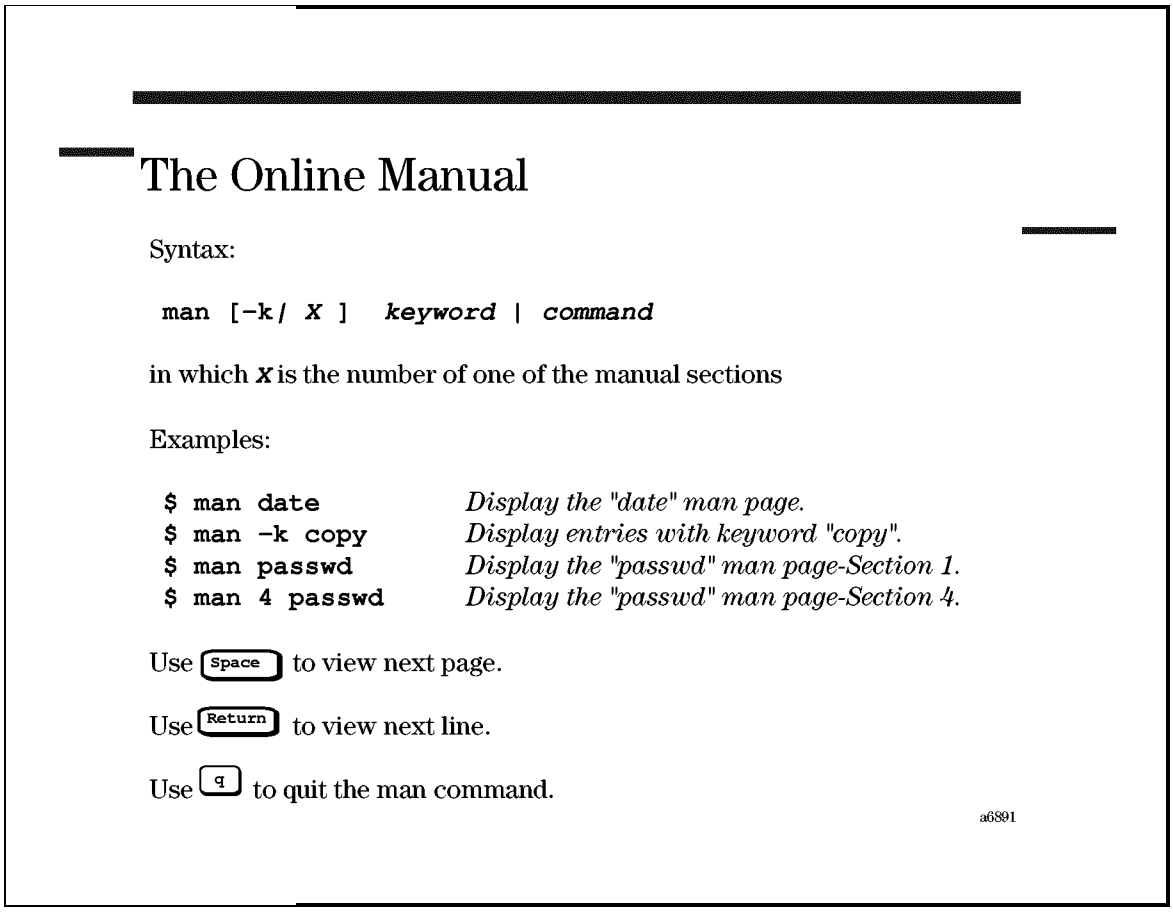
```
banner(1)                                                    banner(1)

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
```

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

Instructor Notes

Point out the different sections of a manual page. Also point out that not all commands will include entries for all manual sections.

2-9. SLIDE: The Online Manual



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.

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 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.

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**.

2-9. SLIDE: The Online Manual Instructor Notes

Teaching Tips

Students should be logged in by now, but if they are not, they can log in at this time to run the following exercise.

Exercise

Objective: To practice with the **man** command and the screen control keys, the students can do the following:

- Log in, if you have not done so yet.
- Display the manual entry for the **ls** command: **man ls**
- Press Space twice to advance forward two screens.
- Press Prev once.
- Quit the **man** command: press q .
- Press Return. Note where the prompt is.
- Press Clear Display to clear the screen.

Students often want to know how to display the previous screen and how to scroll up and down. The Screen Control section in the student notes will describe how students can scroll their display when using HP keyboards and terminals.

Point out the fact that the system administrator has several tasks to do if the online manual is to work as described. This is a chance to describe the value of the system administration class. Note that the -k option requires that the system administrator set up the file **/usr/share/lib/whatis** . This can be done with a **catman -w** command.

The following summarizes the manual directory entries:

Table 2-1. Online Manual Entries

	nroff Source	Formatted Entry
Compressed	/usr/share/man/manX.Z/*	/usr/share/man/catX.Z/*
Uncompressed	/usr/share/man/manX/*	/usr/share/man/catX/*

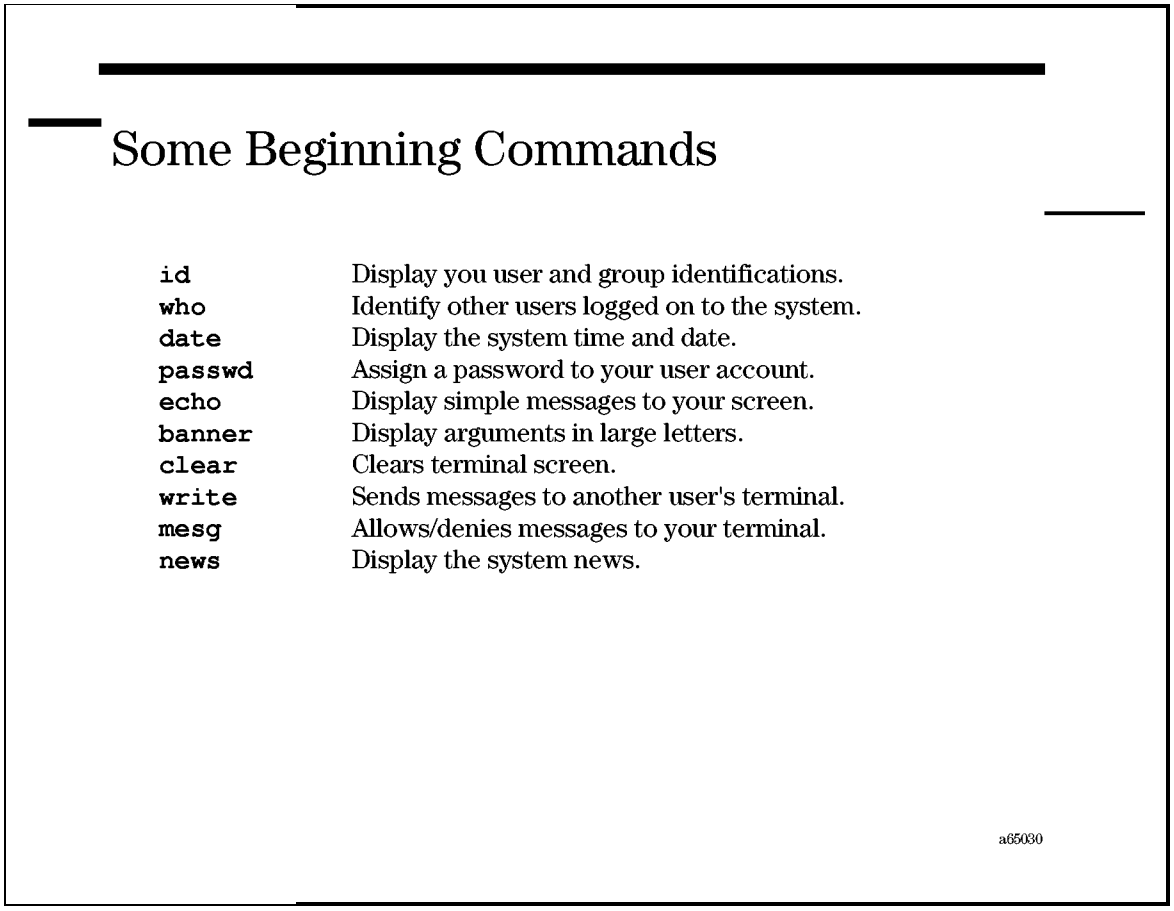
Transition

Now that you know how to log in and how to access the online manual pages, you are ready to learn some UNIX system commands.

Break

If your students are getting restless and need a break, this would be a good place to take a short break before moving into the presentation of the beginning commands.

2-10. SLIDE: Some Beginning Commands



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.

2-10. SLIDE: Some Beginning Commands Instructor Notes

The remainder of the module introduces some simple commands to the students so that they can get comfortable with the keyboard and have some fun sending mail and messages to the other students.

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'**

a6892

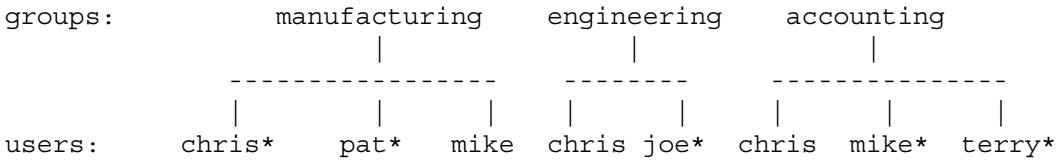
Student Notes

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. 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.

2-11. SLIDE: The `id` Command — Instructor Notes

Key Points

- The computer *must* know your user and group identifiers in order to determine if you have permission to access a file or to run a program.
- Make sure users understand the group concept.

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
```

a56619

Student Notes

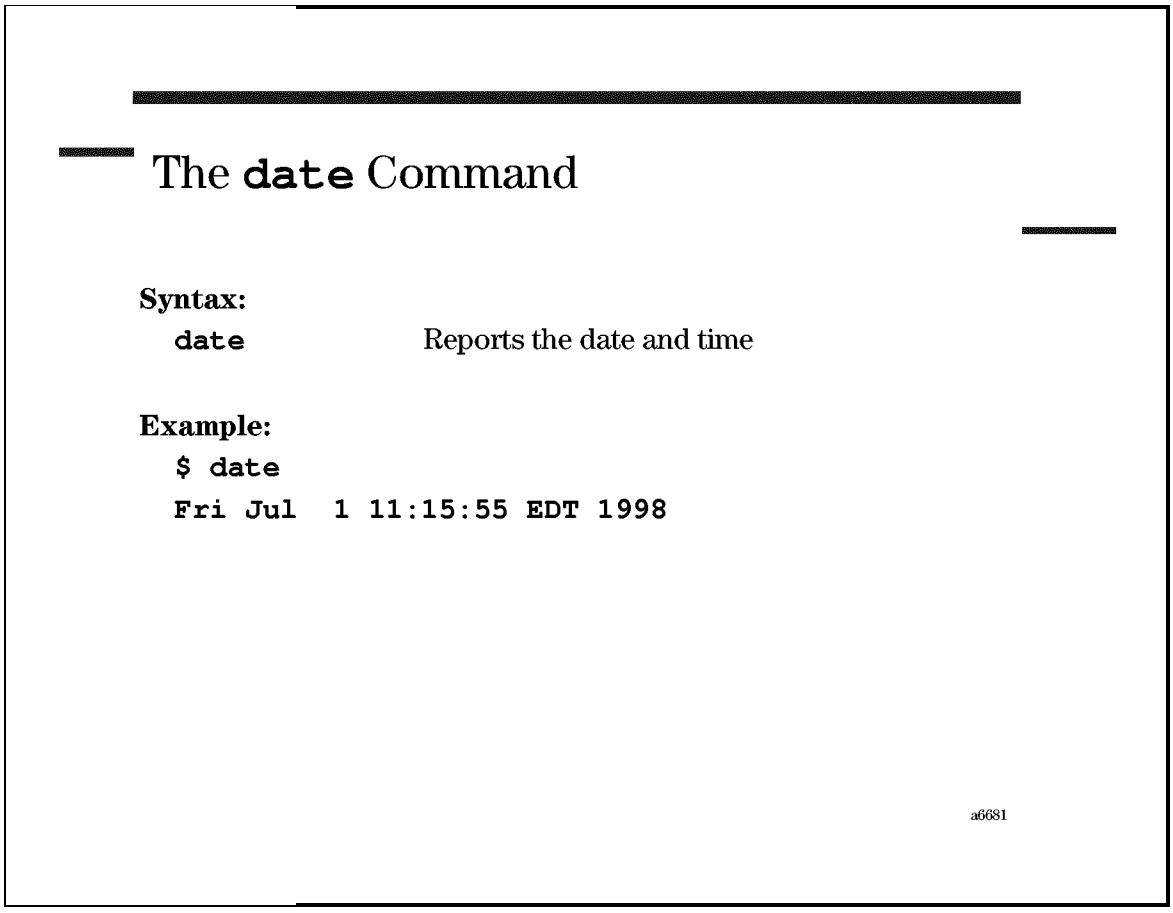
The **`who`** command reports which users are logged into a system, what terminal port each is connected to and login time information. **`who am i`** just reports the user name and port information of the local terminal session. Finally, the **`whoami`** command reports the user name that the system 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.

2-12. SLIDE: The `who` Command — Instructor Notes

Teaching Tips

If you are presenting your class with a network of systems, instead of terminals connected to a single system, the **`who`** command is not very informative. You might want to try the **`rwho`** command instead. This will display the users' names and the system name that they are logged into.

2-13. SLIDE: The `date` Command



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.*

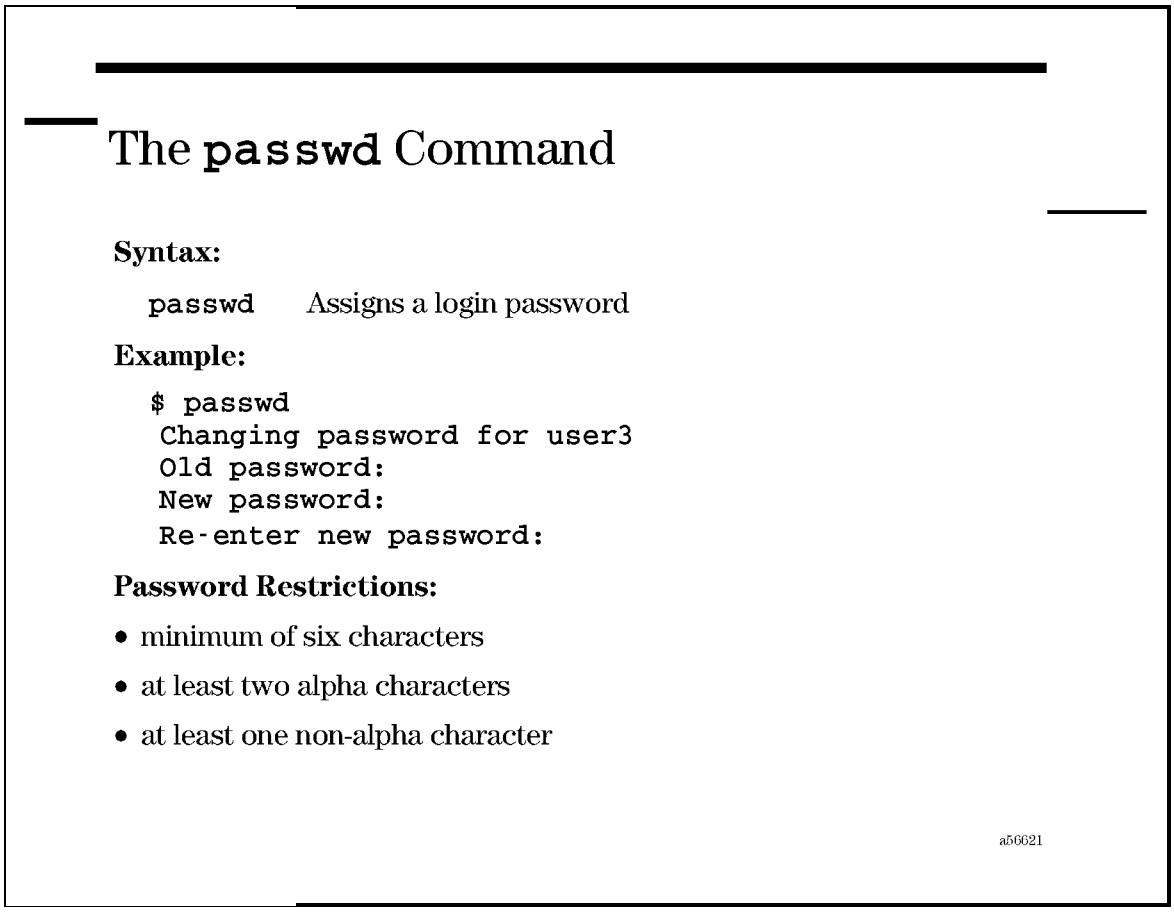
2-13. SLIDE: The `date` Command — Instructor Notes

Teaching Tips

You may point out that the super-user can change the date with this command. You may also want to point out the options available to format the output of **`date`**, such as

```
date +%m/%d/%y
```

2-14. SLIDE: The `passwd` Command



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 which 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.

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 non-alphabetic.

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.

2-14. SLIDE: The `passwd` Command — Instructor Notes

Teaching Tips

Show the students the **passwd** command. Point out that no user of the system can find out another user's password because all passwords are stored in encrypted format. The system administrator (super-user) can reset a password without knowing what the user's password is. Since the password is stored in encrypted format in the **/etc/passwd** file, even the system administrator cannot determine the actual password.

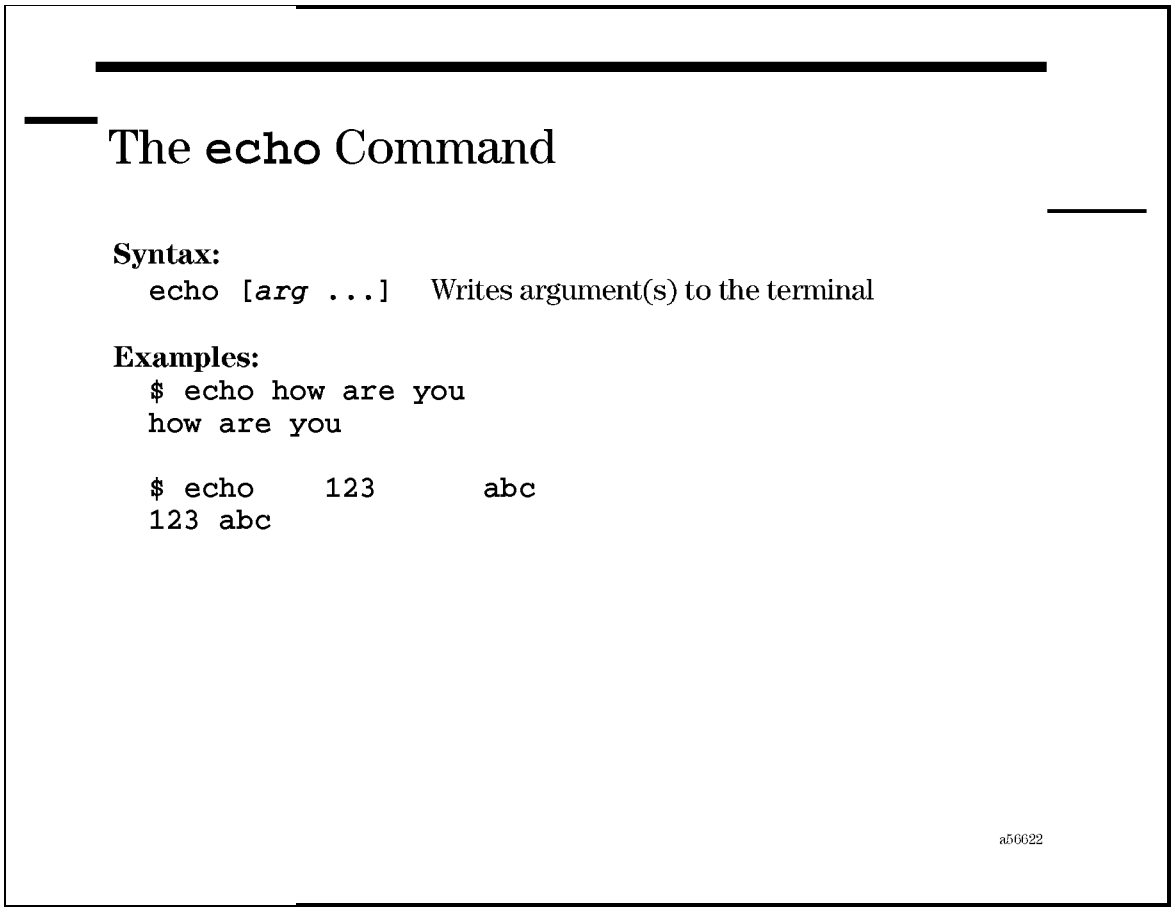
NOTE: For users whose login name is greater than eight characters, the password can only be changed with:

passwd *username*

Otherwise **passwd** receives a truncated version of the user's login name from **/etc/utmp**. See **logname(1)**, **getlogin(3c)**, and **utmp(4)** for additional information.

NOTE: For HP-UX 11.0, **passwd** will not change passwords for user login names longer than eight characters.

2-15. SLIDE: The echo Command



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.

2-15. SLIDE: The `echo` Command — Instructor Notes

Teaching Tips

Point out that right now the **`echo`** command may not seem very useful, but it will be used extensively during shell programming to display messages to the user and display the values of shell variables. These are covered in more detail in other modules.

Key Points

- The **`echo`** command displays the arguments to the screen, separated by a blank space. The shell uses one or more blanks to delimit arguments. Arguments separated by more than one blank space will be echoed with only one space as a delimiter. In the quoting module, students see how spaces can be displayed.

- The **`echo`** command does not distinguish between text and numbers. Everything initially is interpreted by the shell as text. If the context of the command requires a numerical computation, the shell will convert the text to a numerical representation to complete the command.

2-16. SLIDE: The `banner` Command

The **banner** Command

Syntax:

banner ***arg*** **[*arg* ...]** Displays arguments in large letters

Example:

```
$ banner hello

#    #   ######    #        #        ######
#    #   #         #        #        #    #
######   ######    #        #        #    #
#    #   #         #        #        #    #
#    #   #         #        #        #    #
#    #   ######    ######   ######   ######
```

a6893

Student Notes

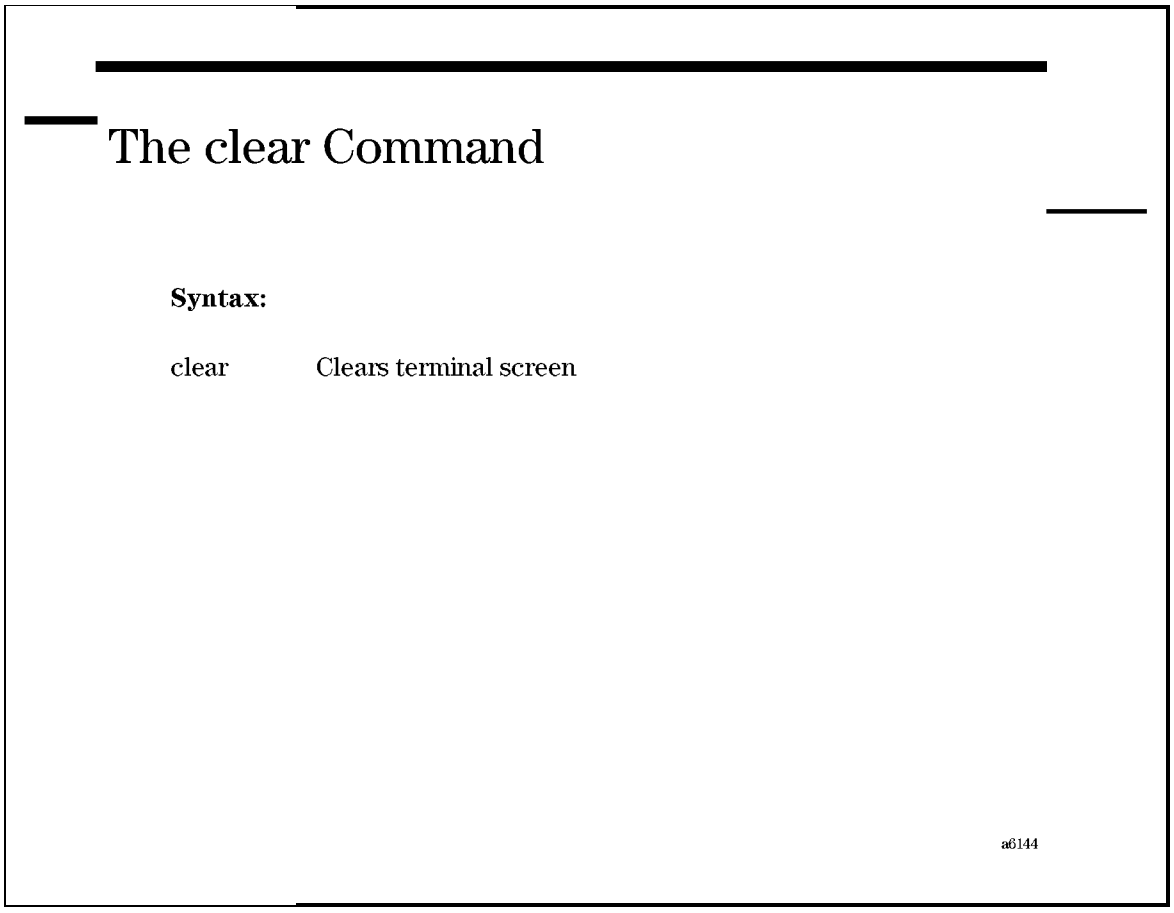
The **banner** command was originally developed, and is still used, for labeling the output from line printers. The banner command displays the command line arguments in large capital letters, one argument per line.

2-16. SLIDE: The `banner` Command — Instructor Notes

Teaching Tips

Point out that the main purpose of the banner command is to label output from the line printer.

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.

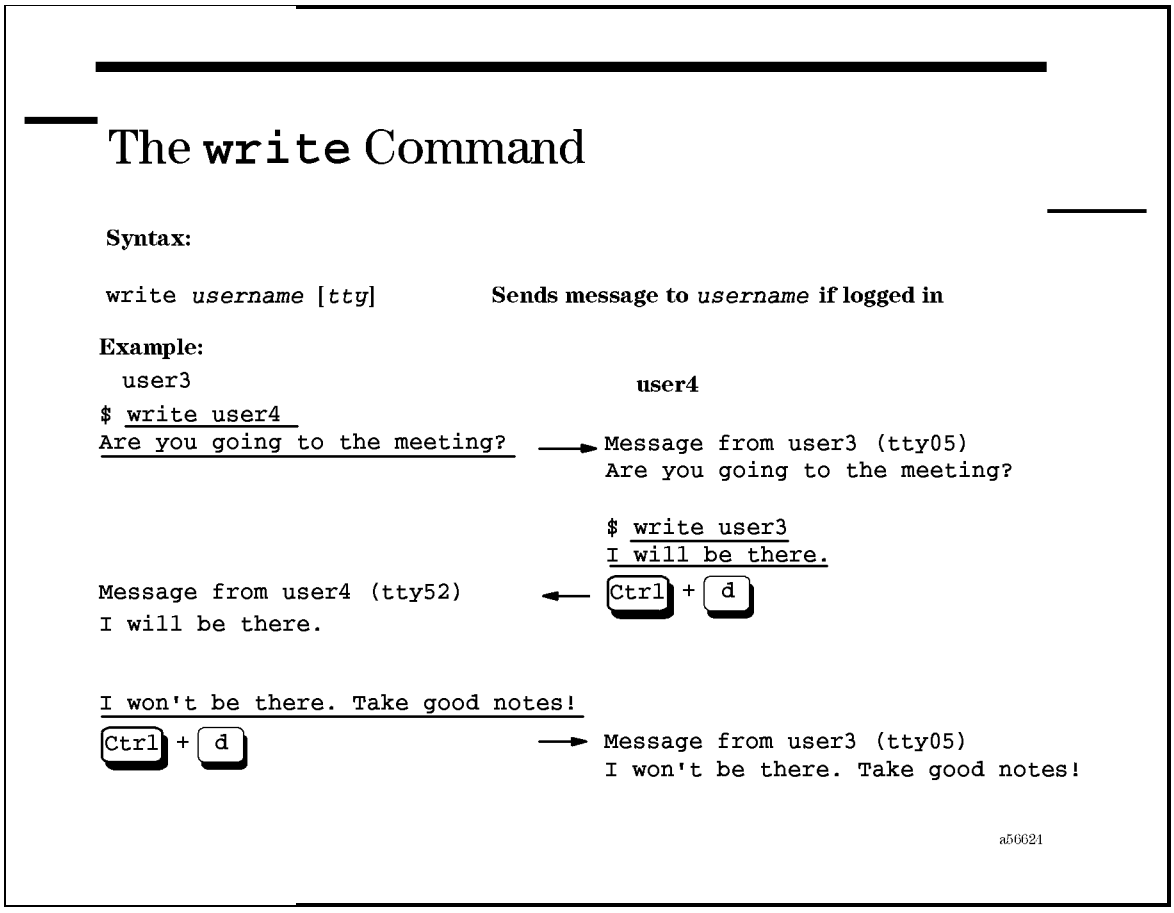
2-17. SLIDE: The `clear` Command — Instructor Notes

Purpose

To introduce a way to clear the screen.

The **`clear`** command reads the **`TERM`** environment variable for the terminal type, then reads the appropriate terminfo database to determine how to clear the screen.

2-18. SLIDE: The `write` Command



Student Notes

The **write** command can be used to send a message to another user's terminal who 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.

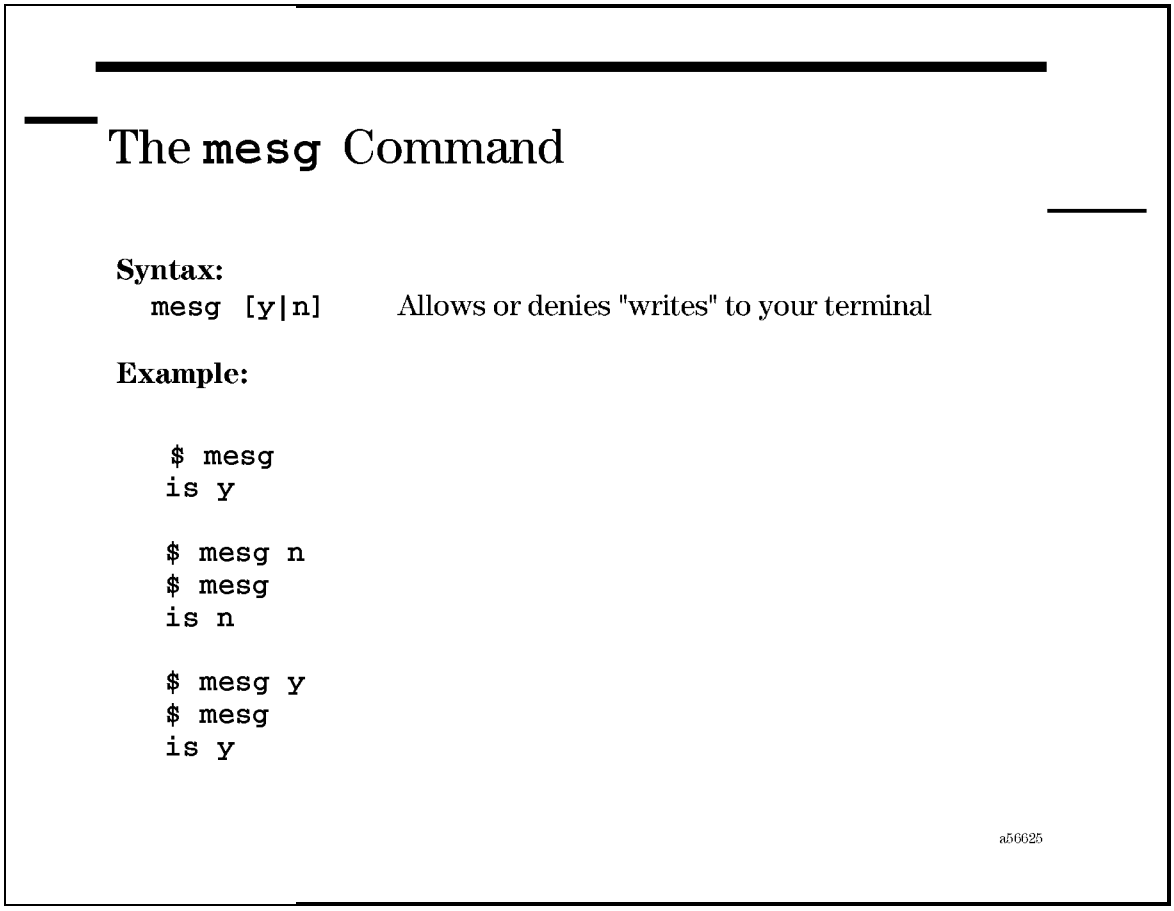
If the person to whom you wish to write is not logged on, you will get the message: *user* **is not logged on**, in which *user* is the user name of the person you tried to reach.

2-18. SLIDE: The `write` Command — Instructor Notes

Teaching Tips

Point out the uses of **`write`**. Note that the super-user can write to anyone at any time. You may also want to mention that users can use write interactively by waiting for a response to the message they sent before ending with CTRL + d.

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 their 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**).

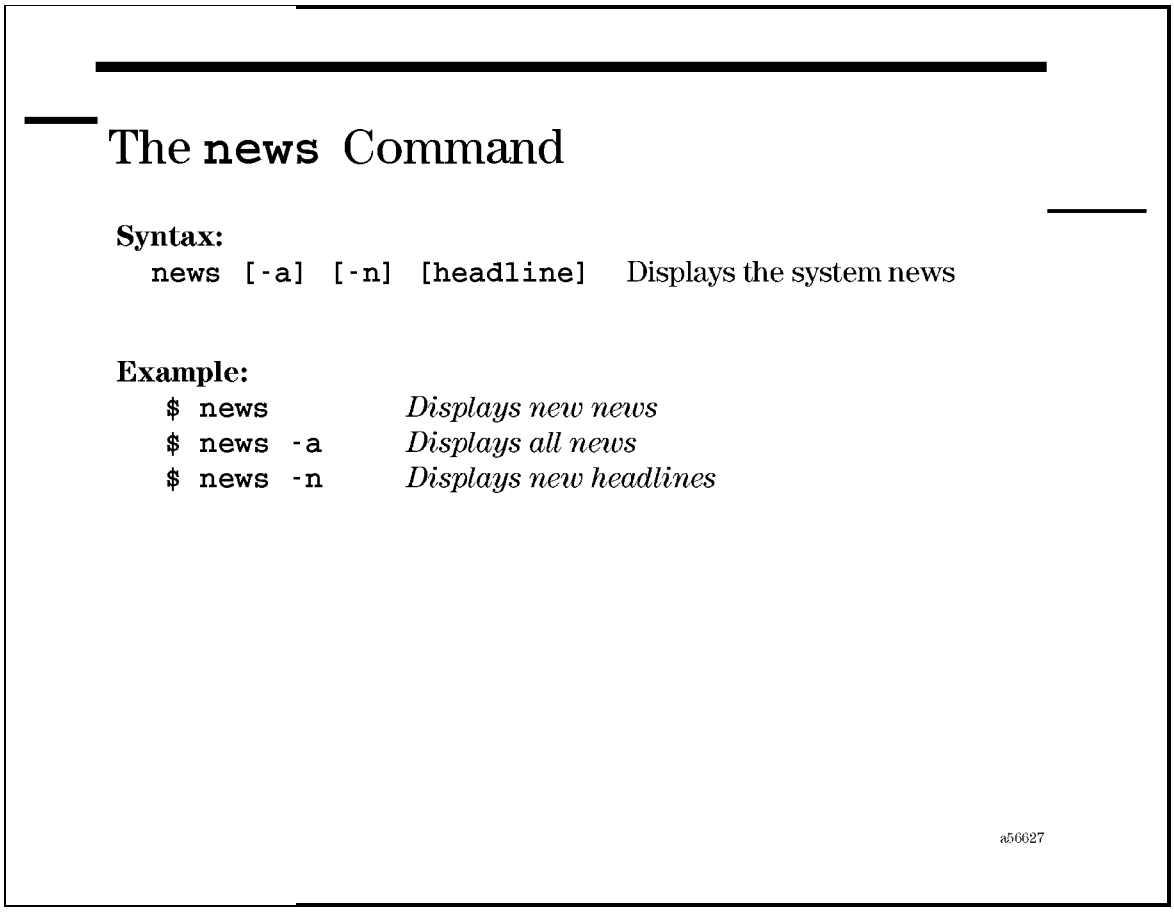
Even when you disable messaging, the system administrator can still send messages to your terminal.

2-19. SLIDE: The `mesg` Command — Instructor Notes

Teaching Tips

Remind the class that denying "writes" does not affect the super-user. We will understand what the **`mesg`** command is doing (for example, taking away write permission from the user's terminal for group and other) if we know file permissions and access.

2-20. SLIDE: The `news` Command



Student Notes

Messages that are of interest to all users on the system can be broadcast through the **news** facility. This is commonly used by the system administrator 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 that 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, **news** knows that it has not been read because the time stamp on your **.news_time** file is earlier than the new news message.

2-20. SLIDE: The `news` Command — Instructor Notes

Key Points

- **`news`** messages are displayed only once, by default.
- You can use **`news -a`** to see all news messages.
- **`news`** maintains the messages that have or have not been read by comparing the time stamp on the news file with each user's **`.news_time`** file.

Teaching Tips

You might want to mention that each news message is stored as a file under the directory **`/var/news`**.

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
$ 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.

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?

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)`**.

10. Using the *UNIX Reference Manual*, find the **`cp`** command. What is its function? What is the minimum number of arguments that it requires?

11. Using the *HP-UX Reference Manual*, find the **ls** command. What is its function? What is the minimum number of arguments that it requires?

12. Issue the command **ll /usr/bin**. You will see several screens worth of data scroll by. 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 **ll** command?

13. Log out of your terminal session. Log back in with the CAPS lock on. How can this situation be corrected without logging off and then back in again. (Hint: Look at the manual page for the **stty** command.)

2-21. LAB: General Orientation — Instructor Notes

Time: 45 minutes

Purpose

To practice with the shell command interpreter by entering simple commands. You will enter the messaging and user communication commands.

Notes to the Instructor

Assigpartners for each student so that they can use the **write** command.

Introductory Exercises 1–4	Logging in, logging out, and entering simple **echo** commands
Intermediate Exercises 5–9	Communications, **who**, and **write**
Advanced Exercises 10–13	Using manual pages to find other options

All students should at least get through the intermediate exercises.

Solutions

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

Answer:

You may have had a problem if you made a mistake while typing in your user name or password and tried correcting it with the Backspace key. Remember, the # key is used to erase while logging in.

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

Answer:

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
$ banner
```

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

Answer:

```
$ echo                              correct
$ echo hello                        correct
$ echohello                         incorrect
$ echo HELLO WORLD                  correct
```

The **echo** command will work with zero or more arguments. As the arguments are just seen as strings of characters, and echoed back to the screen, it does not matter whether they are uppercase or lowercase.

The shell needs white space (spaces or tabs) to separate commands from arguments. The third command line doesn't work because the shell is trying to execute a command called **echohello** instead of executing the **echo** command and passing the argument *hello* to it.

```
$ banner                            incorrect
$ banner HELLO                      correct
$ BANNER hello                      incorrect
```

The **banner** command requires at least one argument, unlike the **echo** command. Therefore, the second entry is legal, because **banner** does not care if the string(s) to be echoed are uppercase or lowercase. In the third instance the shell will look for a command called **BANNER**, which is not a legal shell command. Remember, the shell is case sensitive, and therefore **banner banner** is not the same as **BANNER**.

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

Answer:

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

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?

Answer:

```
$ who
$ who am i
$ whoami
```

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

Answer:

```
$ mesg
```

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?

Answer:

```
$ who                                  Confirm that your partner is logged in.
$ write partner
message contents
message contents
Ctrl + d                               Conclude conversation.
```

If your partner is not logged on, you will get the message:

```
partner is not logged on.
```

If your partner has disabled messaging on his or her terminal, you will get the message :

```
Permission denied.
```

8. Read the system's news. What command did you use? Can you display the news *after* you have read a message?

Answer:

```
$ news
this is a news message
$ news
no new news
$ news -a
this is a news message
```

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)**.

Answer:

```
$ date +%m-%d-%y
```

10. Using the *UNIX Reference Manual*, find the **cp** command. What is its function? What is the minimum number of arguments that it requires?

Answer:

The **cp** command is used to copy one or more files. It requires at least two arguments: a source file name and a destination file name.

11. Using the *HP-UX Reference Manual*, find the **ls** command. What is its function? What is the minimum number of arguments that it requires?

Answer:

The **ls** command is used to display file names. It requires no arguments. Notice it has many options available. Each option will extend the capability of the **ls** command, and each option is identified as a single letter.

12. Issue the command **ll /usr/bin**. You will see several screens worth of data scroll by. 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 **ll** command?

Answer:

ll /usr/bin should generate several screens worth of output. Issuing the **clear** command after moving the cursor to the top of the current screen will clear only the last screen of output. Scrolling down will display the previous screens.

13. Log out of your terminal session. Log back in with the CAPS lock on. How can this situation be corrected without logging off and then back in again. (Hint: Look at the manual page for the **stty** command.)

Answer:

Notice that if you hit the Caps Lock key, it has no effect. You must use the **stty** command to disable the Caps lock:

```
$ STTY -LCASE
```

then hit the Caps Lock key on your keyboard. You will now be able to enter uppercase and lowercase letters. This interface is provided for terminals that support only uppercase input, so that they can interpret the commands properly that are normally defined as all lowercase.

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.

Overview of Module 3

Audience

general user	General system user

Product Family Type

gui	Graphical user interface

Time

Lab	45 minutes
Lecture	60 minutes

Language

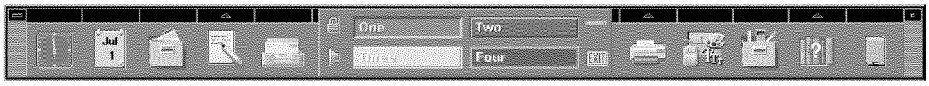
usenglish	U.S. English

Platform

hpux	HP-UX computer systems

3-1. SLIDE: Front Panel Elements

Front Panel Elements



a57642

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.
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 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 used as a drop zone.

CDE Reference Manuals

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

3-1. SLIDE: Front Panel Elements — Instructor Notes

Purpose

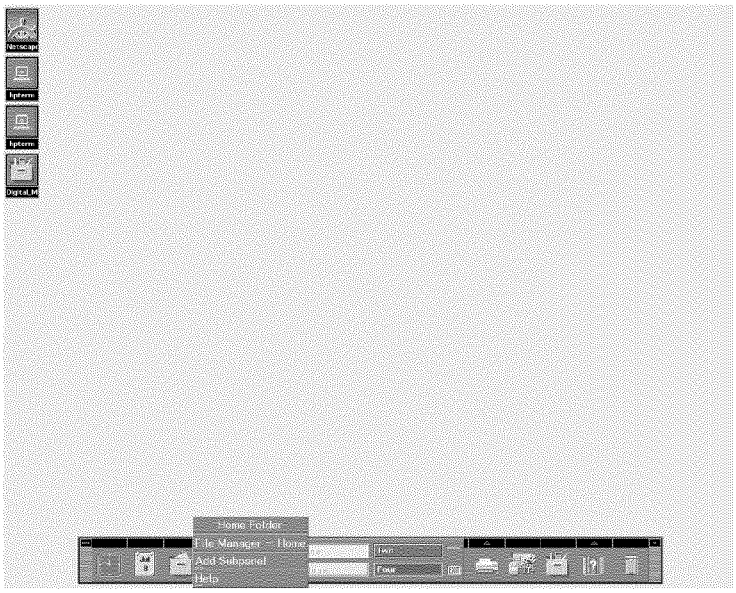
To present a brief overview of CDE features and benefits. Most will be covered in more detail, so there isn't need to get into too much detail here.

Teaching Tip

You may want to be sure students understand how to access subpanel menus before continuing on. The only controls that have subpanel menus are the ones with up arrows above the control. Once the subpanel is displayed, the arrow turns to a down arrow. Click on the down arrow to close the subpanel.

3-2. SLIDE: Front Panel Pop-Up Menus

Front Panel Pop-up Menus



a57643

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.

3-2. SLIDE: Front Panel Pop-Up Menus Instructor Notes

Purpose

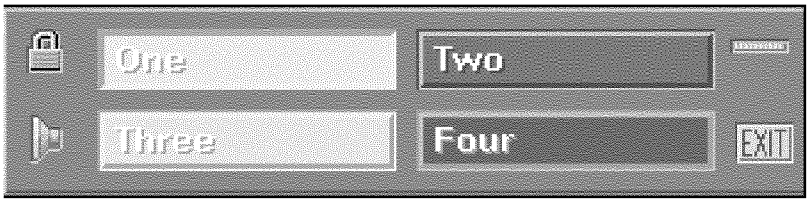
To give an idea of the functionality of pop-up menus.

Key Points

- Pop-up menus are available for the entire desktop, including each application and control in the Main Panel, subpanels, and within the applications.
- This begins to touch upon customization (for example putting the Terminal on the Main Panel). You can touch upon the subject lightly, but this module is not intended to cover customization in detail.

3-3. SLIDE: Workspace Switch

Workspace Switch



a57644

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.

3-3. SLIDE: Workspace Switch — Instructor Notes

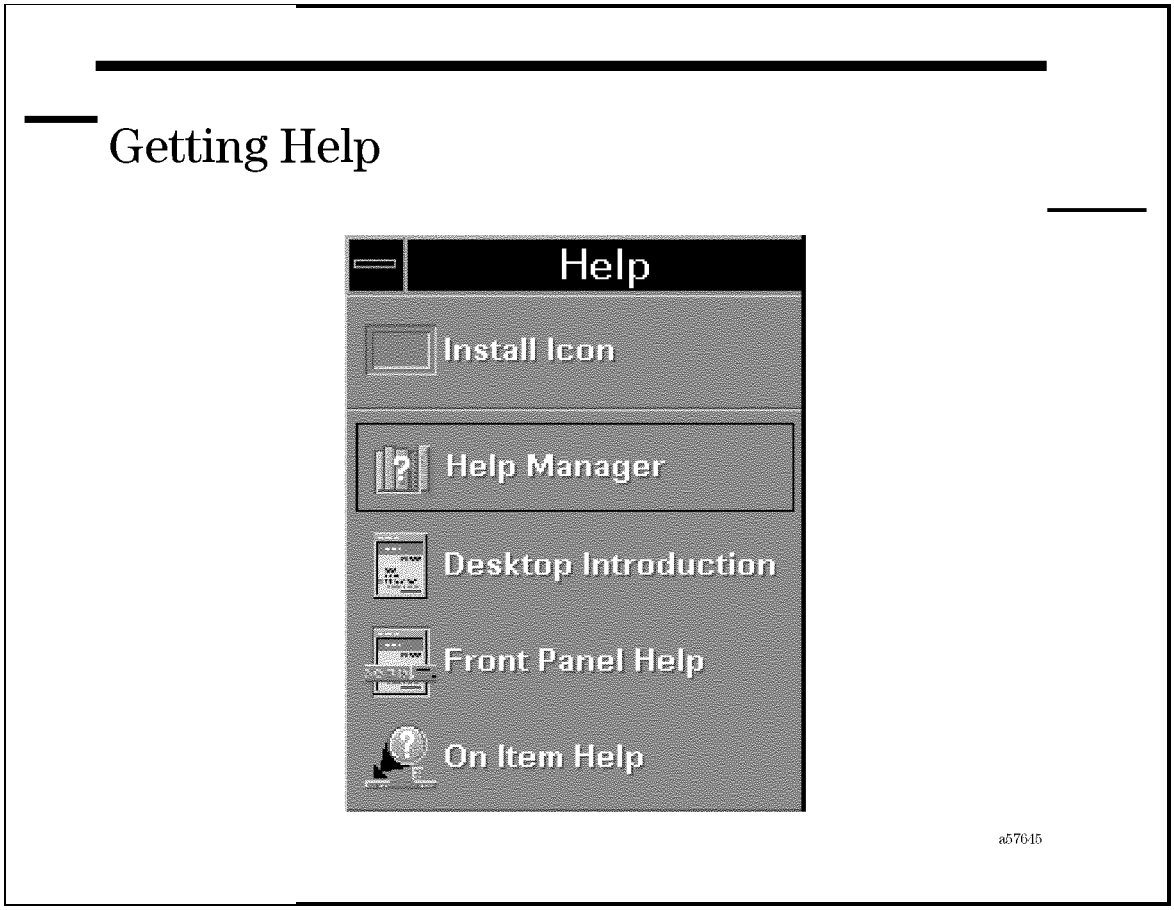
Purpose

To understand the purpose of the workspace switch, and how to make the switch from one workspace to another.

Teaching Tip

Don't get too bogged down in the details of workspace manager. There are some configuration files that manage the workspace manager including **`$HOME/.dt/dtwmrc`**. Students may ask if they can increase the number of workspaces or name them. The easiest way to do so is through the pop-up menus, although they can be done with the configuration files as well.

3-4. SLIDE: Getting Help



Student Notes

Online help is available for each standard application within CDE. The menubar 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.

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.

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** on the menu bar, and then choosing **Print**.

3-4. SLIDE: Getting Help — Instructor Notes

Purpose

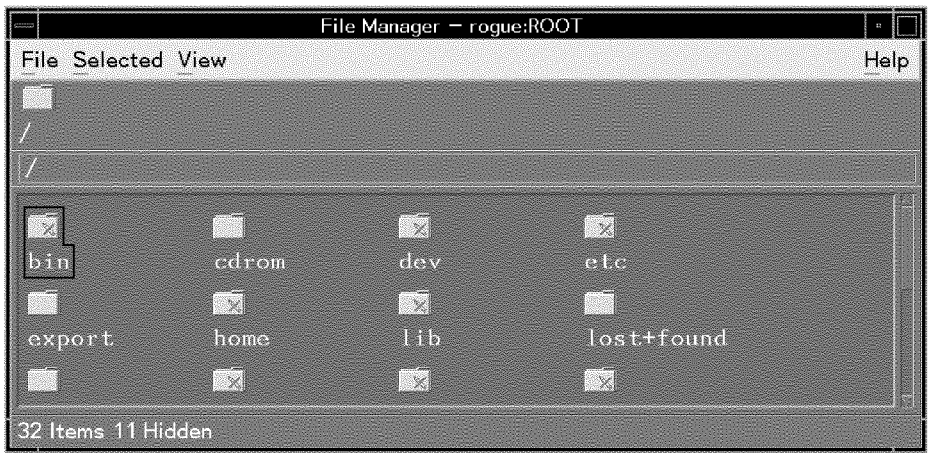
To give an overview of how to get help on the desktop applications.

Teaching Tip

Give students a chance to play around with the help facility, perhaps trying what is discussed in the student notes. You may get questions on how to integrate their own application's help within Help Manager. That is beyond the scope of this slide. If this comes up refer them to the *CDE 1.0 Advanced User's and System Administrator's Guide*,P/N B1171-90102.

3-5. SLIDE: File Manager

File Manager



a57646

Student Notes

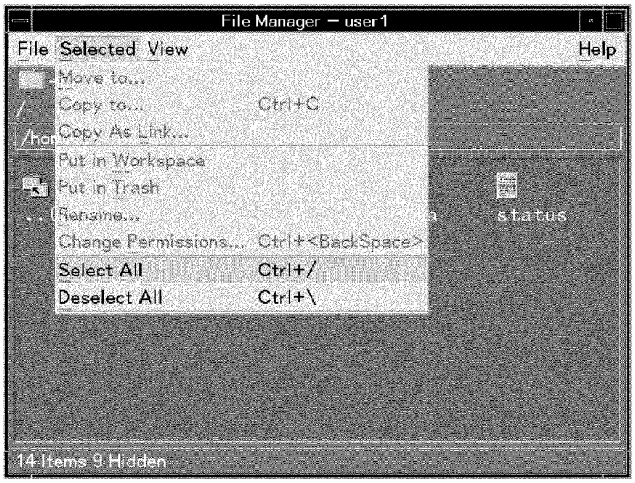
The File Manager allows you to manipulate files (i.e. moving, copying printing, etc) on your system without having to learn complex commands. Files, folders (directories), and applications are displayed as icons. The File Manager is activated by clicking on the File Manager control in the Front Panel.

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 menubar. 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

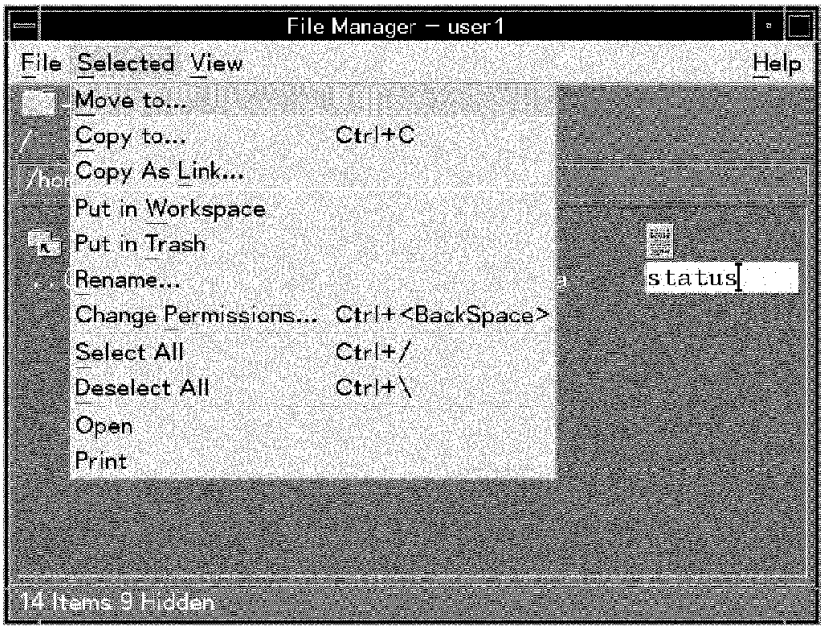
mouse to draw a box around the icons you want to select, then release the mouse button. Once you have selected your object(s), you may then take action by using the File Manager Menu. If you do not select at least one object, some of the menu actions will be unavailable. For example, on the **`File Manager Selected`** menu many of the actions are not highlighted because there is no selected object to take action upon.



a576125

Figure 3-1.

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 3-2.

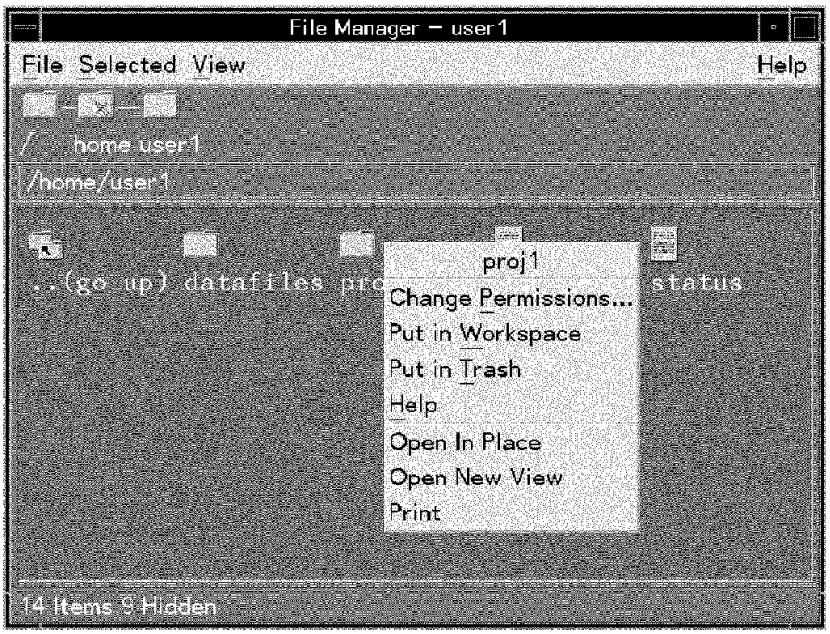
Drag-and-Drop

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, trash can, 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 3-3.

3-5. SLIDE: File Manager — Instructor Notes

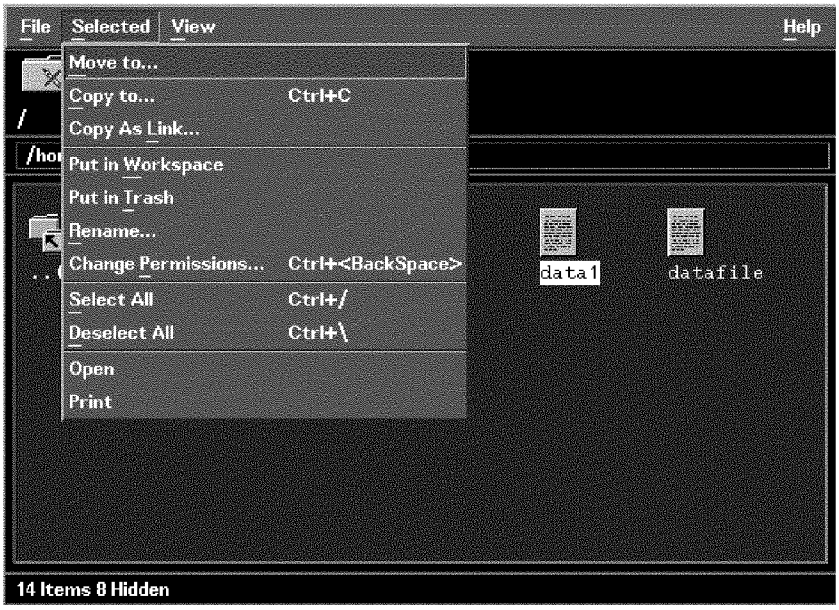
Purpose

To introduce the File Manager Window and Desktop and learn basic file management skills.

Actual File Manager tasks are covered on the next slide.

3-6. SLIDE: File Manager Menu Tasks

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.

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 containing 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.

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.

3-6. SLIDE: File Manager Menu Tasks Instructor Notes

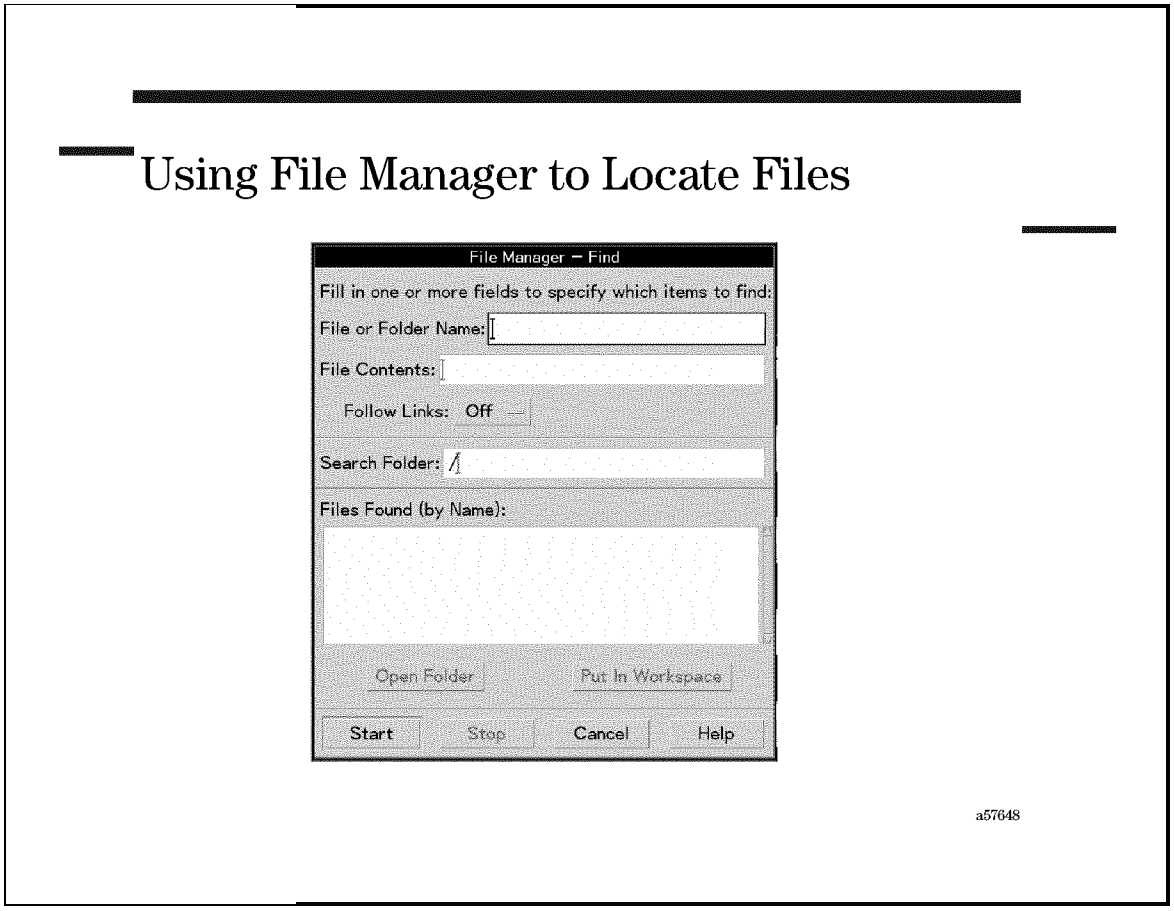
Purpose

To give students the understanding of what types of file related activities can be done using File Manager.

Key Points

- A file or folder cannot have action taken upon it with out first being selected. If users do not select the file, they will not have the option of specific tasks.
- The **`Copy As Link`** selection does a symbolic link, however it acts like a hard link. This is to accommodate the fact that links may be done across filesystems (thus necessitating a symbolic link). This may cause some confusion.
- Once you select to put a file or folder in the workspace, you can remove it from the workspace by positioning the mouse pointer directly on the icon, and pressing mouse button 3. One of the selections of the pop-down menu is to **`Remove from Workspace`**.
- When renaming a file or folder icon, users must press Return or the new name will be lost.

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.

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

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.

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

3-7. SLIDE: Using File Manager to Locate Files — Instructor Notes

Purpose

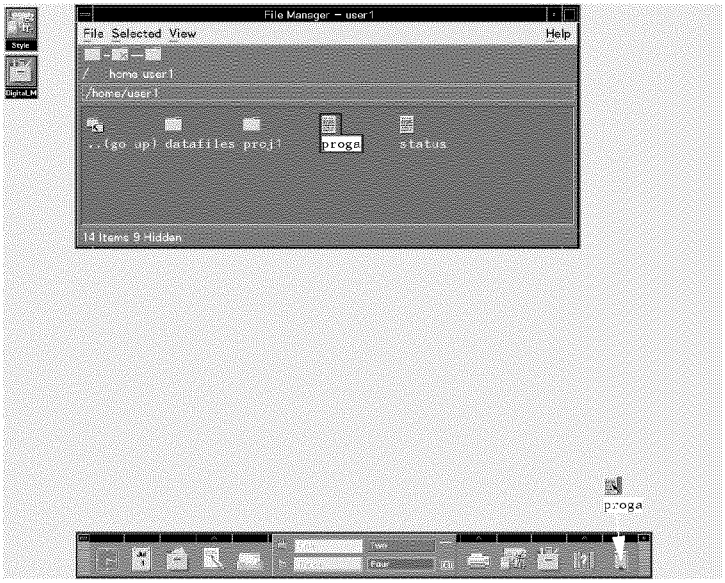
To understand how to find files and folders, without necessarily knowing where in the directory structure they lay, or even what their name is.

Key Points

- When specifying the File or Folder name, be sure *not* to include the parent directory name. *Only* the file or folder you are searching for is specified. Otherwise the search will be unsuccessful. The parent directory information is specified in the **`Search Folder:`** field.
- Specifying both name and contents filter will speed up the search

3-8. SLIDE: Deleting Objects

Deleting Objects



a57649

Student Notes

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

To Place an Object in the Trash Can

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

To Retrieve an Object from the Trash Can

- Double click on the Trash Can icon in the Front Panel to open the Trash Can window. You will see a list of objects which have previously been deleted displayed.
- Select the object you want to restore.
- 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 Trash Can window by clicking on the Trash Can 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!*

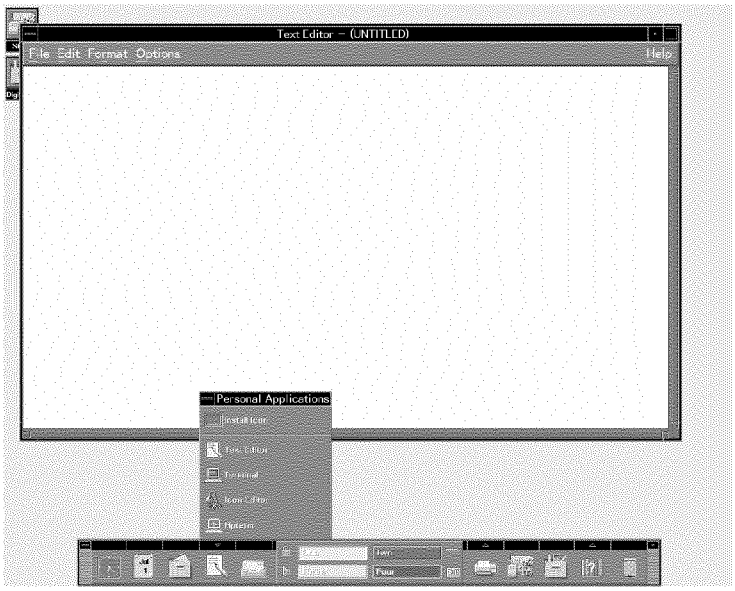
3-8. SLIDE: Deleting Objects — Instructor Notes

Purpose

To understand how to delete files and folders from the system or File Manager

3-9. SLIDE: Using the Text Editor

Using the Text Editor



a57650

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.

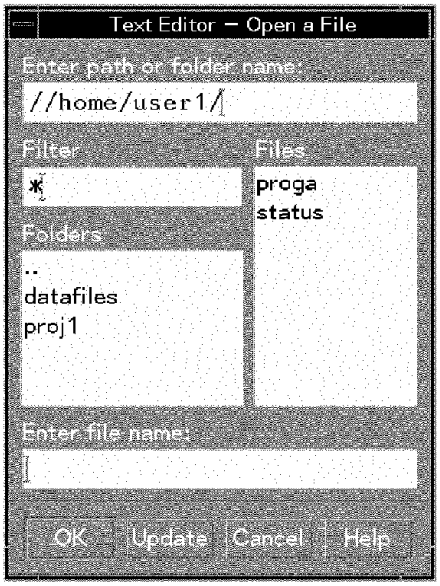


Figure 3-4.

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 which 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.

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

- 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.
- Choose **Delete** from the Edit menu or press the Delete key.

Clear Text

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

- 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.
- Choose **Clear** from the Edit menu.

Find and Changing Text

- Choose **Find/Change...** from the Edit menu
- Type the text you want to find in the **Find** field.
- Type the replacement text in the **Change To** field. If you want to delete the text altogether, this field can be left blank.
- Press Return or click Find to begin the search.
- If a match is found, the cursor will be positioned at the 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.
- Click Close when done.

To Undo an Edit

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

Correct Misspelled Words

- Choose Check Spelling from the Edit menu. The Spell Dialogue Box will be displayed (Fig 4–5).

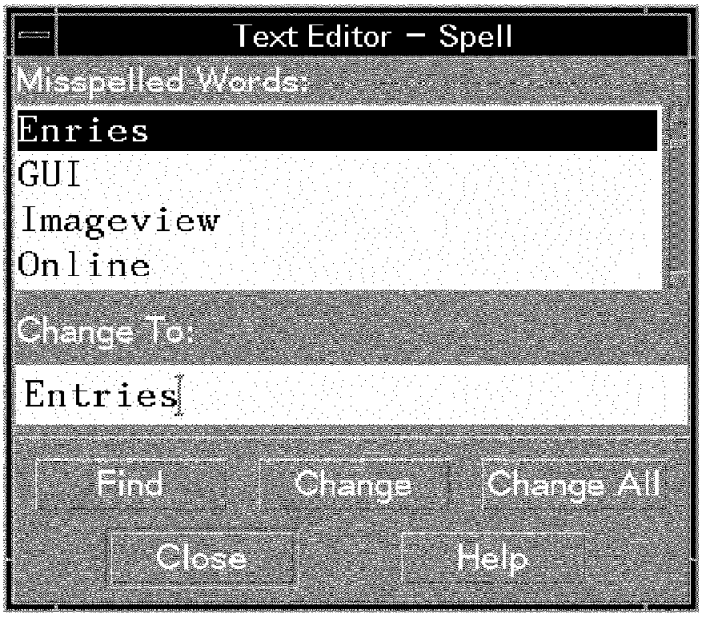


Figure 3-5.

- Type the correct word into the **Change To** field.
- 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.
- Click Close when you are done.

Formatting The Document

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

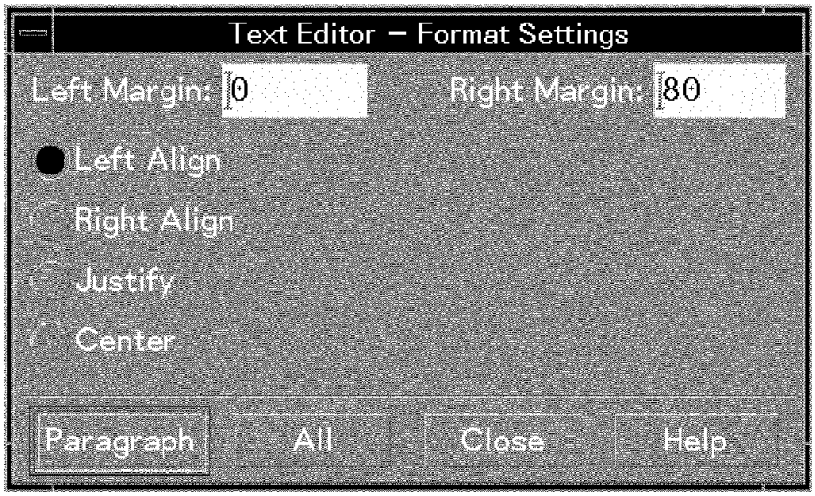


Figure 3-6.

- Enter margins
- Select left, right, justify (block style), or center alignment
- 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`**.
- 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 you want to go to and press Return.

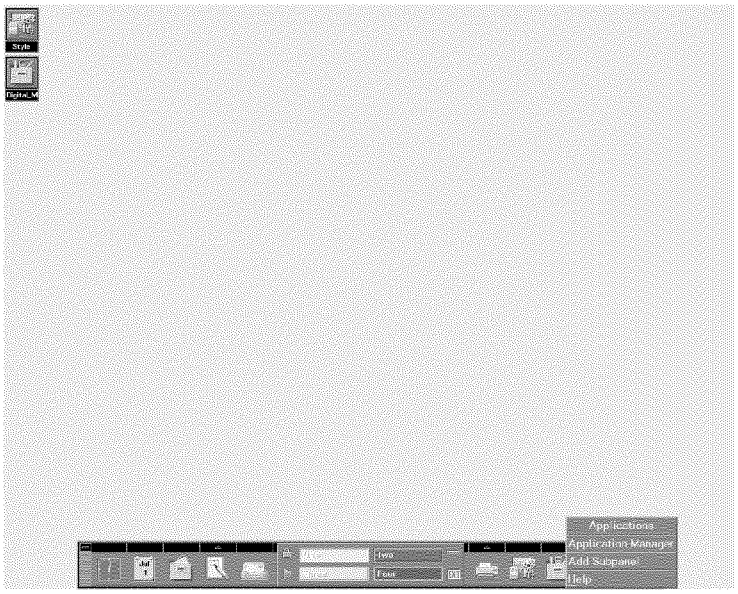
Printing a Document

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.

3-9. SLIDE: Using the Text Editor — Instructor Notes

3-10. SLIDE: Running Applications Using the Application Manager

Running Applications from the Application Manager



a57651

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.

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.

3-10. SLIDE: Running Applications Using the Application Manager

Instructor Notes

Purpose

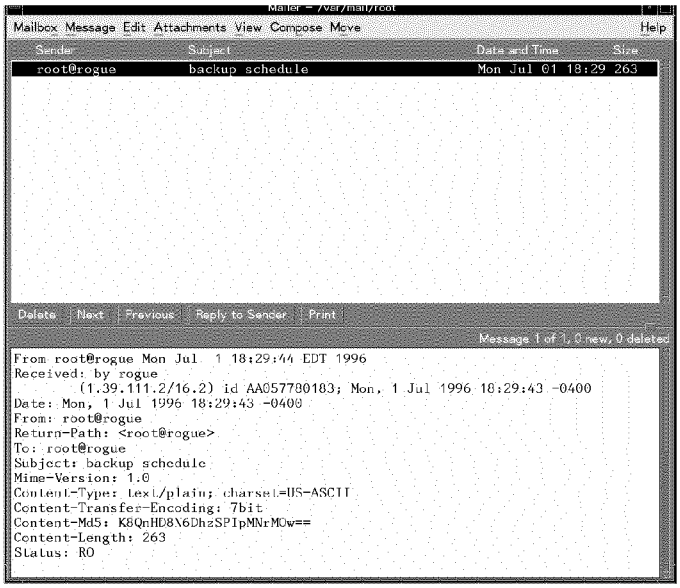
To give an overview of running applications from the Application Manager.

Teaching Tip

- This is not intended to cover how to add applications or provide any customization to the Application Manager.
- Give the students a chance to explore some of the folders, particularly the Desktop_Apps and Desktop_Tools Application Groups to see what is available.

3-11. SLIDE: Using Mailer

Using Mailer



a57652

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 3–7 indicates that there is no unread mail, while the icon in figure 3–8 indicates that there is mail to be read.

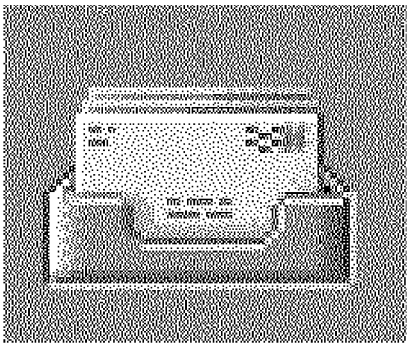

a576134

Figure 3-7.

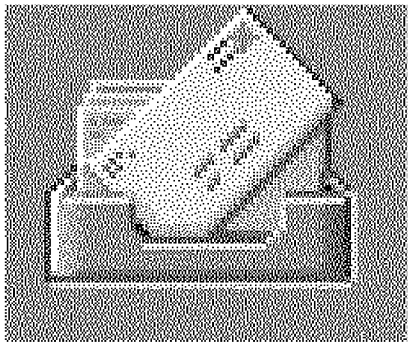

a576135

Figure 3-8.

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 .

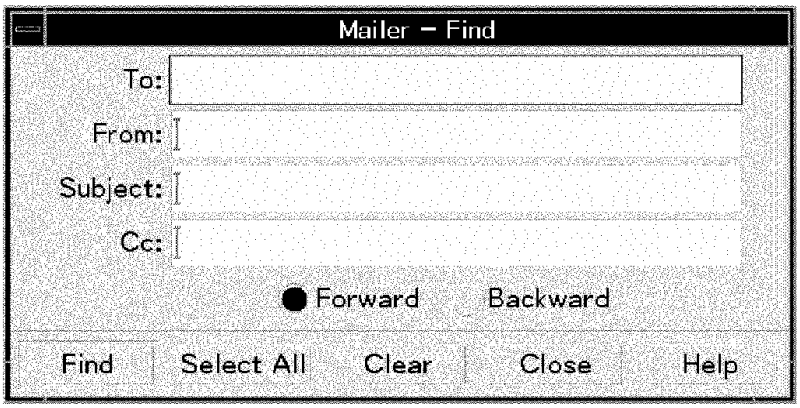


Figure 3-9.

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.

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.
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.

- Choose **Message Header List** from the Category menu of the Mail Options dialog box.

- Select **When I close the mailbox** under Destroy Deleted Messages.
- Click OK or Apply to make your changes take effect.

3-11. SLIDE: Using Mailer — Instructor Notes

Purpose

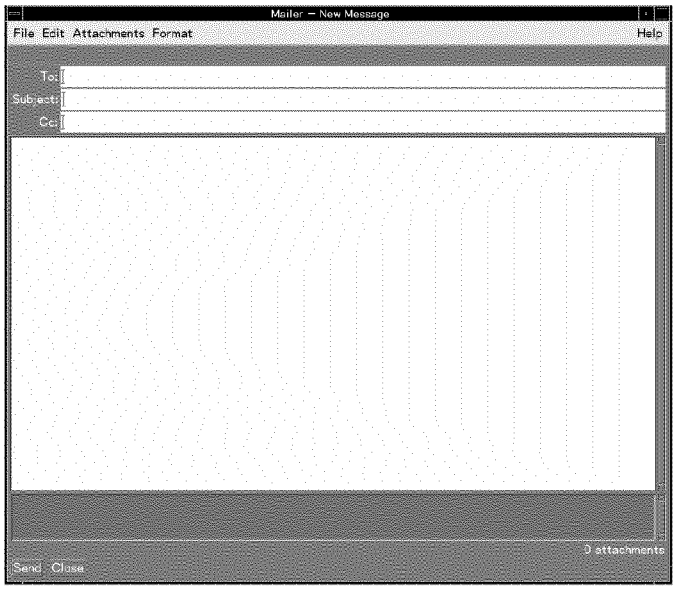
To present the basic Mailer usage

Key Points

- Mailer doesn't care what email tool was used to create the messages sent to the user. For example, other users may have used elm, mailx, cc:mail, etc to create the message. Mailer can still read it.
- When Finding a message, it is possible that Mailer may find messages that contain a superset of the search criteria. For example, if a Find is conducted searching for a message that contains mary in the From field, if there is a message from maryellen it will match the Find.

3-12. SLIDE: Sending Mail

Sending Mail



a57653

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.

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** menubar.
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 4–10).

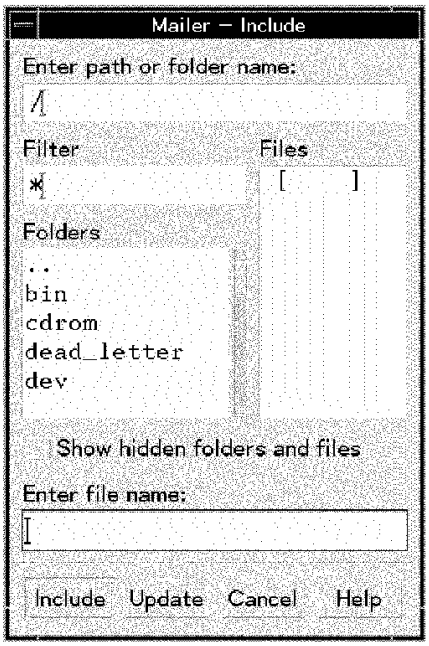


Figure 3-10.

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.

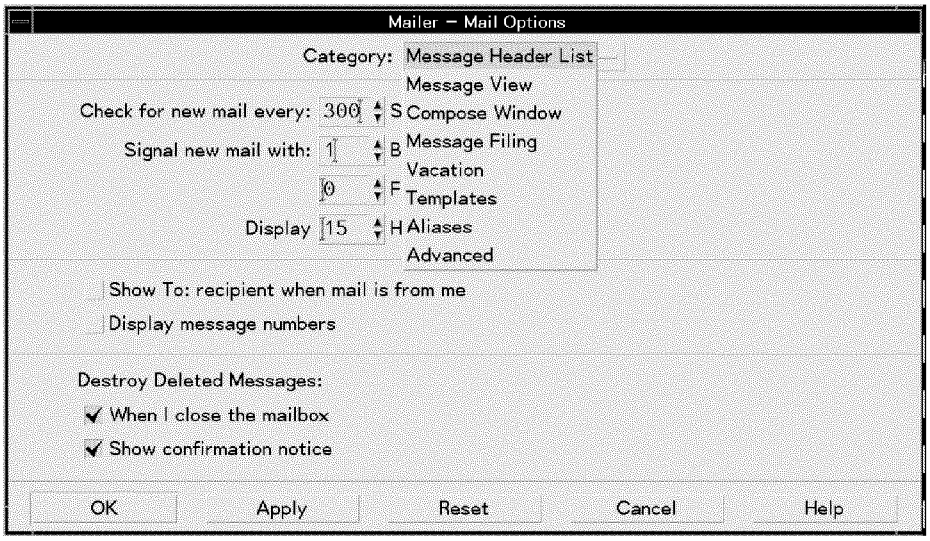
Rather than seeing the text of the file in the Message View area, you will see an attachment at the bottom of the screen with the file name. Double clicking on the attachment icon will open a Text Editor session with that included file.

3-12. SLIDE: Sending Mail — Instructor Notes

Editing the messages works the same way as editing text files. It is probably best to give students an overview of what needs to be done, and encourage them to get hands on experience to feel comfortable.

3-13. SLIDE: Customizing Mailer

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 recipients name, or the senders name.
- If message numbers are displayed or not.
- Whether to automatically destroy deleted messages, or to confirm deletion upon exiting the Mailer.

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.
- 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.

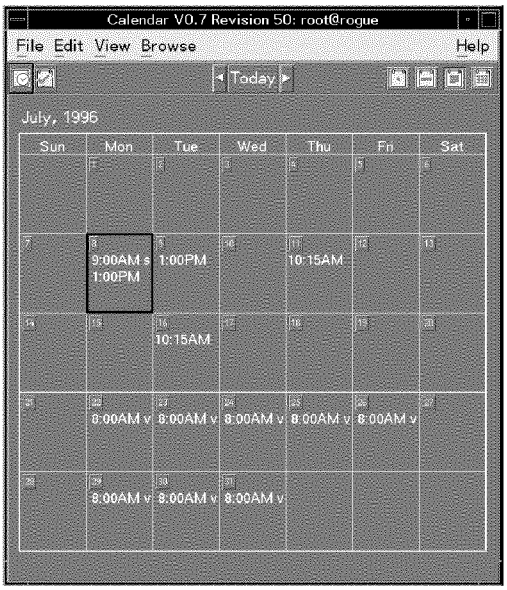
3-13. SLIDE: Customizing Mailer — Instructor Notes

Purpose

To show what options are available for customization. For more details, refer to the B1171-90101 *CDE 1.0 User's Guide*

3-14. SLIDE: Using Calendar

Using Calendar



a57655

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 mini-calendar 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.

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

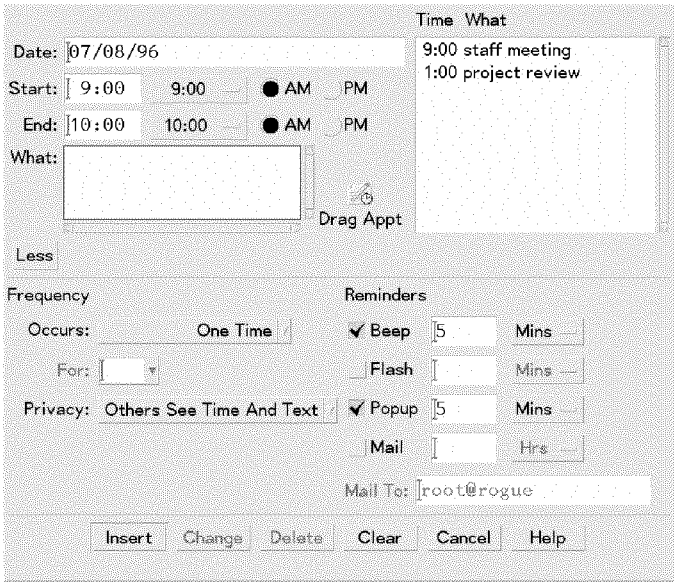
3-14. SLIDE: Using Calendar — Instructor Notes

Purpose

To introduce the calendar management utility. In order to effectively use this utility, the **`rpc.cmsd`** daemon had to have been started. This is usually registered in **`/etc/inetd.conf`** when the desktop is installed and should need no other configuration. This daemon is a small database manager for appointment and resource scheduling.

3-15. SLIDE: Scheduling Appointments

Scheduling Appointments



a57656

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:

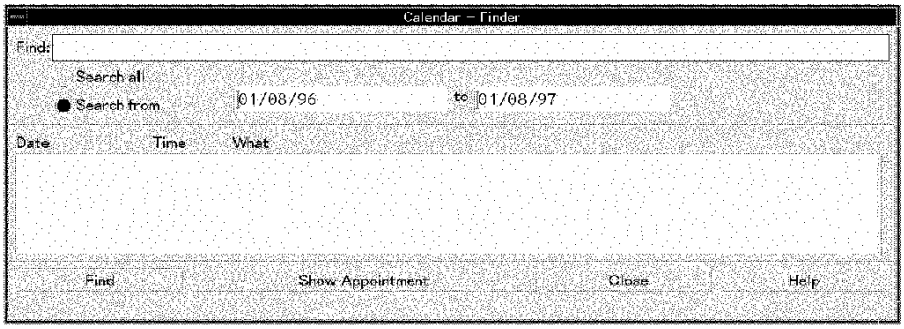
- Display the day you want to make the appointment
- Choose **`Appointment...`** from the Edit menu to activate the Calendar Appointment Editor.
- Choose Start and End times.
- Specify what the Appointment is (this will be truncated in the calendar square).
- If you want additional options, click `More` 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.

Changing or Deleting Appointments

- Display the day you want to change an appointment
- Activate the Appointment Editor
- Highlight the appointment you want to delete or change
- If making a change click on the area you want to change (i.e. start/stop time)
- 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

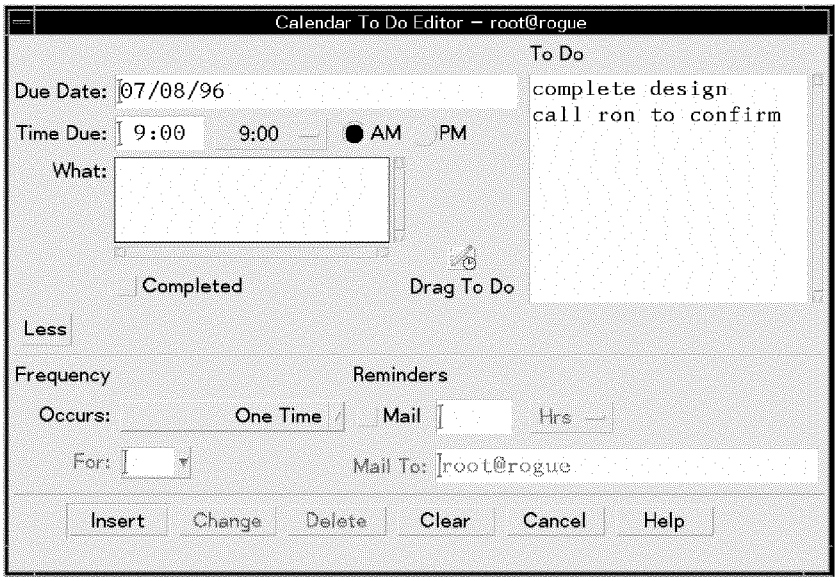
Figure 3-11.

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

3-15. SLIDE: Scheduling Appointments Instructor Notes

3-16. SLIDE: To Do Items

To Do Items



a57657

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).

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

Marking an Item Complete

- Select **To Do List...** from View menu bar

- The dialog box will display all To Do items, Completed To Do items, or Pending To Do items. Select which you want displayed.
- Click on the box to insert a check, which marks complete the To Do item. To remove, click again to toggle off.
- Click OK

alternative method

- Select **To Do ...** from Edit menu to display a given day's To Do items.
- Click Completed to mark the item complete. (This can be toggled on or off)
- Click Change.
- Click Cancel to close the editor.

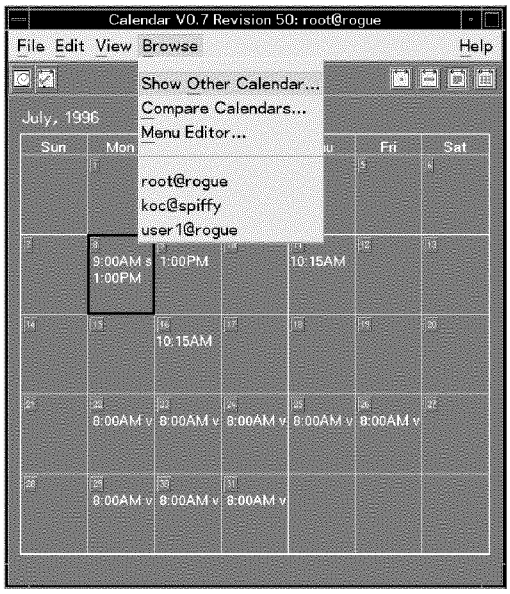
3-16. SLIDE: To Do Items — Instructor Notes

Purpose

To explain the To Do item editor, and how to mark complete. Depending upon what type of view to the calendar the user has (that is, month, week, day) will determine how many To Do items are listed—a month's worth, week's worth, or day's worth.

3-17. SLIDE: Browsing Calendars on a Network

Browsing Calendars on a Network



a57658

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 *calendar-name@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.

- Select **Choose Calendars** from the Browse menu.
- Select the name of the calendar(s) you want to view.
- Calendars are overlayed 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.

- Select one or more entries in the Browse list.
- Click on an unshaded area available to all entries.
- 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.

3-17. SLIDE: Browsing Calendars on a Network — Instructor Notes

Purpose

To learn how to display calendars across the network and schedule appointments in other user's calendars.

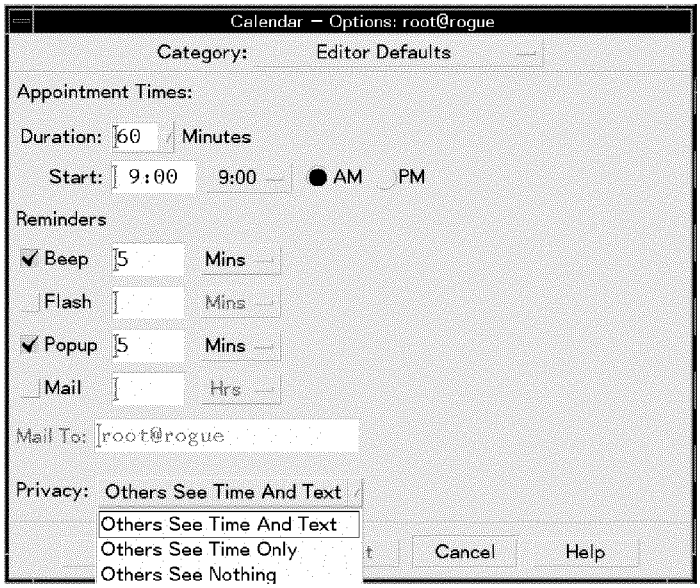
Key Points

Before being able to view others' calendars, as well as schedule appointments in another user's calendar, they would have to grant permission for you to do so.

Users also have the ability to mail reminders to the group by clicking on the `Mail` button. The appointment is integrated into the mail message as an attachment and is pre-addressed to all members of the group.

3-18. SLIDE: Granting Access to your Calendar

Granting Access to your Calendar



a57659

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.

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.

3-18. SLIDE: Granting Access to your Calendar — Instructor Notes

Purpose

To explain how to grant access to your calendar to others on the network.

Key Points

- If you do not grant access Insert or Change access, by default others will not be able to anything other than browse
- To deny access to a user, highlight that entry in the access list and click Delete. To save changes click OK.

3-19. LAB: Using CDE

Lab Objective

To become comfortable using the basic elements of the CDE Desktop Environment

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`**.

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.

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 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.

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.

3-19. LAB: Using CDE — Instructor Notes

Solutions

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

Answer:

1. Position the mouse cursor over the directory **class** and double click.
2. Position the cursor over the file **cde_intro**.
3. Choose **Copy To** from the **Selected** menu. A pop-up dialog box will appear prompting you for the file. Type in **cde_intro2** .
4. Press OK.

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

Answer:

1. Position the cursor over the file **cde_intro2** .
2. Choose **Move To** from the **Selected** menu. A pop-up dialog box will appear prompting you for the destination folder. Type in **cde_dir**.
3. Press OK.

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

Answer:

1. Position the cursor over the folder **cde_dir** and double click to change to that directory.
2. Highlight the file **cde_intro2**.
3. A pop-up dialog menu will appear with the current ownership and permission information, including the size and last modified information.
4. Click *off* the write permission.
5. Press OK.

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**.

Answer:

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

2. Type *graphical environment* in the **File Contents** field.
3. Type **class** following your home directory in **Search Folder:** field.
4. Click Start

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

Answer:

1. Click on **File** menu and choose **Find...**.
2. Type **data*** in the **File or Folder Name:** field.
3. Type **class** following your home directory in **Search Folder:** field.
4. Click Start

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.

Answer:

1. Click on **File** menu and choose **Find...**.
2. Type **data?** in the **File or Folder Name:** field.
3. Type **class** following your home directory in **Search Folder:** field.
4. Click Start

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

Answer:

1. Open the **cde_dir** folder.
2. Highlight the file **cde_intro2**.
3. Either choose the **Put in Trash** menu item from the **Selected** menu *OR* click and hold mouse button 1 while dragging the icon down to the Trash Can. Once the file icon is over the trash can release the mouse button.

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

Answer:

1. Double click on the Trash Can icon in the Front Panel to open Trash Can window.
2. Select the file **cde_intro2** to restore.
3. Click on **File** to open the menu bar, then select **Put Back**. The file will return to its original location.

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

   Answer:

   1. Open the **cde_dir** folder.
   2. Highlight the file **cde_intro2**.
   3. Either choose the **Put in Trash** menu item from the **Selected** menu *OR* click and hold mouse button 1 while dragging the icon down to the Trash Can. Once the file icon is over the trash can release the mouse button.
   4. Double click on the Trash Can icon in the Front Panel to open Trash Can window.
   5. Select the **cde_intro2** file to restore.
   6. Click on **File** to open the menu bar, then select **Shred**. The file will be permanently removed.

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.

    Answer:

    1. Position the mouse cursor to the beginning of the first paragraph. Press and hold mouse button 1 while dragging the cursor across the area to be copied. Release the button.
    2. Choose **Copy** from the Edit menu. A copy of the tet is stored on a clipboard.
    3. Position the cursor to the end of the file.
    4. Choose **Paste** from the Edit menu.

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

    Answer:

    1. Choose **Find/Change ...** from the Edit menu.
    2. Type **CDE** in the **Find** field.
    3. Type **Common Desktop Environment** in the **Change** field.
    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.

12. Correct all misspelled words in the file.

Answer:

1. Choose **Check Spelling** from the **Edit** menu. The Spell dialog box will be displayed.
2. Type the correct work 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.

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

Answer:

1. Position the mouse cursor on a portion of the workspace switch that is not occupied by other controls or workspace buttons and press mouse button 3. Choose **Add Workspace**
2. Position the mouse cursor on the new workspace called **New** and press mouse button 3. Choose **Rename**.

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

Answer:

1. Click on the up arrow above the Text Editor control on the Front Panel.
2. Position the cursor next to the Terminal icon and press mouse button 3.
3. Select **Copy to Main Panel**.
4. Click on the down arrow above the Terminal control on the Front Panel to close the Personal Applications subpanel menu.

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

Answer:

1. Open the Application Manager.
2. Double-click the Desktop_Apps group icon to display its contents.
3. Scroll down until you see the **Man Page Viewer** action icon.
4. Double-click the action icon to execute the application.
5. Type in **ls** in the dialog window to execute the action.

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

Answer:

1. Choose **New Message** from the Compose Menu
2. Enter your mail partner's email address in the **To** field and the subject in the **Subject** field.
3. Once your have addressed the message, press Return to go to the text area and compose the message.
4. Click the Send button.

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

Answer:

To send the reply:

1. Select the message for reply.
2. From the Compose menu choose **Reply to Sender**
3. Enter reply.
4. Click Send.

To forward the message:

1. Select the message to forward.
2. Choose **Forward Message** from the Compose menu.
3. Enter the mail address for the recipients in the To: field.
4. Include your own comments if desired.
5. Click Send.

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.

Answer:

Once you have created the file using Text Editor, do the following to create the template:

1. In the Mailer, click **Mail Options** from the **Mailbox** menubar.

2. Click the **Category** button and choose **Templates**. The template dialog box will appear.
3. Type the name **monthly** in **Menu Label** field.
4. Enter the name of the file **status** in the **File/Path:** field.
5. Click Add to include the template in the list of templates.

To use the template:

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

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.

Answer:

1. Open the calendar to display the current month. Click on 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.
5. If you want this to occur on a regular basis, click More .

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.

Answer:

1. Choose **Menu Editor** from the Browse Menu.
2. Type **calendar-name@hostname** in the **User Name** field.
3. Click Add Name.
4. Click OK.
5. Repeat for your calendar and your mail partner's.

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.

Answer:

1. Select **Compare Calendars** from the Browse menu.
2. Select the name of the calendars you want to view.

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

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.

Answer:

1. Choose **Options...** from the File menu.
2. From the Category menu, choose **Access List** to display the Access List and Permissions dialog box.
3. In the **User Name** field, type **calendar-name@hostname** for the calendar to which you want to grant access.
4. Select View, Insert, and Change permissions.
5. Click Add to add the calendar to the Access List with the permissions you've chosen.
6. Click Apply.
7. Repeat for yourself so that you can overlay your calendar with your mail partner's.

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

Answer:

1. Browse your menu together with your mail partner's menu
2. Click on an unshaded area available to both your calendars
3. Click Schedule. The Calendar Group Appointment Editor will be displayed. A **Y** in the Access column means that you have insert access to update their schedule. An **N** means that you do not. If you do not have insert access, remind your mail partner to grant you access.

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.

Overview of Module 4

Audience

general user General system users

Product Family Type

open sys Open systems environment

Abstract

To many first-time UNIX system users, the file system is one of the most difficult things to comprehend. This module is designed to allow the students to successfully navigate and use the hierarchical structure. The focus here is on directories.

Time

Lab 45 minutes

Lecture 45 minutes

Prerequisites

m45m Logging In/Orientation

In order to successfully complete this module, the student must be able to log in.

Instructor Profile

UX General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000 Logon access to an HP-UX system

Software Requirements

UX11 HP-UX release 11.0

Material List

P/N B2355-90033(T) — *HP-UX Reference Manual* , one per terminal

Lab Instructions

setup1 — Create user logon of **user1**, **user2**, ... **user***n*, where *n* is the number of students in the class. Set up one user per student.

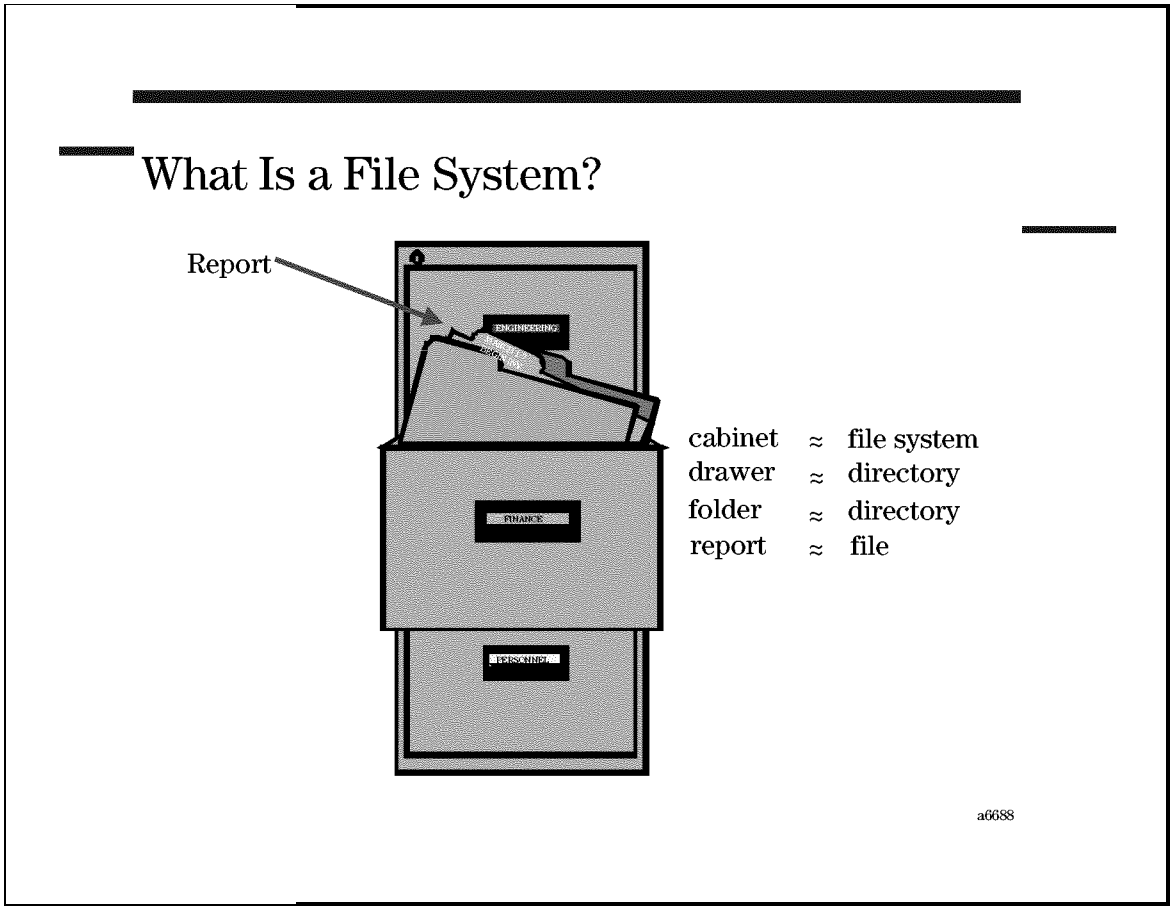
copyfiles — Copy the lab files to the users' home directories.

Lab Files

```
total 14
drwxr-xr-x 5 karenk users 1024 May 28 16:12 car.models
-rw-r--r-- 1 karenk users 17 May 28 16:12 cherry
-rw-r--r-- 1 karenk users 17 May 28 16:12 collie
drwxr-xr-x 4 karenk users 1024 May 28 16:12 dog.breeds
-rw-r--r-- 1 karenk users 17 May 28 16:12 poodle
-rw-r--r-- 1 karenk users 17 May 28 16:12 probe
-rw-r--r-- 1 karenk users 17 May 28 16:12 taurus

tree/car.models:
total 6
drwxr-xr-x 2 karenk users 24 May 28 16:12 chrysler
drwxr-xr-x 4 karenk users 1024 May 28 16:12 ford
drwxr-xr-x 2 karenk users 24 May 28 16:12 gm

tree/car.models/chrysler:
total 0tree/car.models/ford:
total 4
drwxr-xr-x 2 karenk users 24 May 28 16:12 sedan
drwxr-xr-x 2 karenk users 1024 May 28 16:12 sports

tree/car.models/ford/sedan:
total 0tree/car.models/ford/sports:
total 2
-rw-r--r-- 1 karenk users 18 May 28 16:12 mustang

tree/car.models/gm:
total 0tree/dog.breeds:
total 4
drwxr-xr-x 2 karenk users 1024 May 28 16:12 retriever
drwxr-xr-x 2 karenk users 24 May 28 16:12 shepherd

tree/dog.breeds/retriever:
total 6
-rw-r--r-- 1 karenk users 27 May 28 16:12 golden
-rw-r--r-- 1 karenk users 29 May 28 16:12 labrador
-rw-r--r-- 1 karenk users 26 May 28 16:12 mixed
```

```
tree/dog.breeds/shepherd:
total 0
```

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**.

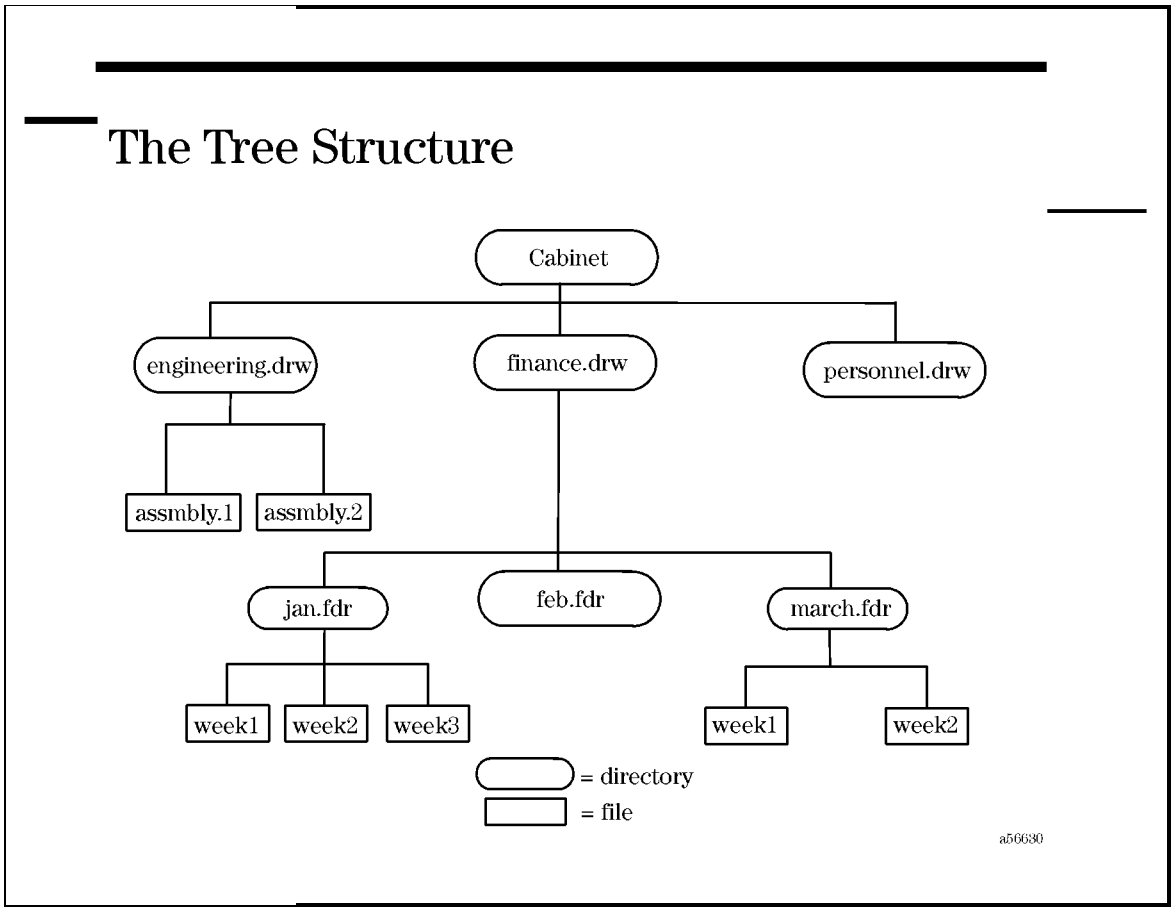
A UNIX system's file system is very similar to a file cabinet. The entire file system is analogous to the file cabinet, as it contains all of the drawers, file folders, and files. A drawer is similar to a subdirectory in that it can contain reports or file folders. A file folder would also represent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

4-1. SLIDE: What Is a File System? Instructor Notes

Teaching Tips

- Describe a UNIX system's file system. The "file cabinet" analogy may help.
- Define the concept of a **file** and a **directory**.

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.

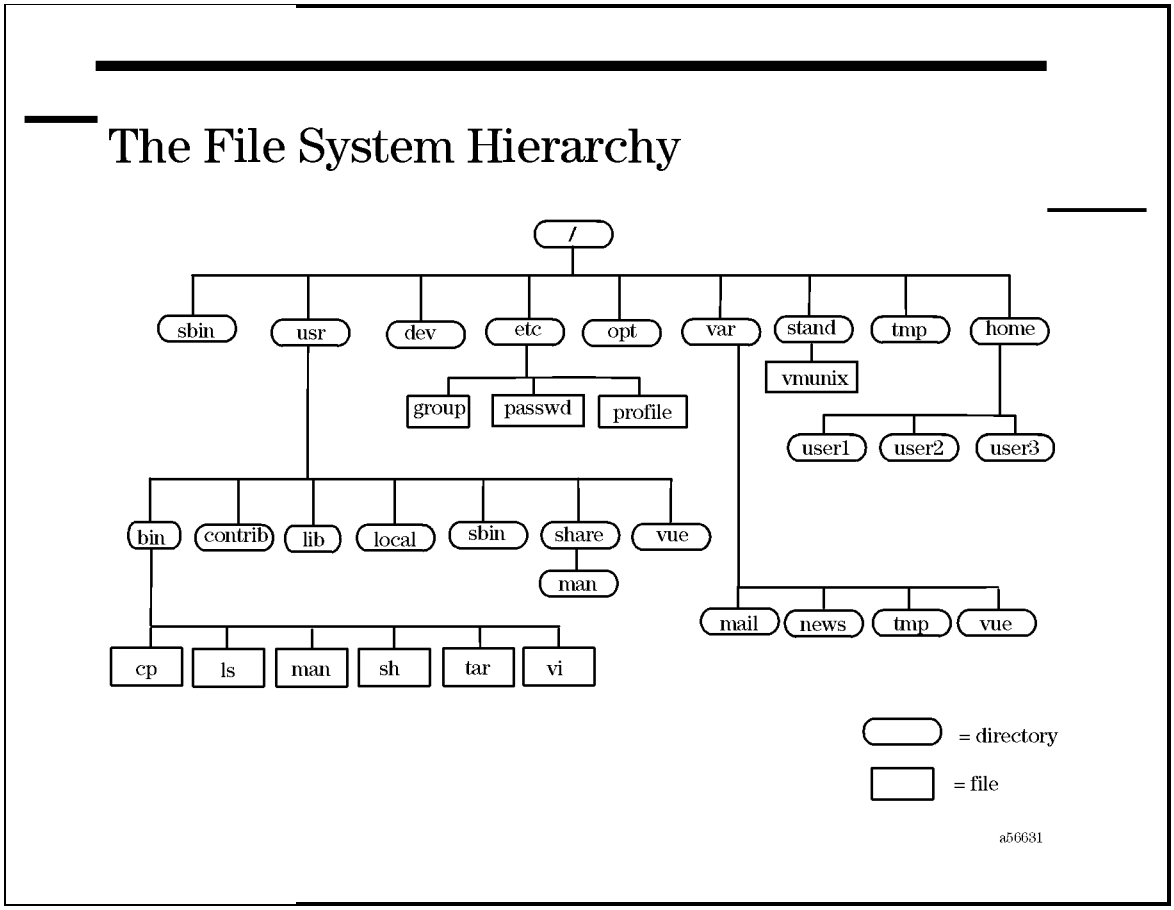
The slide illustrates a graphical tree representation of the filing cabinet from the first slide.

4-2. SLIDE: The Tree Structure Instructor Notes

Key Points

- A directory can store files and other directories (subdirectories).
- Every entity in the tree will be a file or directory.

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 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 and 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**.
/opt	This directory will contain applications and products. The developers and the administrators of HP-UX system will use it to install new products or local applications.
/usr/bin	This directory contains the programs for all reference manual section 1 commands that are necessary for basic UNIX system operation and file manipulation. These are normally accessible by all users. ("bin" is short for binary).
/usr/sbin	This directory contains the programs for all reference manual section 1m commands. They are system administration commands. You must be super-user to use many of them. These are documented in the reference manual sections 1m .
/usr/lib	This directory contains archive and shared libraries used for applications.
/usr/share	This directory contains vendor independent files (the most important is the manual).
/usr/share/man	This directory contains all files associated with the online manual pages.
/usr/local/bin	This directory usually stores locally developed programs and utilities.
/usr/contrib/bin	This directory usually stores public programs and utilities. You might retrieve these from a bulletin board service or a user group.
/sbin	This directory 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	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. The **/home** directory normally contains one subdirectory for each user account on the system. 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 underneath your *HOME* directory; at a maximum, you will be able to

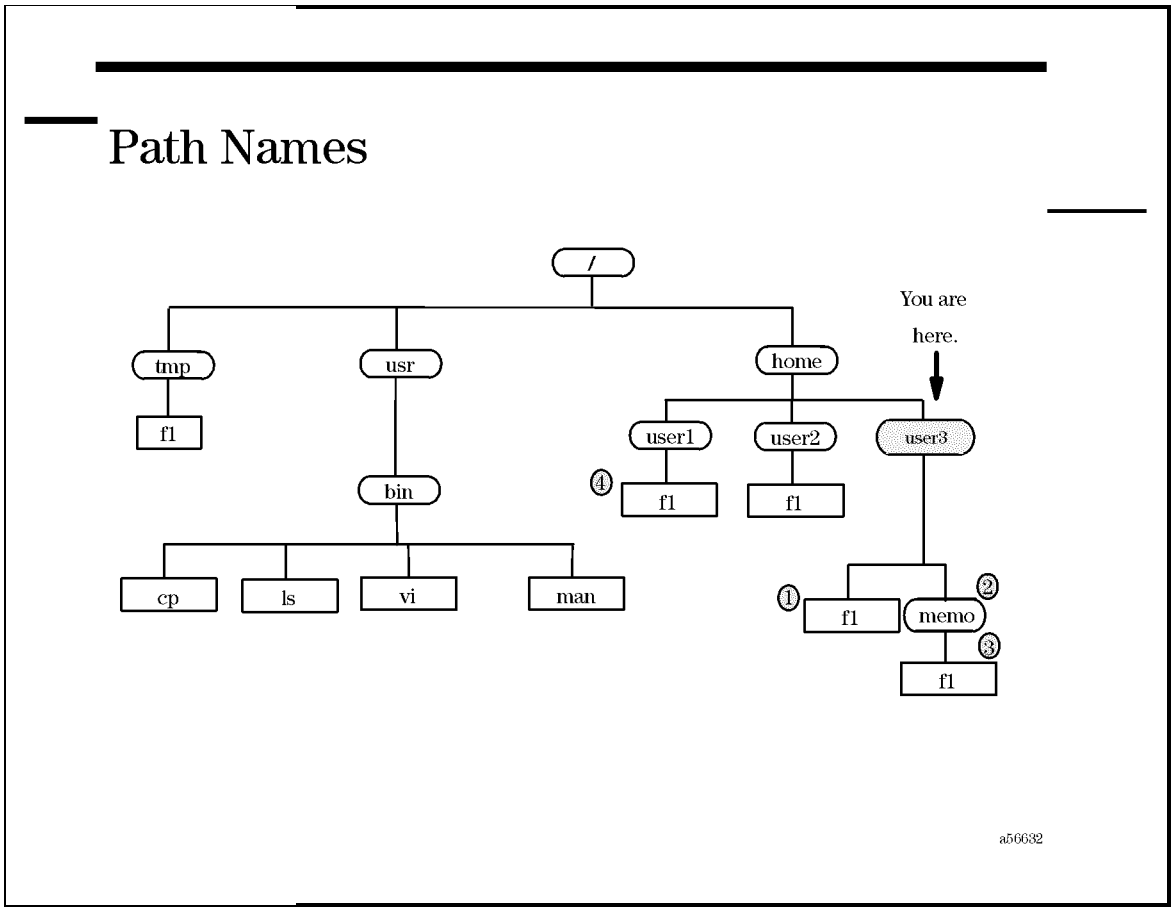
	move to *any* directory in the UNIX system hierarchy (the default). It is up to your system administrator to restrict users' access to specific directories on the system.
/etc	This directory holds many of the system configuration files. These are documented in the reference manual sections 4.
/stand/vmunix	This file 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	This directory commonly is used as a scratch space for Operating System 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	This directory 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	This directory will be used to mount other devices (laserROM for instance).
/var/mail	This directory contains a "mailbox" for each user who has incoming mail.
/var/news	This directory contains all of the files representing the current news messages. Their contents would all be displayed by entering **news -a**.
/var/tmp	This directory commonly is used as a scratch space for users.

4-3. SLIDE: The File System Hierarchy Instructor Notes

Key Points

- A UNIX system's file system is hierarchical, as in the file cabinet example.
- The UNIX system uses a conventional file system hierarchy to store the hundreds of files and programs that make up a complete UNIX system distribution.
- The top of the directory structure is called **`root`** and is denoted with a single forward slash (**`/`**).
- The HP-UX kernel is stored in the file called **`/stand/vmunix`** .
- Directories are available to users to organize their files.
- Commands are available to users to easily create new directories as needed and move or copy files from one directory to another.
- Directories can contain files and other directories (subdirectories). There is no limit to the number of subdirectories that you can create.
- Each user has a *HOME* directory. The user has complete control over all items created under his or her *HOME* directory.
- Users usually have access to all directories on the system.

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 ...]
```

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 will be 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, you will be initially 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 **/**
- not dependent on your current location in the hierarchy
- always is 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
- will never start with a **/**
- is unique relative to your current location only
- is often shorter than the absolute path name

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, but the UNIX system knows which one to retrieve because it knows that if you are currently located in the directory **/home/user1** to retrieve **/home/user1/f1** or if you are currently located in the directory **/home/user3/memo** 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: It is important that you know what directory you are currently located when accessing files with relative path names to ensure that you are accessing the correct file 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.

4-4. SLIDE: Path Names — Instructor Notes

Key Points

- Go through the bulleted items in the student notes that define and differentiate absolute path names from relative path names.
- Using the hierarchy on the slide, illustrate the differences between the relative and absolute paths.
- Stress that absolute paths are not dependent on your current location, and that relative paths are.
- When using relative path names, it is important that you know where you are in the hierarchy, especially when there are many files on the system with the same name stored under many directories. You might want to use file removal as a good example. You intend to remove the file **`f1`** that is under **`/home/user3/memo`**, but you are in the directory **`/home/user1`** and you issue the command: **`rm f1`**. You have just removed the wrong file.
- Relative paths are often used to save typing.

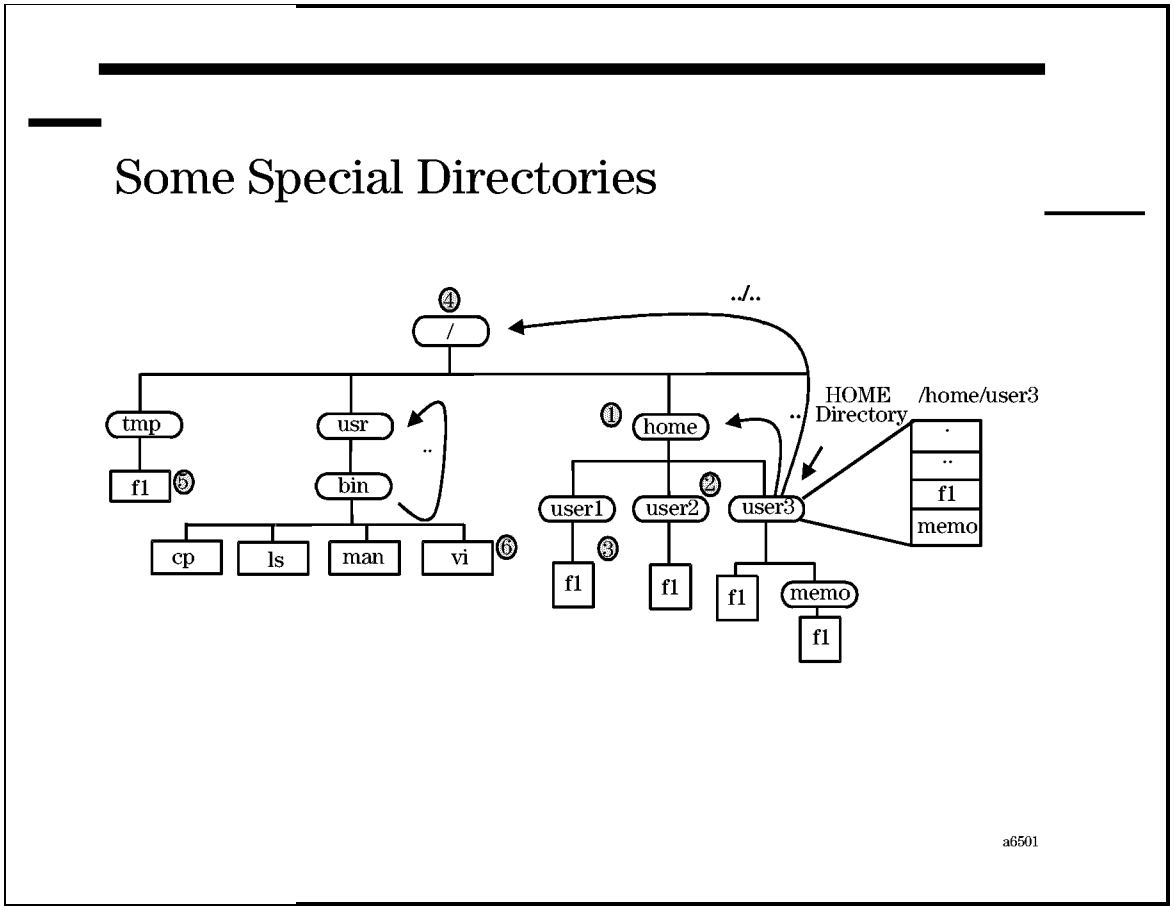
Teaching Questions

- If your current directory is **`/`**, what would be the relative path designations to all of the files named **`f1`**?

```
tmp/f1
home/user1/f1
home/user2/f1
home/user3/f1
home/user3/memo/f1
```

Notice that relative path names do not save you anything here.

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.

Login Directory

When a new user is added to the system, he or she will be assigned a login ID and possibly a password, and 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 any 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 just 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 /
../user2    represents /home/user2
../user1/   represents /home/user1/f1
f1
../../tmp/  represents /tmp/f1
f1
```

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

4-5. SLIDE: Some Special Directories Instructor Notes

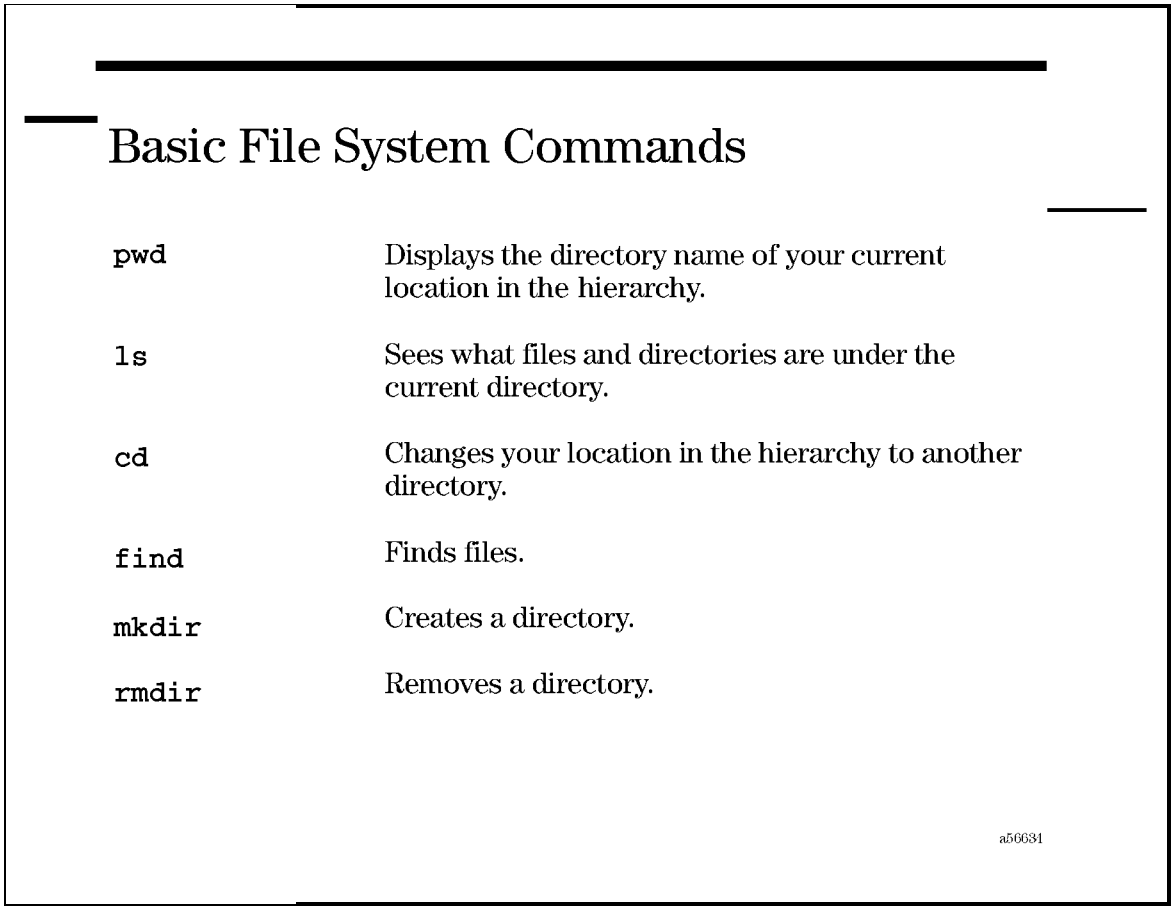
Key Points

- Discuss the representation of dot and dot dot.
- Use the examples on the hierarchy slide and in the student notes.
- Stress that dot and dot dot are an extension for relative paths, and are therefore dependent on your current location in the hierarchy.
- Dot dot allows you to move up through the hierarchy.

Other Examples

You might want to review some of the other directories, and point out their `..` directories and `../..` directories. `/tmp` is illustrated as an alternate example.

4-6. SLIDE: Basic File System Commands



Student Notes

A directory, like a file folder, is a way to organize your files. The remainder of this module will introduce basic directory manipulation commands so that you can:

- 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.

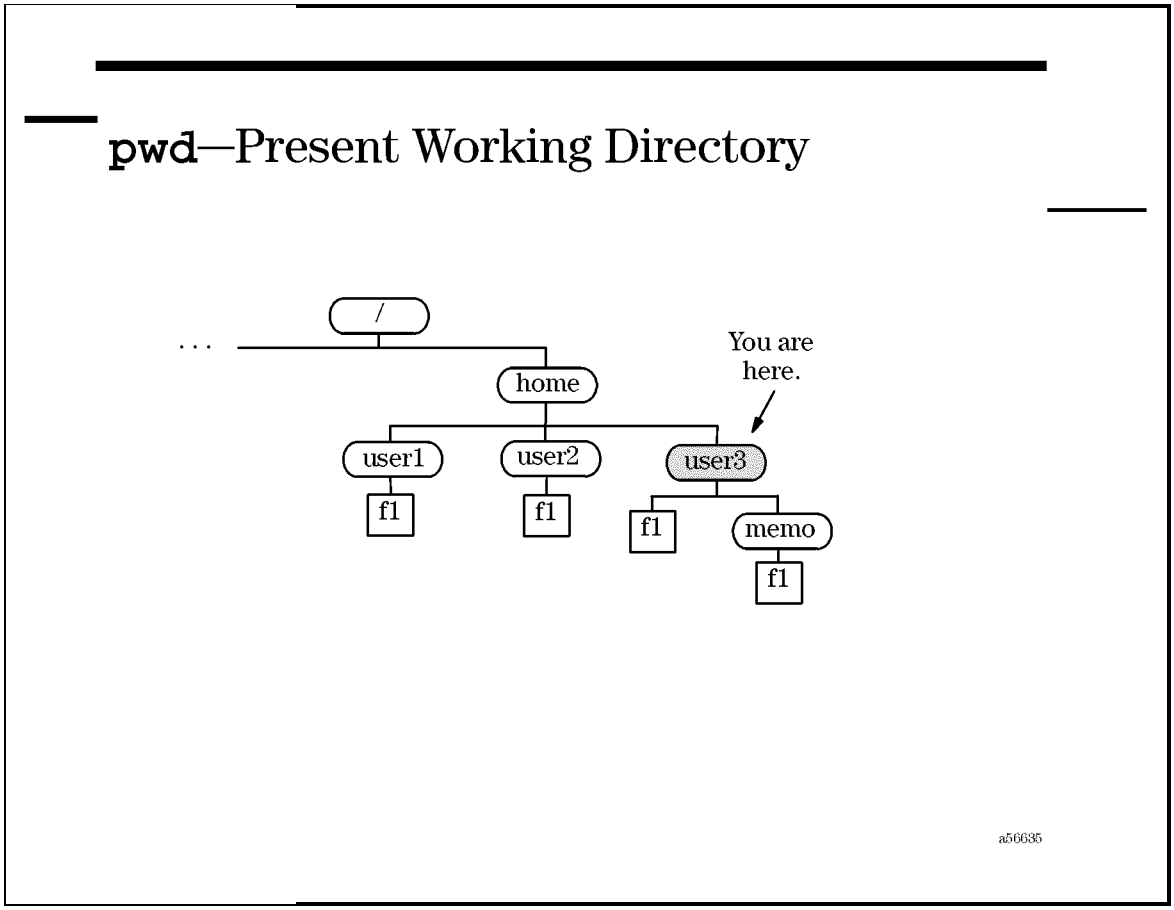
In this module we will not deal with the files within a directory. We will examine directories only.

4-6. SLIDE: Basic File System Commands Instructor Notes

Teaching Tips

Point out that we are examining *only* directories.

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 and 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 should issue this command frequently to display their location as they move through the file system.

4-7. SLIDE: pwd — Present Working Directory — Instructor Notes

Teaching Tips

Point out that most UNIX system commands are a mnemonic with an associated meaning. The students are often able to learn and remember the new commands if they also know the expanded representation of each command.

As new commands are presented, an expanded translation will also be provided.

Key Points

- **pwd** represents present working directory.
- **pwd** always displays the absolute path of your current location in the hierarchy.
- This command is very important for new users so that they know where they are in the file system.

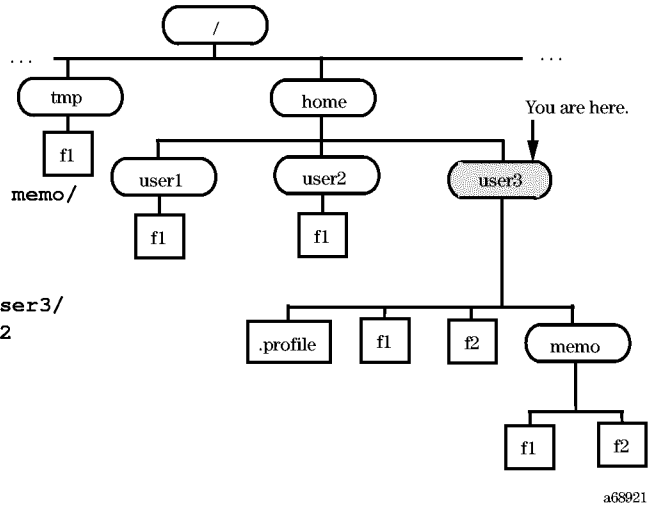
4-8. SLIDE: `ls` — List Contents of a Directory

`ls` – List Contents of a Directory

Syntax:
`ls [-adlFR] [pathname(s)]`

Example:

```
$ ls
f1 f2 memo
$ ls -F
f1 f2* memo/
$ ls -aF
.profile f1 f2* memo/
$ ls memo
f1 f2
$ ls -F /home
user1/ user2/ user3/
$ ls -F ../user2
f1
```



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` will report information associated with the designated file. When the path of a directory is provided, `ls` will display 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. These commonly hold configuration information for your user session or applications.

-d Lists characteristics of the directory, instead of the contents of the directory. Often used with **-l** to display 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 dir
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:

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

4-8. SLIDE: `ls` — List Contents of a Directory — Instructor Notes

Teaching Tips

Point out that there are many options to the **`ls`** command; have the class look up the **`ls`** command in the manual to prove it.

Key Points

- The **`ls`** command accepts relative or absolute path names as arguments.
- The **`ls`** command accepts multiple options on a single command line.
- This is the first opportunity students have had to see the use of relative and absolute paths.
- Distinguish between the output of a directory argument versus a file argument.
- Describe hidden files and how they can be displayed.
- When listing out the contents of another directory, you have *not* changed your position in the hierarchy. This requires the **`cd`** command, which is presented on the next slide.

4-9. SLIDE: `cd` — Change Directory

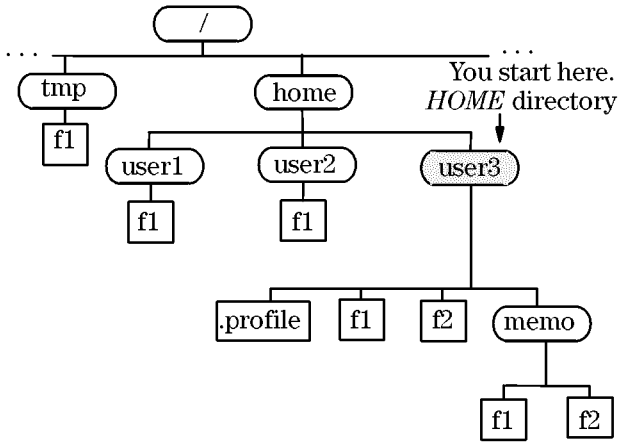
cd—Change Directory

Syntax:

```
cd [dir_pathname]
```

Example:

```
$ pwd
/home/user3
$ cd memo; pwd
/home/user3/memo
$ cd ../..; pwd
/home
$ cd /tmp; pwd
/tmp
$ cd; pwd
/home/user3
```



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. So 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.

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
```

4-9. SLIDE: `cd` — Change Directory Instructor Notes

Teaching Tips

When using the **cd** command it is important to use the **pwd** command to confirm that you are located in the directory to which you intended to move.

When presenting the examples on the slide you also might want to provide the corresponding absolute or relative path.

Ask what directory you would be in after executing each example on the slide.

```
$ cd memo          /home/user3/memo
$ cd ../..         /home
$ cd /tmp          /tmp, could also have issued cd ../tmp
$ cd               /home/user3
```

You might also want to present the use of the semicolon (;) to enter multiple commands on a single line. This would allow you to change directory and display the directory changed to:

```
$ cd memo; pwd
/home/user3/memo
```

Many users like to be able to return to a previous directory. The **cd -** command satisfies this need.

The **cd** command also allows string substitution from the previous **cd** command execution. **cd** accepts two arguments: **cd** *old new*.

```
$ cd /home/user3/tree
$ cd user3 lisa                    Replaces user3 with lisa in previous cd command
$ pwd
/home/lisa/tree
```

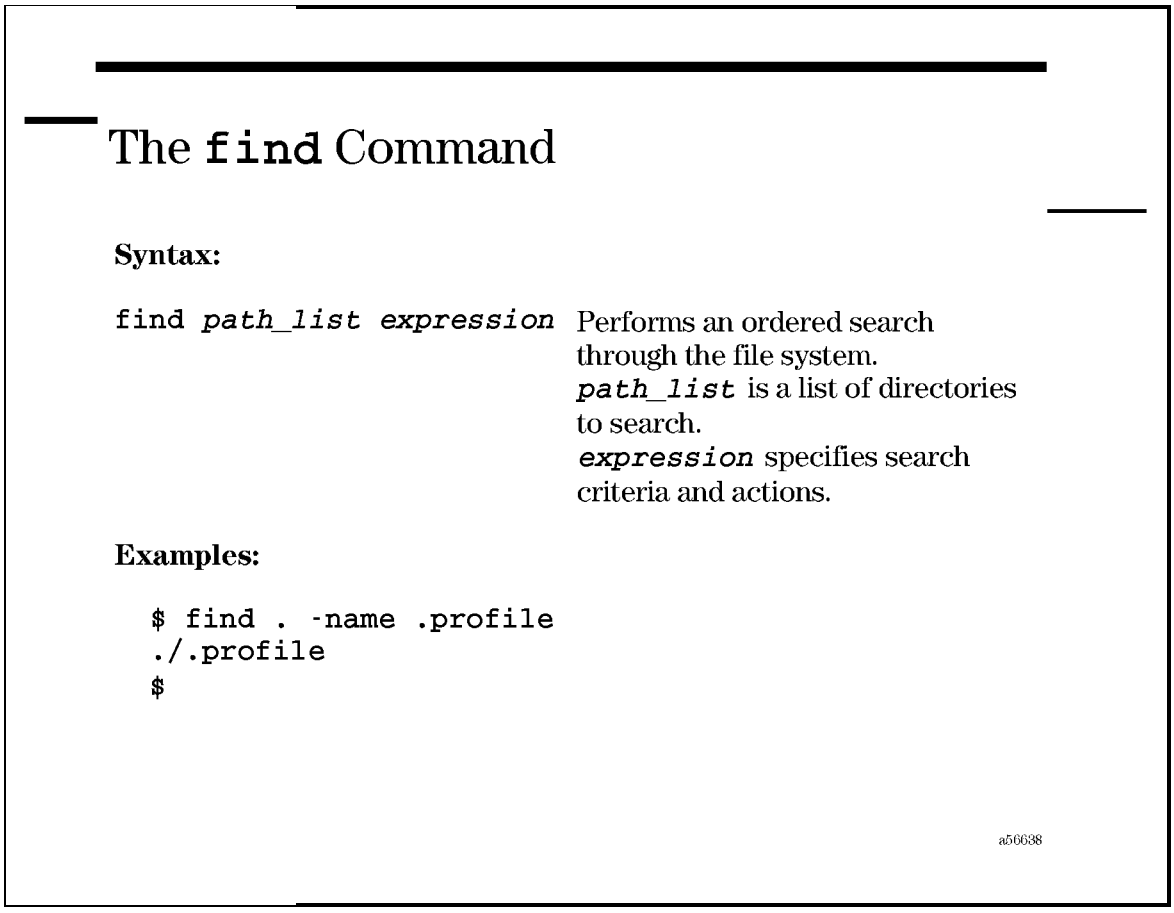
Key Points

- **cd** accepts relative and absolute path names.
- **cd** with no arguments takes you *HOME* .

Teaching Questions

Can you **cd** to a file?
NO—you get an error message.

4-10. SLIDE: The `find` Command



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.

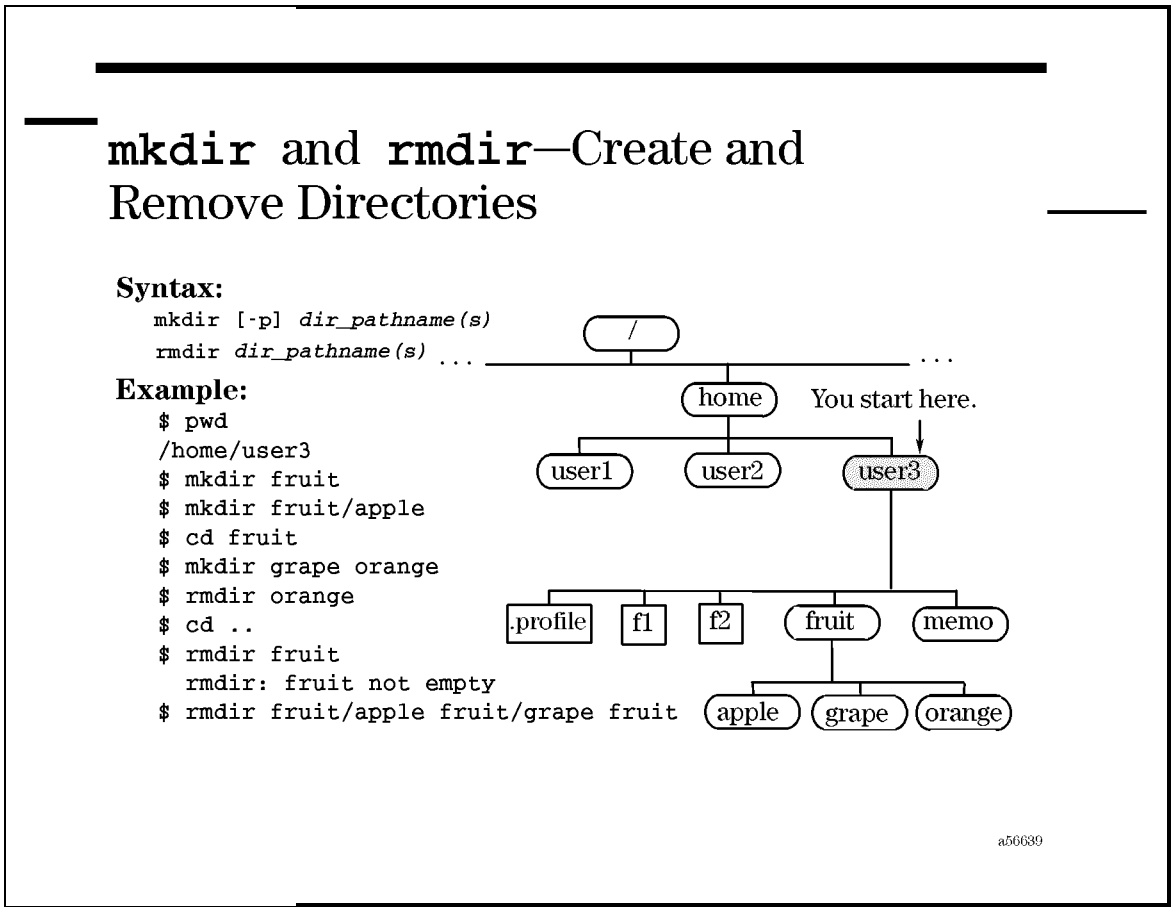
4-10. SLIDE: The `find` Command — Instructor Notes

Teaching Tips

We introduce **`find`** here because many students, when they find out that **`whereis`** does *not* search the whole file system, want to know what command *will* search the entire file system. **`find`** is often used when backing up files.

It may be a good idea to have the students do a **`man`** on **`find`**, or look it up. You can then point out the great power of **`find`** by briefly reviewing all or some of the options and suggesting real-life scenarios in which a particular option might be useful. The *size* option is a good choice to explain.

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



Student Notes

The **`mkdir`** command allows you to make 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. Alternatively, you can use the following option:

`-p`	This 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 would make the **fruit** directory if it does not already exist:

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

The **rmdir** command allows you to remove a directory. Directories must be empty (that is, hold no entries except dot and dot dot) in order to be removed. Also, 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, absolute or relative path names can be provided.

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

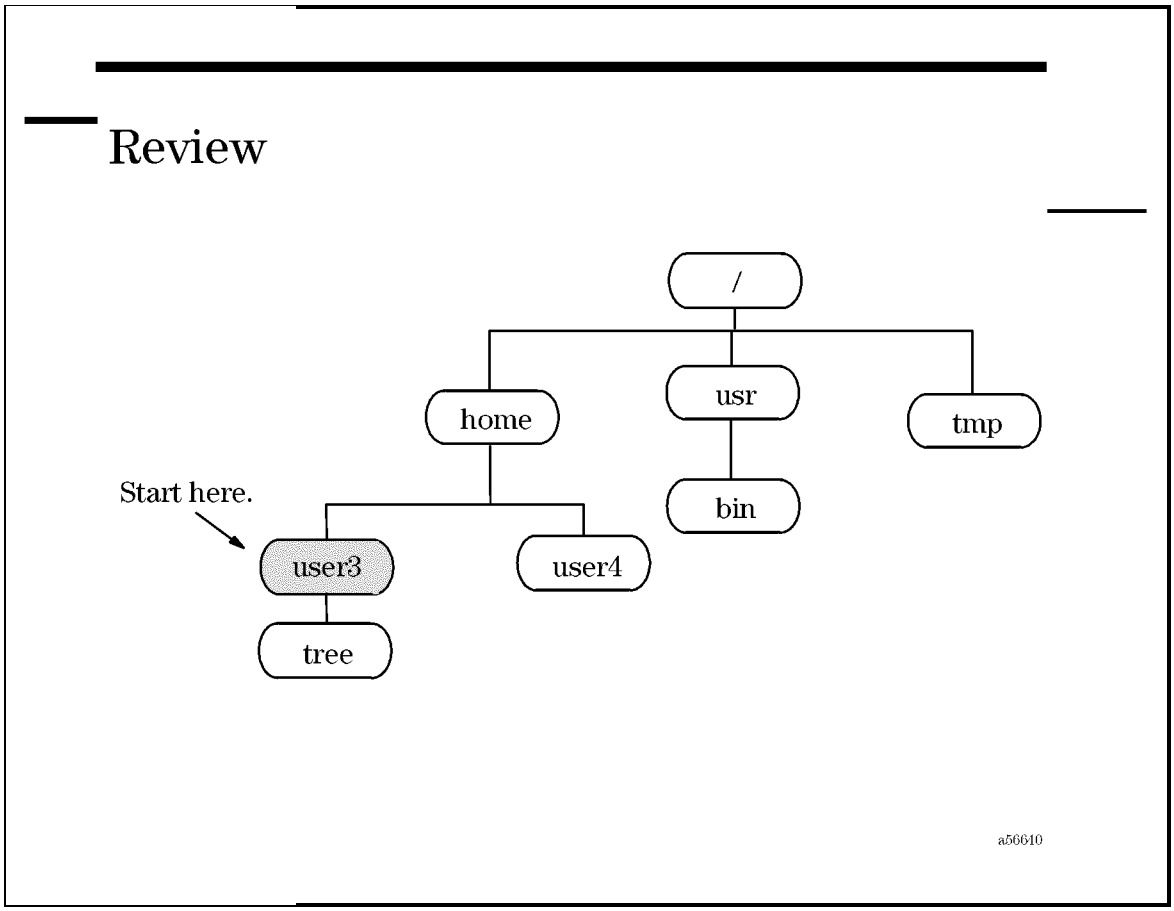
Instructor Notes

Teaching Tips

Point out that directories must be empty (except for the dot and dot dot subdirectories) to be removed. Also point out that the directory to be removed cannot be anyone's current directory.

mode is referred to in the student notes discussion of options. File permissions are covered in more detail in the File Permissions and Access module.

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       _________________
```

4-12. SLIDE: Review Instructor Notes

Teaching Tips

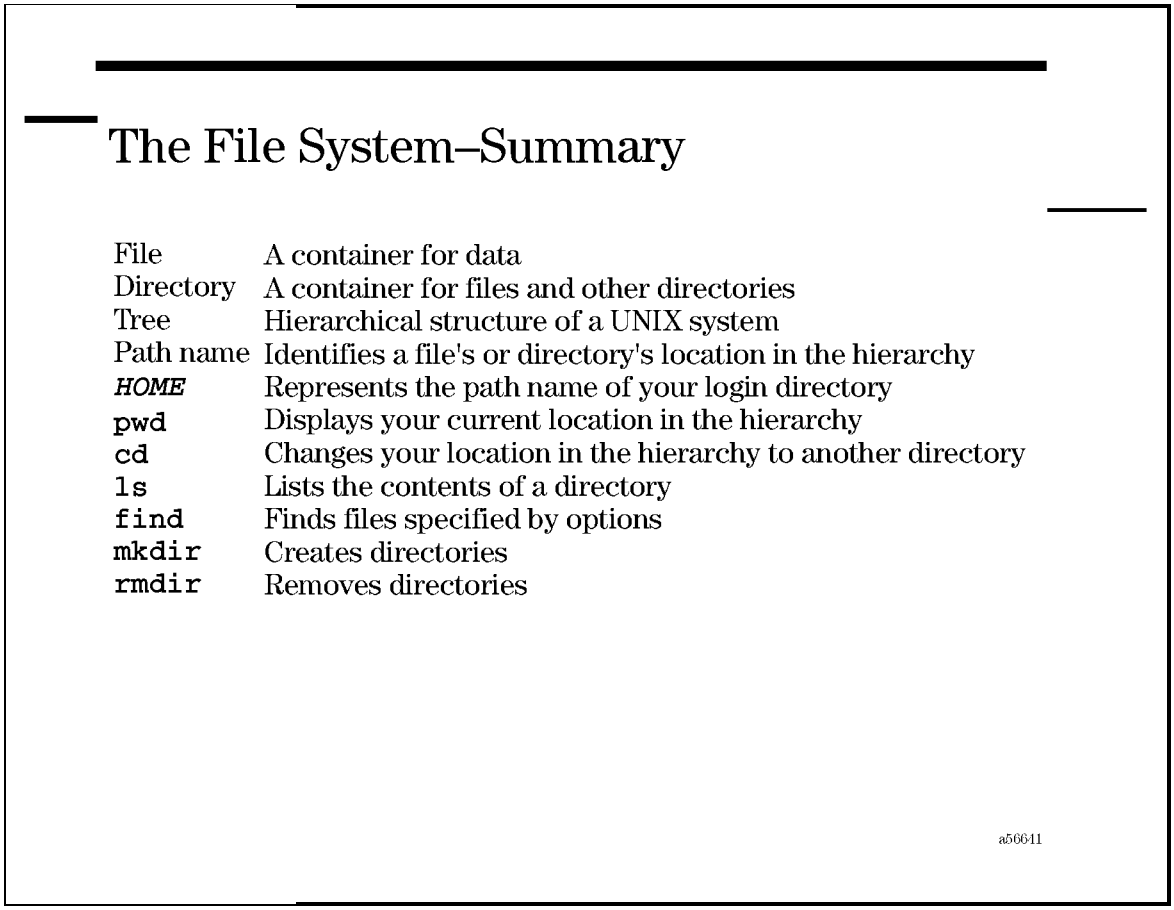
Have the class think about the slide for a minute, then ask some selected students for the answers. The responses should be something like:

For	Response
`$ pwd`	`/home/user3`
`$ cd ..` `$ pwd`	`/home`
`$ cd usr`	sh: usr: not found
`$ pwd`	`/home`
`$ cd /usr` `$ pwd`	`/usr`

`$ cd ../`
`tmp`
`$ pwd`
	`/tmp`

`$ cd .` `$ pwd`	`/tmp`

4-13. SLIDE: The File System — Summary



Student Notes

4-13. SLIDE: The File System — Summary Instructor Notes

This page can serve as a handy reference for students as they learn the UNIX system commands.

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?

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, remove any directories created as a result of your effort before trying again. 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 could this be 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.

4-14. LAB: The File System — Instructor Notes

Time: 30 minutes

Purpose

To practice maneuvering through the hierarchical file structure. These exercises will require the use of the **pwd**, **cd**, and **ls** commands.

Solutions

1. What is the name of your *HOME* directory?

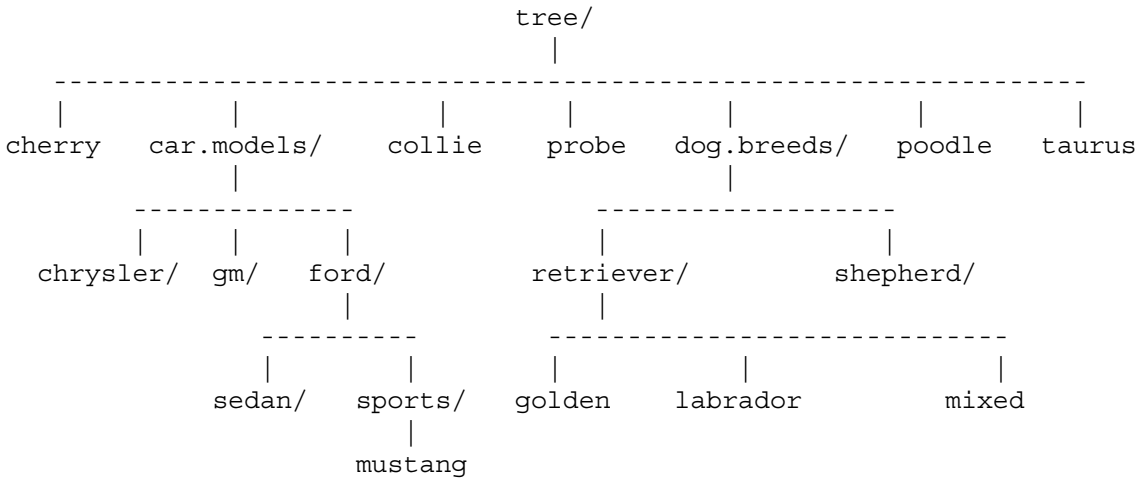
Answer:

Login and then issue the **pwd** command. It should display something similar to **/home/*YOUR_USER_NAME*** .

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.

Answer:

The exercise consists of a lot of **ls** (**lsf**) commands. Or, as an alternative, you could have used the **-R** (recursive) option. The directory map should look like



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?

Answer:

Full path name **/home/ *YOUR_USER_NAME* /tree/dog.breeds/retriever/labrador**

Relative path name **tree/dog.breeds/retriever/labrador**

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?

Answer:

```
$ cd
$ cd tree/dog.breeds/retriever
$ cd ../shepherd
$ cd ../../car.models
$ cd
```

To verify each destination

```
$ pwd
```

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?

Answer:

```
$ cd
$ mkdir junk
$ cd junk
$ pwd
/home/YOUR_USER_NAME
/junk
```

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

Answer:

```
$ mkdir junk/dirA junk/dirA/dir1 junk/dirA/dir2 junk/dirA/dir1/dirc
```

or

```
$ mkdir -p junk/dirA/dir1/dirc junk/dirA/dir2
```

If you entered the directory names in the order in which they are presented in the exercise, it will fail, because the command executes the arguments from left to right.

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?

Answer:

```
$ ls junk/dirA
$ ls /home/YOUR_USER_NAME/junk/dirA
```

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

Answer:

```
$ rmdir junk/dirA/dir1/dirc
$ rmdir junk/dirA/dir1
$ rmdir junk/dirA/dir2
$ rmdir junk/dirA

$ rmdir junk/dirA/dir1/dirc junk/dirA/dir1 junk/dirA/dir2 junk/dirA
```

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.

Answer:

```
$ cd
$ pwd
/home/YOUR_USER_NAME
$ ls -l /usr/bin/cp /usr/bin/vi            Absolute path names
$ ls -l ../../usr/bin/cp ../../usr/bin/vi
                                           Relative path names
```

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.

Overview of Module 5

Audience

general user — General system user

Product Family Type

open sys — Open systems environment

Abstract

This module is designed to teach students the uses of basic file manipulation commands such as **cp**, **mv**, **ln**, **cat**, **more**, and **rm**. The students must know how to log in/out and know some basic UNIX commands to navigate through the file system (**pwd**, **cd**, **ls** etc.).

Also the module will discuss the line printer spooler system that is available with most UNIX systems. During the course of this module, we will look at the terminology of the line printer system and commands that users can run to interact with the **lp** system. The management of the line printer spooler system is exclusively an administrative task; users need to know the **lp** spooler commands that will allow them to interact with the **lp** system.

Time

Lab — 30 minutes

Lecture — 45 minutes

Prerequisites

m46m — Navigating the File System

In order to successfully complete this module, the student must be able to log in and navigate the file system.

Instructor Profile

UX — General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000	Logon access to an HP-UX system

Software Requirements

UX11	HP-UX release 11.0

Material List

P/N B2355-90033(T)	*HP-UX Reference Manual* , one per terminal

Lab Instructions

setup1	Create user logon of **`user1, user2, ... user`** *n*, where *n* is the number of students in the class. Set up one user per student.
copyfiles	Copy the lab files to the users' home directories.

Lab Files

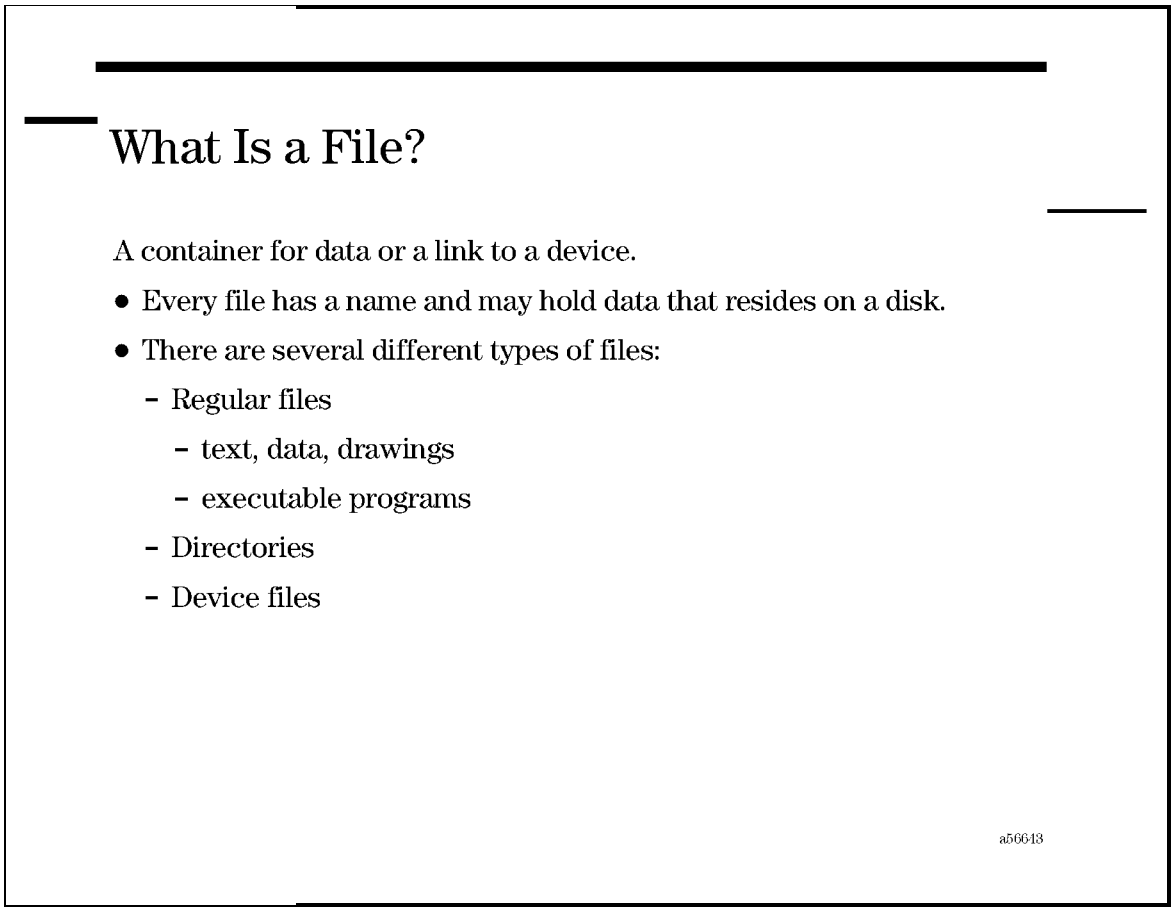
```
total 124
-rw-r--r-- 1 karenk users       17 May 28 16:12 apple
-rw-r--r-- 1 karenk users       17 May 28 16:12 banana
-rw-r----- 1 karenk users     3081 May 28 16:12 funfile
-rw-r--r-- 1 karenk users       16 May 28 16:12 grape
-rw-r--r-- 1 karenk users       16 May 28 16:12 lemon
-rw-r--r-- 1 karenk users       15 May 28 16:12 lime
-rw-r--r-- 1 karenk users       61 May 28 16:12 names
-rw-r--r-- 1 karenk users       18 May 28 16:12 orange
-rw-r--r-- 1 karenk users       16 May 28 16:12 peach
-rw-r--r-- 1 karenk users       61 May 28 16:12 scaveng.README
drwxr-xr-x 6 karenk users     1024 May 28 16:12 scavenger
drwxr-xr-x 4 karenk users     1024 May 28 16:12 tree

./scavenger:
total 8
drwxr-xr-x 5 karenk users     1024 May 28 16:12 east
drwxr-xr-x 5 karenk users     1024 May 28 16:12 north
drwxr-xr-x 5 karenk users     1024 May 28 16:12 south
drwxr-xr-x 5 karenk users     1024 May 28 16:12 west

./scavenger/east:
total 6
drwxr-xr-x 2 karenk users       24 May 28 16:12 1_mile
```

```
drwxr-xr-x 2 karenk users     1024 May 28 16:12 2_mile
drwxr-xr-x 2 karenk users       24 May 28 16:12 3_mile

./scavenger/east/1_mile:
total 0

./scavenger/east/2_mile:
total 2
-rw-r--r-- 1 karenk users       50 May 28 16:12 README

./scavenger/east/3_mile:
total 0

./scavenger/north:
total 6
drwxr-xr-x 2 karenk users     1024 May 28 16:12 1_mile
drwxr-xr-x 2 karenk users       24 May 28 16:12 2_mile
drwxr-xr-x 2 karenk users       24 May 28 16:12 3_mile

./scavenger/north/1_mile:
total 2
-rw-r--r-- 1 karenk users       16 May 28 16:12 README

./scavenger/north/2_mile:
total 0

./scavenger/north/3_mile:
total 0

./scavenger/south:
total 6
drwxr-xr-x 2 karenk users       24 May 28 16:12 1_mile
drwxr-xr-x 2 karenk users     1024 May 28 16:12 2_mile
drwxr-xr-x 2 karenk users     1024 May 28 16:12 3_mile

./scavenger/south/1_mile:
total 0

./scavenger/south/2_mile:
total 2
-rw-r--r-- 1 karenk users      46 May 28 16:12 README

./scavenger/south/3_mile:
total 2
-rw-r--r-- 1 karenk users      45 May 28 16:12 README

./scavenger/west:
total 6
drwxr-xr-x 2 karenk users    1024 May 28 16:12 1_mile
drwxr-xr-x 2 karenk users      24 May 28 16:12 2_mile
drwxr-xr-x 2 karenk users      24 May 28 16:12 3_mile

./scavenger/west/1_mile:
total 2
```

```
-rw-r--r-- 1 karenk users     604 May 28 16:12 README

./scavenger/west/2_mile:
total 0

./scavenger/west/3_mile:
total 0

./tree:
total 14
drwxr-xr-x 5 karenk users    1024 May 28 16:12 car.models
-rw-r--r-- 1 karenk users      17 May 28 16:12 cherry
-rw-r--r-- 1 karenk users      17 May 28 16:12 collie
drwxr-xr-x 4 karenk users    1024 May 28 16:12 dog.breeds
-rw-r--r-- 1 karenk users      17 May 28 16:12 poodle
-rw-r--r-- 1 karenk users      17 May 28 16:12 probe
-rw-r--r-- 1 karenk users      17 May 28 16:12 taurus

./tree/car.models:
total 6
drwxr-xr-x 2 karenk users      24 May 28 16:12 chrysler
drwxr-xr-x 4 karenk users    1024 May 28 16:12 ford
drwxr-xr-x 2 karenk users      24 May 28 16:12 gm

./tree/car.models/chrysler:
total 0./tree/car

.models/ford:
total 4
drwxr-xr-x 2 karenk users      24 May 28 16:12 sedan
drwxr-xr-x 2 karenk users    1024 May 28 16:12 sports

./tree/car.models/ford/sedan:
total 0./tree/car

.models/ford/sports:
total 2
-rw-r--r-- 1 karenk users      18 May 28 16:12 mustang

./tree/car.models/gm:
total 0./tree/dog

.breeds:
total 4
drwxr-xr-x 2 karenk users    1024 May 28 16:12 retriever
drwxr-xr-x 2 karenk users      24 May 28 16:12 shepherd

./tree/dog.breeds/retriever:
total 6
-rw-r--r-- 1 karenk users      27 May 28 16:12 golden
-rw-r--r-- 1 karenk users      29 May 28 16:12 labrador
-rw-r--r-- 1 karenk users      26 May 28 16:12 mixed

./tree/dog.breeds/shepherd:
```

```
total 0
```

5-1. SLIDE: What Is a File?



Student Notes

Everything in the UNIX system is a file, which 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 that contains 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.

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 a **shell script** 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: This 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 there will be a file for each hardware device with which your computer needs to communicate.

5-1. SLIDE: What Is a File? Instructor Notes

Key Points

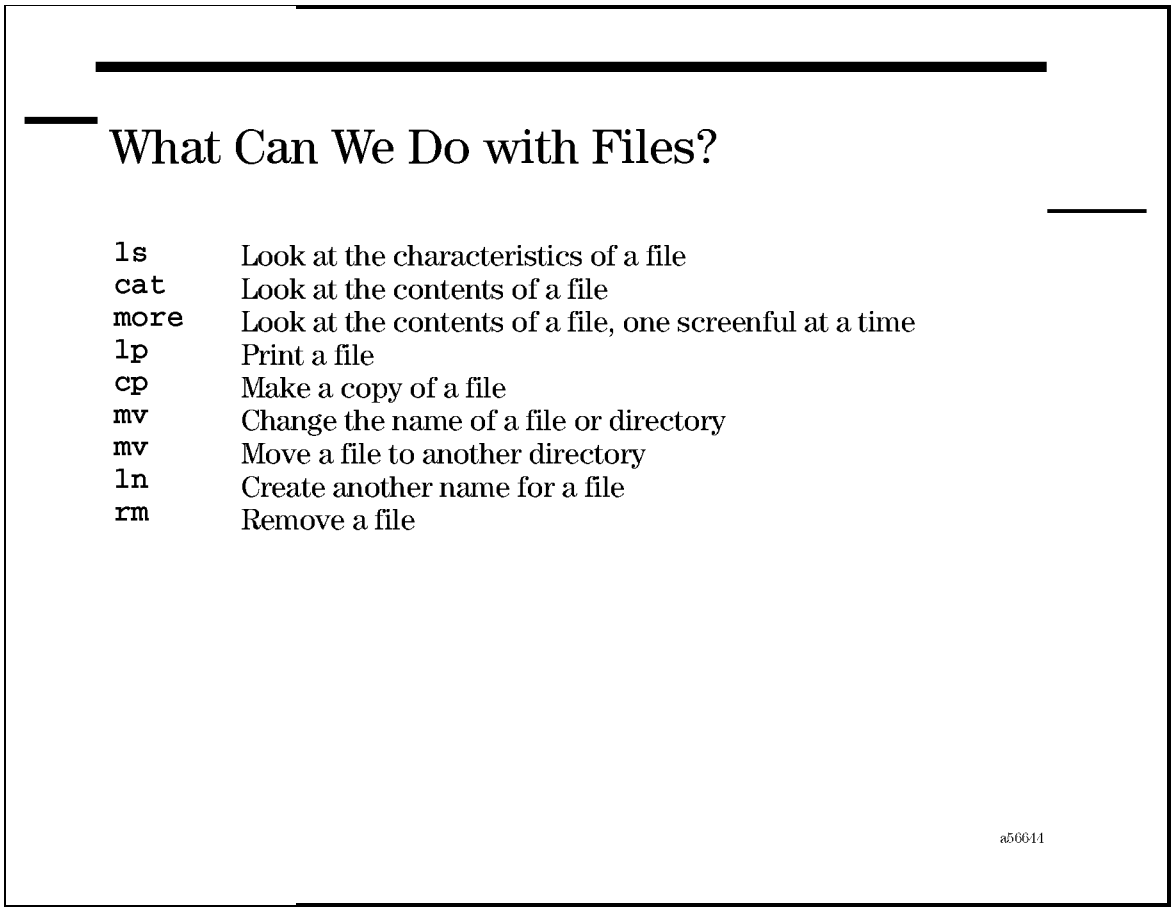
- Everything in the UNIX system is a file.
- A file is only a stream of bytes. The UNIX system imposes no format on the file structure.
- Distinguish between regular and special files.
- A directory is a special file that holds the file and subdirectory names and their associated inode numbers.
- Every device has an associated special file that provides the interface between the kernel and the hardware device.

Teaching Tips

If you have an advanced class, you might want to show them the **`ls -i filename`** and **`od -sc dirname`** commands to display the inode number for a file and the contents of a directory file. When examining the output of the **`od`** command, you should be able to find the inode number and file names for each entry in the directory.

Students should understand the basic concept of an inode in order to understand hard links, which will be presented later in this module.

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



Student Notes

Given that most activity on a UNIX system focuses around files and directories, there are many commands available to manipulate files and directories.

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

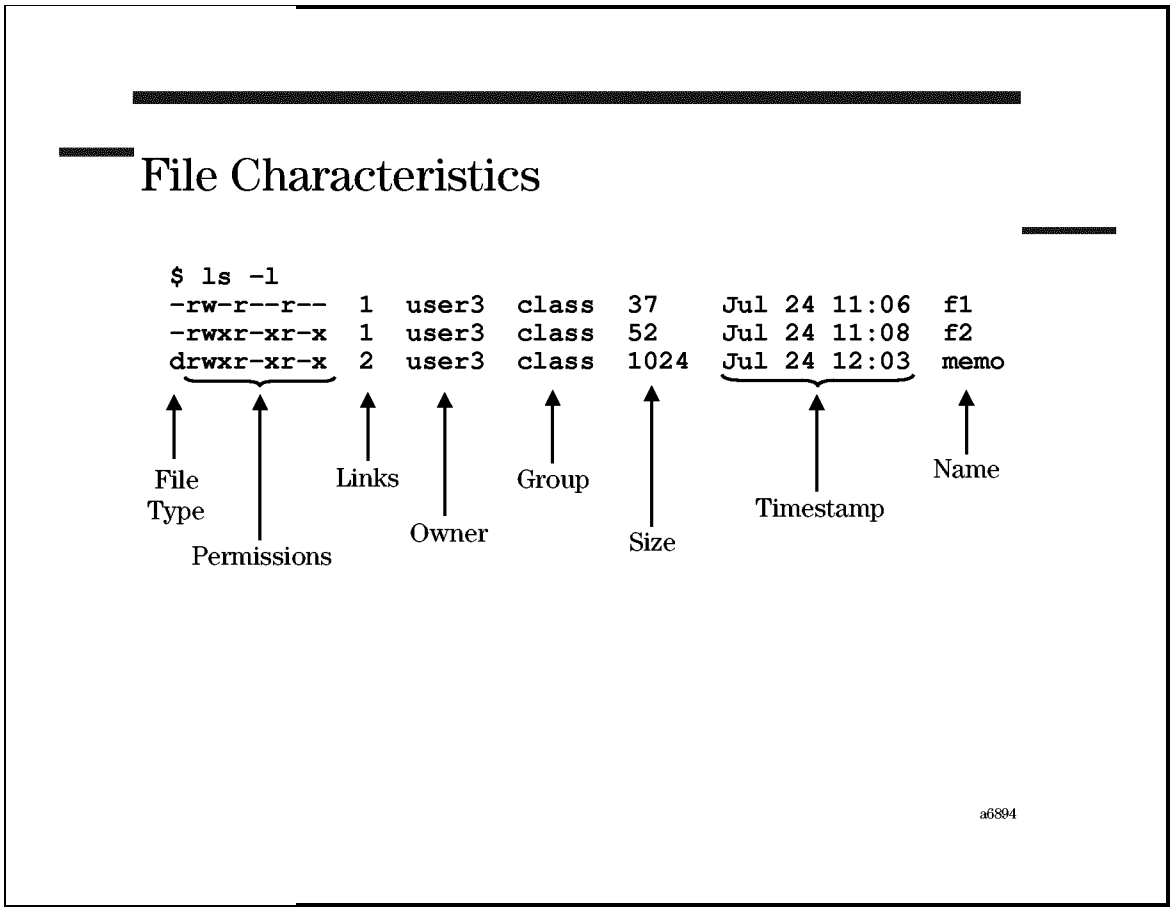
You will also need to create files and manipulate their contents. This is commonly done through the use of an editor such as **vi**.

5-2. SLIDE: What Can We Do with Files? Instructor Notes

Teaching Tips

Many students will ask about creating files and manipulating the contents of files. Point out that creating files is described in the module on **`vi`**. You may want to introduce the **`touch`** command that allows students to create empty files quickly.

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)

File Name Specifications

- maximum of 14 characters
- maximum of 255 characters if long file names are supported
- normally contain alpha characters (a–zA–Z), numeric (0–9), 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 just 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 only 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, and 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)

5-3. SLIDE: File Characteristics Instructor Notes

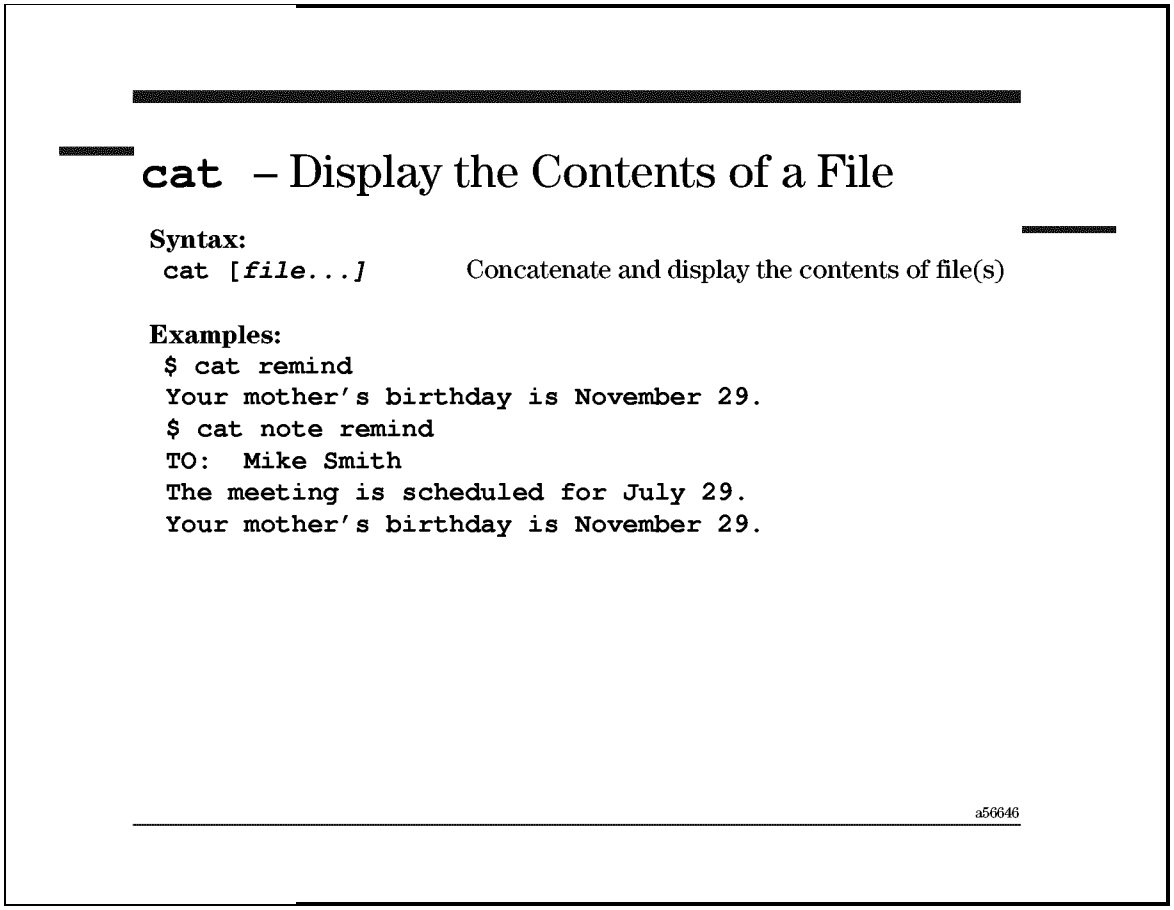
Key Points

- **`ls -l`** displays a file's characteristics.
- Briefly describe each field of the **`ls -l`** output.
- Discuss file name specifications.
- Point out that dot (**`.`**) is just a regular character, and can appear anywhere in the file name, and multiple times.
- Point out the use of the underscore (_) for file names representing two words.
- File name extension conventions may be defined by an application:

`.c`	source code for C programs
`.f`	source code for FORTRAN programs
`.p`	source code for Pascal programs
`a.out`	default compiled program name
`.dat`	data files

- When presenting the file types, you might want to give examples of character device files (terminals, printers) and block device files (disks).
- Mention the file types **`p, c, b`**, and **`n`**, but inform the class that we will not be discussing them in this class. You may mention that types **`b`** and **`c`** are covered in the system administration class, type **`p`** is covered in the system calls class, and type **`n`** is covered in the ARPA/Berkeley class.

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 to look at the contents of a single file. For example,

```
cat funfile
```

writes the contents 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 will be displayed to the screen.

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 by either of the following methods:

Method 1:

1. Try to log out—press Return and then issue the **exit** command.
2. Power cycle your terminal—turn it off, and then turn it 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 terminate your terminal session.

5-4. SLIDE: `cat` — Display the Contents of a File — Instructor Notes

Teaching Tips

- Have the students **`cat funfile`**.
- Point out that if **`cat`** has no files given, then it reads from the terminal and writes to the terminal. We will discuss **`cat`** as a filter later.
- You might want to mention **`cat -v filename`** to display the contents of a file that contains control characters. As an example: **`cat -v .exrc`**.

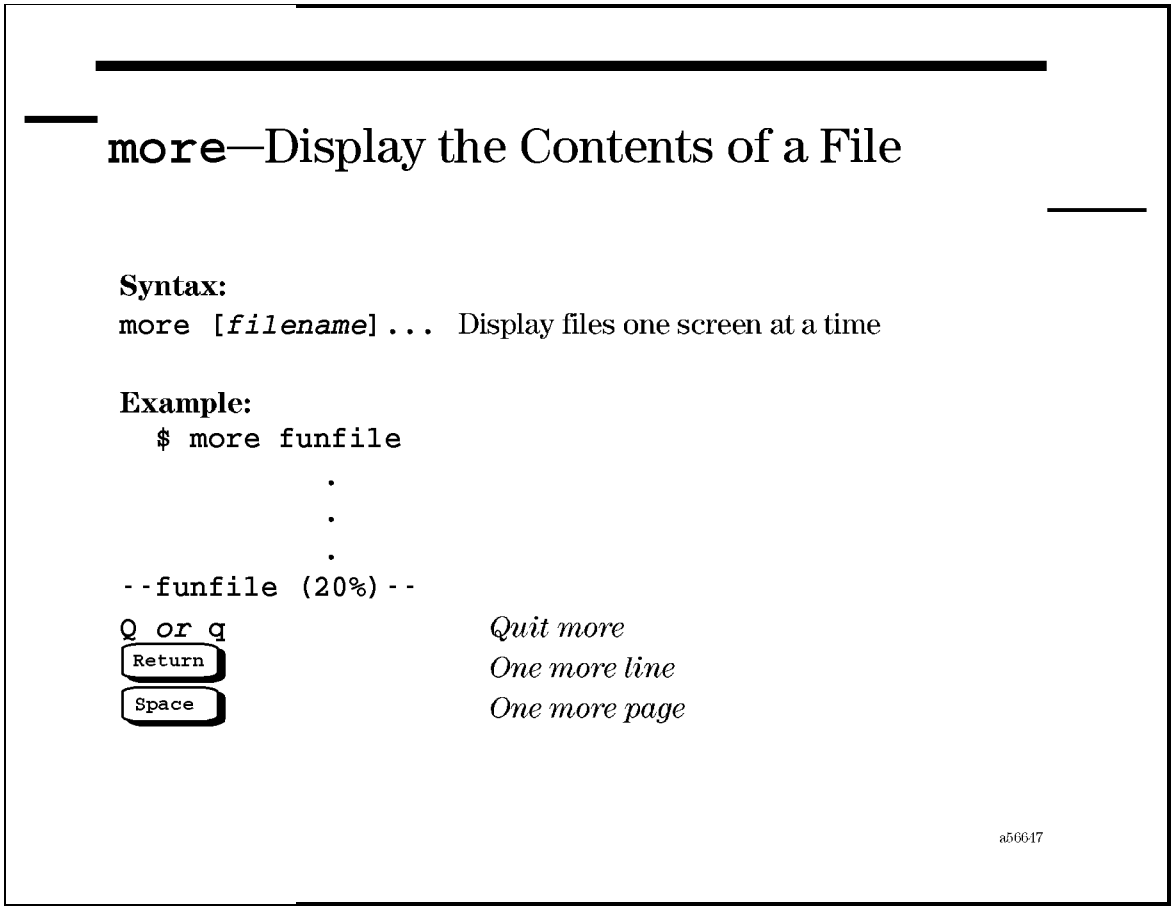
The `file` Command

You might want to present the **`file`** command to determine if a file is a text file or an executable file that could disrupt a user's terminal.

Example

```
$ file /usr/bin/ls
executable                    Do not display contents with cat

$ file /etc/passwd
text                          Can display contents with cat
```

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 only print 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.

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

Instructor Notes

Teaching Tips

- Have the students **more funfile** so that they can see the difference between **more** and **cat**.
- You might want to remind students of the Prev, Scroll + ↑ and Scroll + ↓ to scroll back and forth through their output.
- You might want to tell the students how to search for a text pattern while in **more** by using **/text_string** .
- You might want to mention the **pg** command.

Key Points

When using **more**, the students will have control over the output of their files to the screen. Also, **more** will provide headers when multiple files are being displayed with one **more** command, unlike **cat**.

Activity

You may wish to have the students use the following commands to illustrate the differences between the **cat** and **more** commands.

1. Use **cat** to see the contents of **funfile**.
2. Use **more** to see the contents of **funfile**.
3. With one command **cat** the contents of the files **remind** and **note**.
4. With one command **more** the contents of the files **remind** and **note**.

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

`tail`—Display the End of a File

Syntax:

`tail [-n] [filename]...` Display the end of file(s)

Example:

```
$ tail -1 note
soon as it is available.
```

a56648

Student Notes

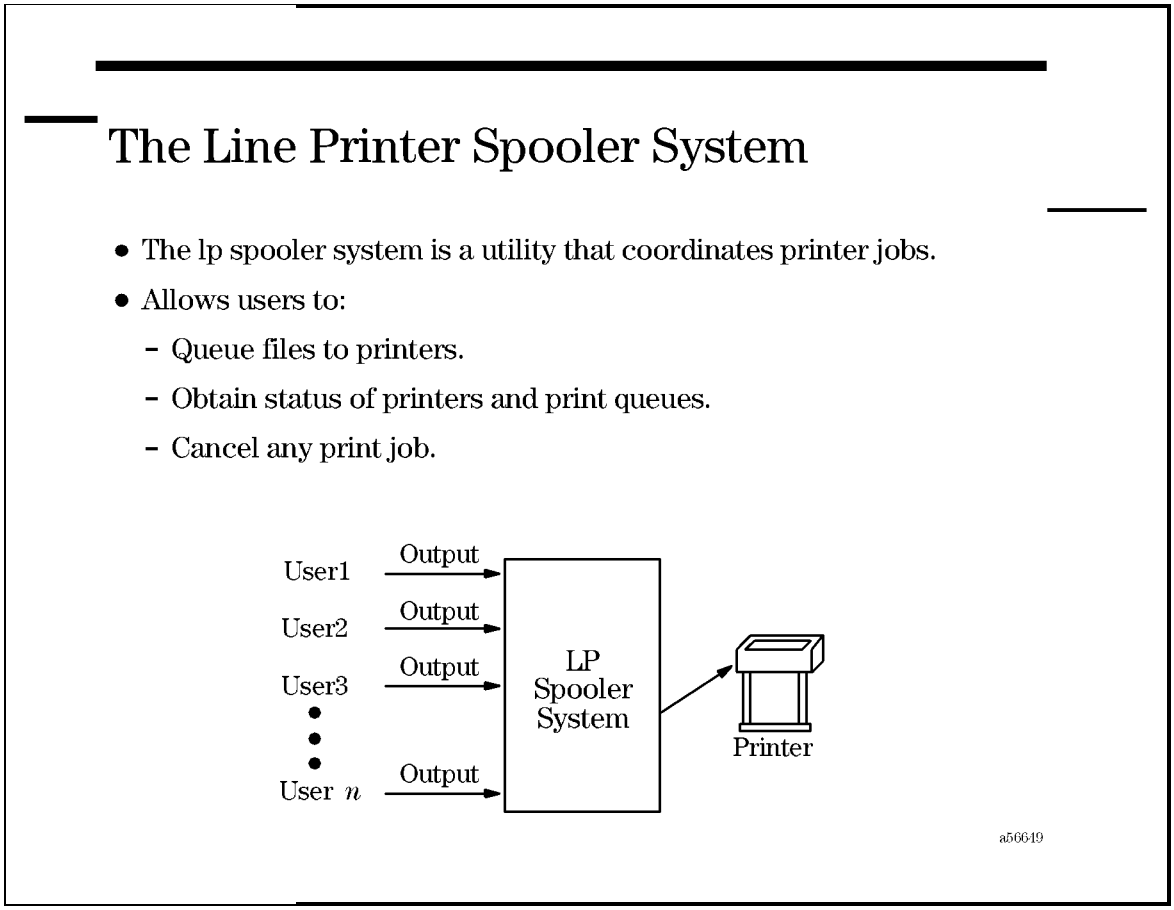
The **`tail`** command is useful for displaying the last *n* lines of a file. (Note: *n* defaults to 10 if it is not supplied.) This is especially useful for long log files that are periodically being appended to. With the **`tail`** command, you can go immediately to the last messages logged instead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

5-6. SLIDE: `tail` — Display the End of a File Instructor Notes

Teaching Tips

There is a companion command to **`tail`** known as **`head`**, which will display the first *n* lines of a file. If the count is omitted, it also defaults to 10 lines.

5-7. SLIDE: The Line Printer Spooler System



Student Notes

The UNIX operating system provides a utility called the **line printer spooler** (or `lp` spooler) that 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 and then appropriately configures the printer 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 wanted to print a file, he or she would have to make sure that no one else was 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.

5-7. SLIDE: The Line Printer Spooler System — Instructor Notes

Purpose

We start our discussion of the `lp` spooler system by presenting an overview of some of the things that the spooling system does.

Key Points

If we did not have a spooler to control print jobs, it would be up to the users to coordinate who uses the printers and when.

Users can submit requests, obtain information about the spooler, and cancel print requests.

Teaching Question

What are some of the things we would want to do through an `lp` spooling system?

A user can cancel any print request, even if it belongs to another user. Why would you want a user to be able to cancel another user's print requests?

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)
$
```

a56650

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 and it will queue the file to be printed on the default printer.

The **`lp`** command has a number of options available that allow you to customize the routing and printing of your jobs.

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 **lp(1)** 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."

5-8. SLIDE: The `lp` Command — Instructor Notes

Purpose

The **lp** command is one of the most frequently executed LP system commands.

Key Points

- The **lp** command allows the user to queue files for printing. Requests are known by their request ID. This ID is assigned to the request when it is submitted using **lp**.

Teaching Question

If there is more than one printer on the system, how can we specify which printer we want **lp** to send a job to?

Answer: Use the **-d***printer* option.

Teaching Tips

- If the students know pipelining, you can use a pipe to send the text to **lp**. For example,

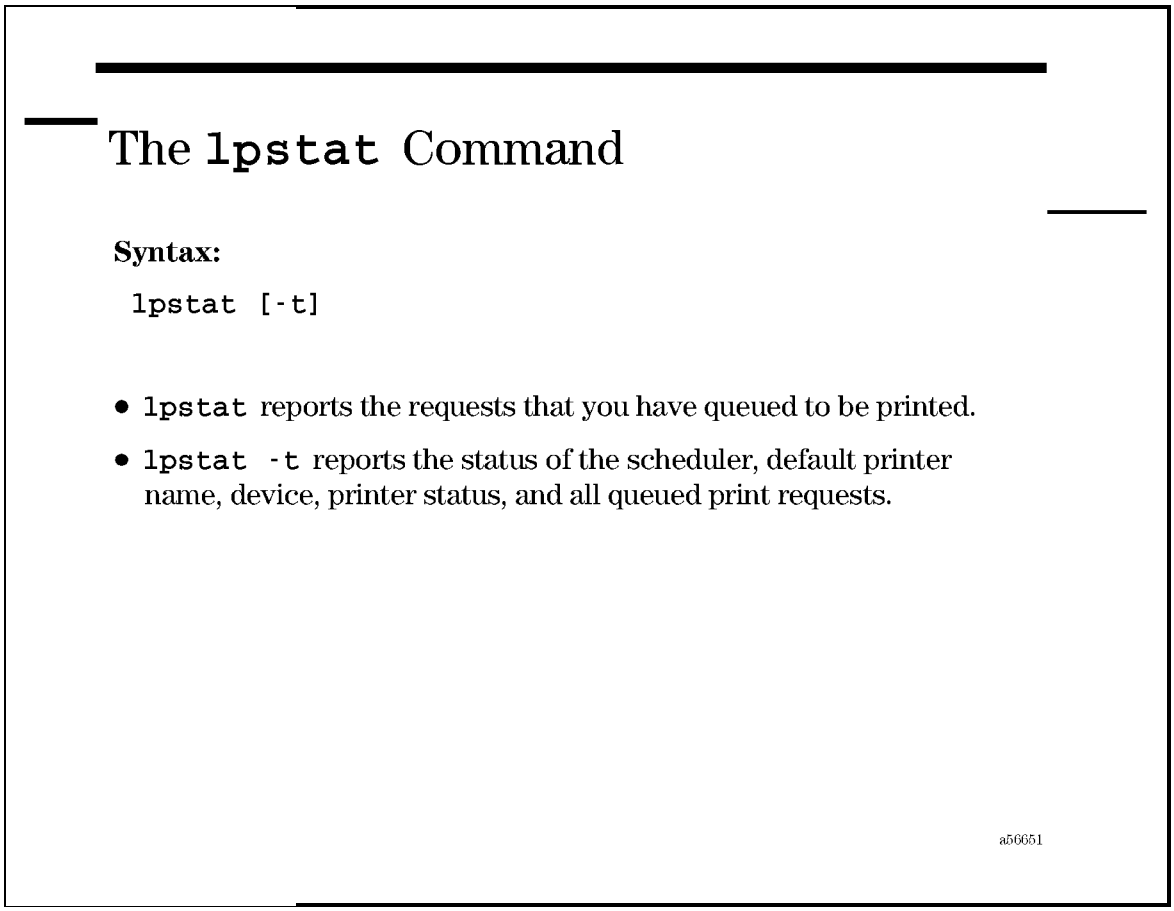
  ```
  $ ls -l | lp -dlaser -o12 -t"Directory Listing"
  request id is laser-114 (standard input)
  ```

  In this case, **lp** reports a request ID of **laser-114** because we specified that the output was to go to the printer named *laser* (**-dlaser**). **lp** also reported that the text has come from standard input. Notice also that the **-o12** argument sets 12 pitch and **-t"Directory Listing"** puts the title *Directory Listing* on the banner page.

- There are many options available to **lp**. It is not our intent to discuss all of them here. However, if students have questions about any of the options, an example of some can be drawn on the chalkboard. The **lp(1)** manual page clearly explains the available options.

  You may also want to cover common **-o** options such as **-onb** to suppress printing the banner page, **-o12** to change the print to 12 pitch, **-oraw** to print graphic dumps, and so on.

5-9. SLIDE: The `lpstat` Command



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.

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 then the date the request was made.

5-9. SLIDE: The `lpstat` Command — Instructor Notes

Purpose

The users should have a way of determining the status of any part of the **`lp`** spooler system. The **`lpstat`** command will report this information.

Key Points

- If a user submits a job to a printer and it does not print immediately, **`lpstat`** can be used to determine the position of the job on the print queue. If a printer is disabled, it will not print requests.
- The **`lpstat`** command reports the status of the various parts of the **`lp`** spooler system. It can be used to obtain information about the following:

  — the scheduler
  — the printers on the system
  — printer devices
  — the default system printer
  — spooler queues
  — queued requests

Teaching Questions

- Could we use **`lp`** to send a request to a printer that is disabled?
  Answer: Yes.
- What does **`lpstat`** report if you do not give it any options?
  Answer: It reports the requests that you have queued to be printed.

Teaching Tips

- Explain the purpose of the command and the types of information it will report on. Inform the students that the slide lists only one of the many **`lpstat`** options. For a complete list of options, they should refer to **`lpstat(1)`** in the *HP-UX Reference Manual*.
- As each of the options is discussed, draw a sample command line using the option on the chalkboard.

Transition

We will look at the **`cancel`** command next.

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`

a56652

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

By giving **`cancel`** a request ID, that specific print request will be canceled. 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
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
```

This command can be executed by any user 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.

5-10. SLIDE: The `cancel` Command — Instructor Notes

Purpose

Occasionally it may be necessary to remove a printing job from the queue or from the printer (if it is the job currently being printed). The **`cancel`** command will do this.

Key Points

- The **`cancel`** command is used to remove requests from a destination after the request has been made with **`lp`**.
- By canceling the current job on the printer, the next request will be printed.
- This command can be executed by any user on any request.
- Line printers cannot print the contents of object files. Nevertheless, some users will mistakenly submit an object file to be printed. The **`cancel`** command will remove the job from the queue, or, if it has started printing already (with undoubtedly poor results), it will stop the printing shortly.
- The **`cancel`** command can be used by any user to cancel any other user's job.

Teaching Questions

- Is there any way a specific job can be removed from a print queue, even if it is the job currently printing?
  Answer: Use the **`cancel`** command.
- Why would we ever need to cancel a printing request?
  Answer: You would not want to print a binary file.
- Why is it that any user can invoke **`cancel`**, even to cancel other users' jobs?
  Answer: Any user should be able to stop a binary file from being printed.
- Can the system administrator keep ordinary users from running **`cancel`**?
  Answer: Yes.

Where Problems Arise

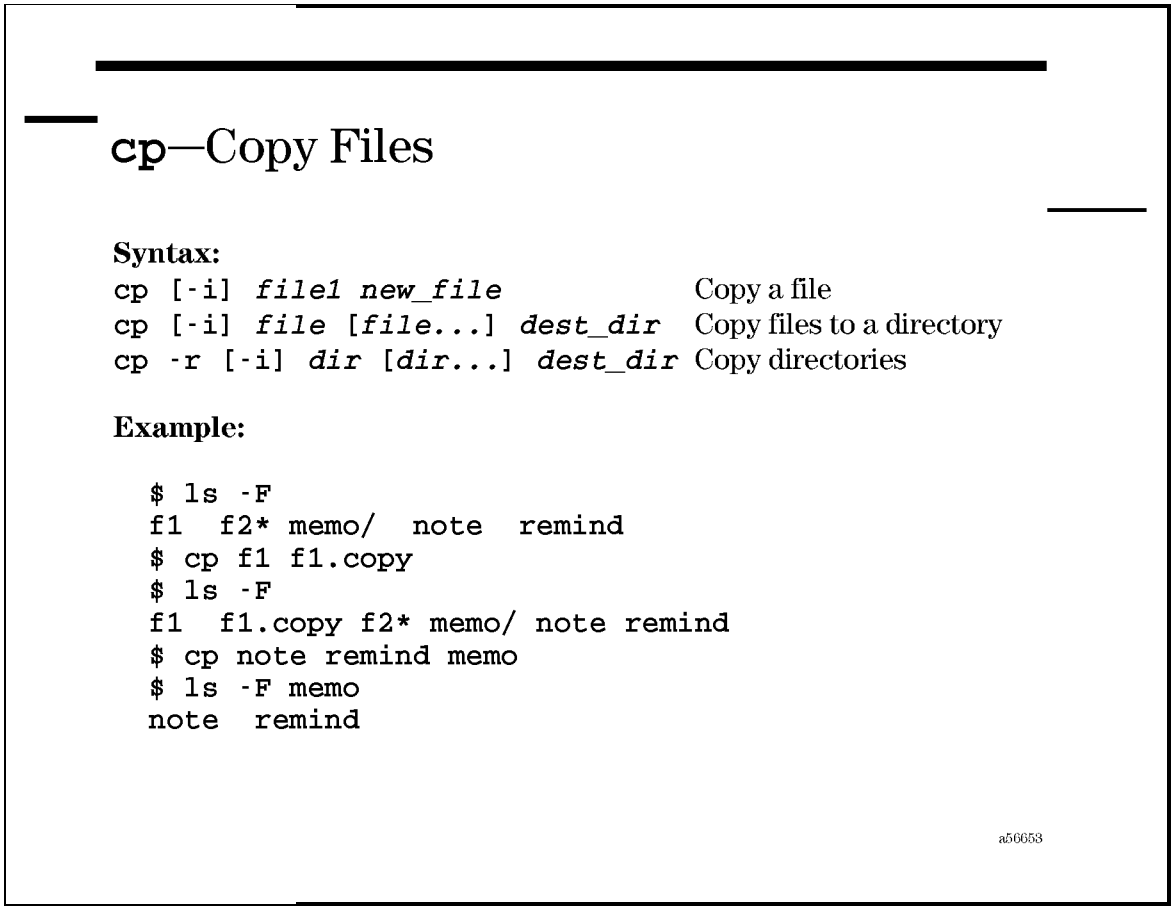
The use and purpose of the **`cancel`** command should not present any difficulty. Some students may question why a user can cancel another user's job. Give some examples of why you would want to cancel someone's request and it will become clear (for example, it is a nontext file, an extremely large file, and so on).

Teaching Tips

Explain the purpose of the command. Ask the students to provide examples of situations in which this command would be useful.

Differentiate between the different arguments that can be passed to **`cancel`**.

5-11. SLIDE: `cp` — Copy Files



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 and/or absolute path names can be provided 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***

- 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—no questions asked!

```
$ cp f1 note
$ cat f1
This is a sample file to be copied.
$ cat note
This is a sample file to be copied.
```

5-11. SLIDE: `cp` — Copy Files — Instructor Notes

Key Points

- **`cp`** requires at least two arguments.
- **`cp`** will copy over existing files—no questions asked.
- Relative or absolute path names can be used for any of the arguments.
- When copying a file to an existing file, **`cp`** does not change the existing file's access permissions, owner, or group designations.

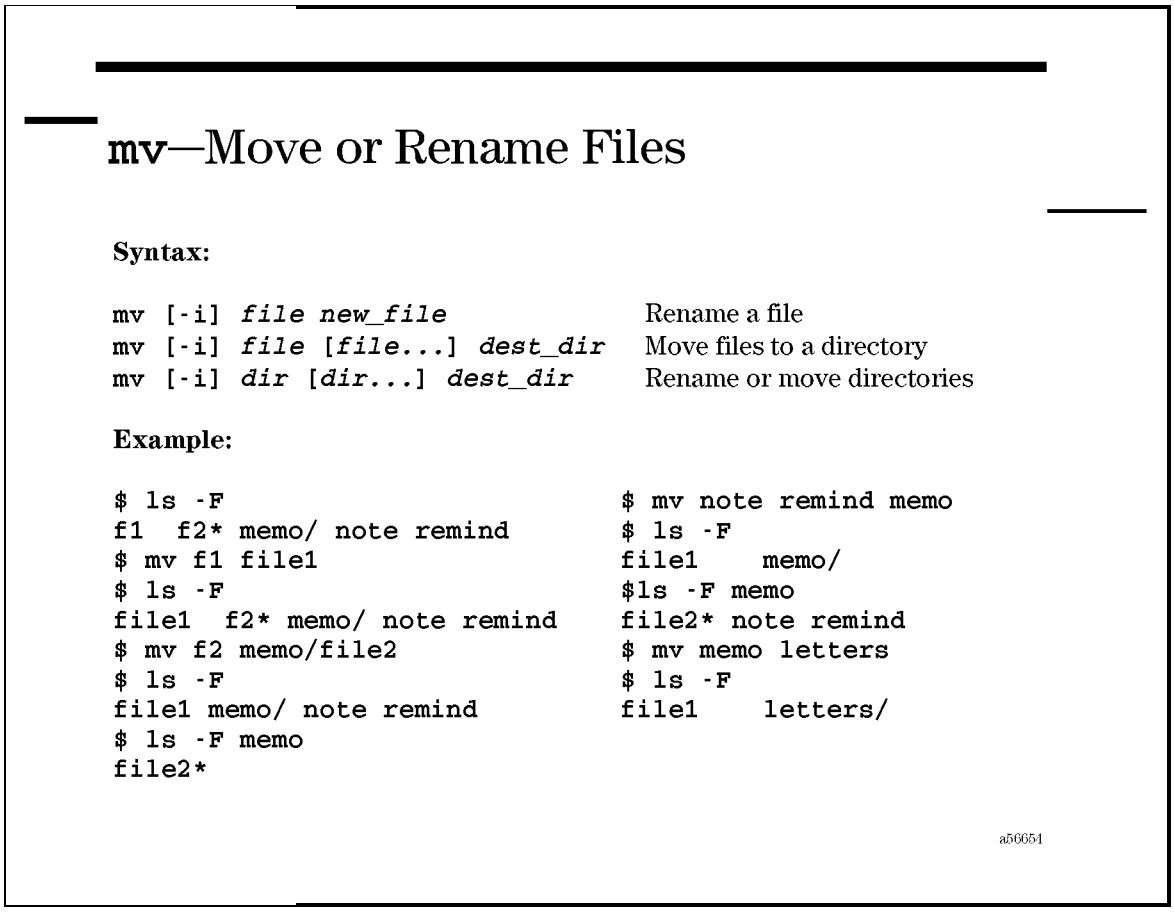
Teaching Tips

Sometimes it is helpful to draw a tree diagram on the board and add the files to the diagram as the **`cp`** commands are executed on the slide.

NOTE: HP-UX 10.0 and POSIX support a **`-i`** option, which will inquire if you want to copy over an existing file. The **`-i`** option will be ignored if the **`-f`** option is used.

NOTE: HP documentation prior to HP-UX 7.0 documented **`cp`** , **`mv`**, and **`ln`** under one man page—found under **`cp`**.

5-12. SLIDE: mv — Move or Rename Files



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 and/or absolute path names can be provided 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.
- 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 over existing files—no questions asked!

```
$ mv file1 note
$ cat file1
cat: cannot open file1
$ cat note
This is a sample file to be copied.
```

5-12. SLIDE: `mv` — Move or Rename Files Instructor Notes

Key Points

- **`mv`** requires at least two arguments.
- **`mv`** will move or rename over existing files—no questions asked.
- Relative or absolute path names can be used for any of the arguments.
- **`mv`** allows a directory as a source and destination.

Teaching Tips

Sometimes it is helpful to draw a tree diagram on the board, and rename and move the files in the diagram as the **`mv`** commands are executed on the slide.

NOTE: HP-UX 10.0 and POSIX support a **`-i`** option, which will inquire if you want to move or rename over an existing file.

5-13. SLIDE: `ln` — Link Files

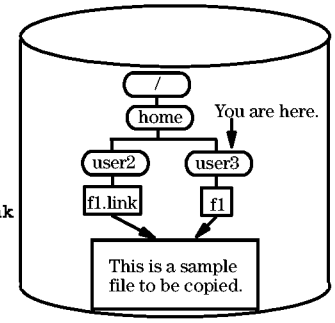
`ln`—Link Files

Syntax:

`ln` ***file new_file***	Link to a file
`ln` ***file*** `[`***file ...***`]` ***dest_dir***	Link files to a directory

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 f1   1789 /home/user2/f1.link
```



a65025

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 access to 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.

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
```

5-13. SLIDE: `ln` — Link Files — Instructor Notes

Key Points

- Links allow many names (aliases) for the same collection of data.
- The contents of a file are deleted ONLY when the link count reaches zero.
- The last person to save the contents of a file *wins*.

Teaching Tips

- Present hard links. You might want to add another hard link to the diagram on the slide in a directory associated with user1. Be sure to present the file sharing capability that links provide.
- You might want to mention the concept of an inode and that linked files share the same inode number. The inode stores all of the file characteristics, such as file type, permissions, number of links, ownership, size, time stamps, and addresses of the data on the disk. Inode numbers can be displayed with the command: **`ls -i`**
- Presenting the topic of symbolic links is optional. Symbolic links are important if you need to link directories or to link files across disks (or file systems). Figures are provided in case you wish to make slides to illustrate symbolic links.

NOTE: HP-UX 10.0 and POSIX support a **`-i`** option, which will inquire if you want to link over an existing file.

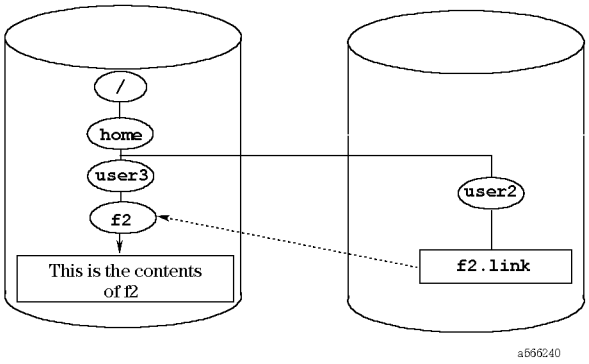


Figure 5-1. Symbolically Linked Files

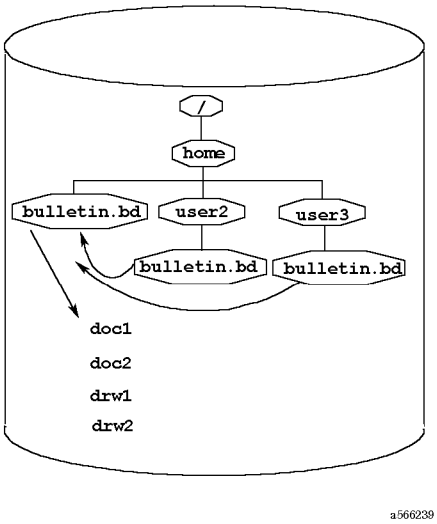


Figure 5-2. Symbolically Linked Directories

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? <user|y|
$ rm fruit
rm: fruit directory
$ rm -r fruit
```

a56656

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.

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**

5-14. SLIDE: `rm` — Remove Files — Instructor Notes

Teaching Tips

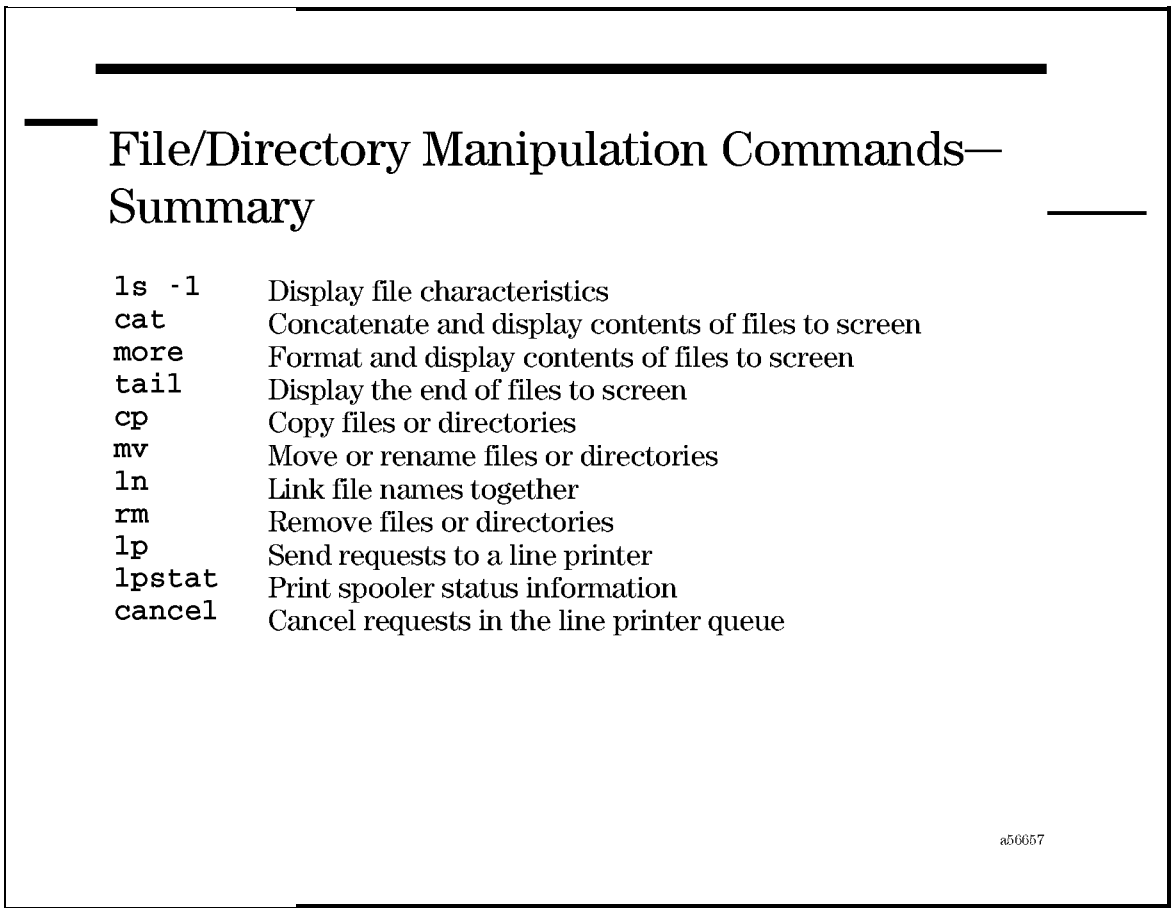
- Inform the class that once a file is removed, *it is gone!* Files can be restored only from a tape backup.
- When removing a directory interactively and recursively (**`rm -ri dirname`**), you will be prompted initially at the top of the directory. If you say *yes* to the parent directories of a file you later want to retain, all of the parent directories will be preserved, even though you may have said *yes* when initially asked if they should be removed.
- Links are removed just as any other file is removed with the **`rm`** command.

Teaching Question

What would be the result of an **`ls -F`** command after the final **`rm -r fruit`** command is executed?

Answer: Only the directory **`memo`** would remain.

5-15. SLIDE: File/Directory Manipulation Commands — Summary



Student Notes

5-15. SLIDE: File/Directory Manipulation Commands — Summary

Instructor Notes

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.

5. If the file **names** is modified, will this 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**?

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.

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 could 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 a directory **scavenger** and a file **scaveng.README** providing the first clue for a scavenger hunt. 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!

Printing Files

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 to be printed.

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.

5-16. LAB: File and Directory Manipulation Instructor Notes

Time: 30 minutes

Lab Objective

To practice using simple file manipulation commands and practice using commands to interact with the **lp** system.

Notes to the Instructor

Be sure to enable the system default printer before students begin the exercises.

Following are the minimum recommended exercises that students should complete:

File Manipulation: 1–7

Directory Manipulation: 1–4

Scavenger Hunt: NONE—Optional

Line Printer: 1–5

Even though the Scavenger Hunt is optional, students will have fun trying to find out what the code word is. It provides a good review of maneuvering through a branch of the file system.

The code word for the Scavenger Hunt is HERSHEY. You might want to have some Hershey's Chocolate Kisses to award the students who successfully discover the code word.

Solutions

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?

Answer:

```
$ cat funfile
```

The file is too long for one screen. The **more** command provides screen scrolling control. For example:

```
$ more funfile
```

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?

Answer:

```
$ more tree
****** tree is a directory ******
```

more knows that **tree** is a special directory file, not a normal text file, so its contents cannot be displayed to the screen in a readable format. You use the **ls** command to display the contents of a directory. For example:

```
$ ls tree
```

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?

Answer:

```
$ more /usr/bin/ls
****** /usr/bin/ls: Not a text file ******
```

more knows that **/usr/bin/ls** is a compiled program, not a normal text file, so its contents cannot be displayed to the screen in a readable format.

```
$ cat /usr/bin/ls
```

This command produces what appears to be garbage. In fact, this is what happens when you use the **cat** command to display a binary (compiled) program. Your terminal settings may have been changed by this. To reset your HP terminal:

- Hit the Break key.
- Simultaneously press Shift + Ctrl + Reset.
- Press Return to get the shell prompt.
- At the prompt, type the commands:

```
$ tset -e -k                          -e: sets erase to ^H, -k: sets kill to ^X
$ tabs
```

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.

Answer:

```
$ cp names names.cp
$ cat names names.cp
```

5. If the file **names** is modified, will this 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**?

Answer:

The files **names** and **names.cp** are individual entities. The content of **names** was overwritten with the content of the file **funfile**. The file **names.cp** is not affected.

```
$ cp funfile names
$ more names names.cp
```

names now contains the same contents as **funfile**, while **names.cp** still contains the content that was in **names**.

6. How do you restore the file **names** ? Issue the command to restore **names**.

Answer:

To restore the contents of the file **names**, copy or move from the file **names.cp**.

```
$ cp names.cp names
```

or

```
$ mv names.cp names
```

7. Make another copy of the file **names** called **names.new**. Change the name of **names.new** to **names.orig**.

Answer:

```
$ cp names names.new
$ mv names.new names.orig
```

8. How do you create two files (called **names.2nd** and **names.3rd**) that reference the contents of the file **names**?

Answer:

```
$ ln names names.2nd
$ ln names names.3rd      or     $ln names.2nd names.3rd
```

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**?

Answer:

The files **names**, **names.2nd**, and **names.3rd** are all referencing the same data on the disk. If one is modified, all three will be modified. From the long listing, you see that their link count has gone up to three, since there are now three names referencing the same data. **names.orig** is still an individual entity, as seen by its link count still being one.

```
$ cp funfile names
$ ls -l names.orig names names.2nd names.3rd
-rw-r--r-- 1 user3 class 37   Jul 24 11:06 names.orig
-rw-r--r-- 3 user3 class 125  Jul 24 11:08 names
```

```
-rw-r--r-- 3 user3 class 125  Jul 24 11:10 names.2nd
-rw-r--r-- 3 user3 class 125  Jul 24 11:12 names.3rd
```

If you do an **ls -i** of the **names** files, their **inode** numbers will be displayed. The inode stores each file's characteristics, such as permissions, number of links, and ownership. Files that are linked together share the same inode.

```
$ ls -i names.orig names names.2nd names.3rd
102 names.orig
322 names
322 names.2nd
322 names.3rd
```

10. Remove the file **names**. What happens to **names.2nd** and **names.3rd**?

Answer:

```
$ rm names
```

The files **names.2nd** and **names.3rd** are unaffected except that their link count will be reduced by one, which can be seen with the **ls -l** command:

```
$ ls -l names.orig names names.2nd names.3rd
names not found
-rw-r--r-- 1 user3 class 37   Jul 24 11:06 names.orig
-rw-r--r-- 2 user3 class 125  Jul 24 11:10 names.2nd
-rw-r--r-- 2 user3 class 125  Jul 24 11:12 names.3rd
```

11. Use the interactive option for **rm** to remove **names.2nd** and **names.3rd**.

Answer:

```
$ rm -i names.2nd names.3rd
names.2nd? y
names.3rd? y
$
```

12. Copy the file **names.orig** back to **names**.

Answer:

```
$ cp names.orig names
```

Solutions

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

Answer:

```
$ cd
$ mkdir fruit
$ mv lime grape orange fruit
```

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

Answer:

```
$ cd
$ mv apple fruit/APPLE
$ mv peach fruit/Peach
$
```

3. Look at the **tree** directory structure in your *HOME* directory. It requires a little organization.

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 could make these changes from any directory, but what directory do you think you should be in?

Answer:

```
$ cd
$ cd tree
$ pwd
/home/YOUR_USER_NAME/tree
$ mv collie poodle dog.breeds
$ mv probe car.models/ford/sports
$ mv taurus car.models/ford/sedan
$ mkdir horses
$ cp car.models/ford/sports/mustang horses
$ mv cherry ../fruit
```

4. Move the **fruit** directory from your *HOME* directory to the **tree** directory.

Answer:

```
$ cd
$ mv fruit tree
```

A directory called **fruit** is created under **tree**.

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.

Answer:

```
$ cd
$ cd tree/fruit
$ mv ../../banana ../../lemon .
```

Solutions

1. Under your *HOME* directory, you will find a directory **scavenger** and a file **scaveng.README** providing the first clue for a scavenger hunt. 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!

Answer:

```
$ cd
$ cd scavenger
$ more scaveng.README
north, 1 mile
$ cd north/1_mile
$ more README
east, 2 miles
$ cd ../../east/2_mile
$ more README
You are on the right track!
south, 3 miles
$ cd ../../south/3_mile
$ more README
You have to keep going
south, 2 miles
$ cd ../2_mile
$ more README
You are almost there
west 1 mile
$ cd ../../west/1_mile
$ more README
CONGRATS
You have found the end of the trail.
The code word is ________
```

Solutions

1. List the current status of the printers in the **lp** spooler system and find the name of the default printer.

Answer:

```
$ lpstat -t
scheduler is running
system default destination: rw
device for rw: /dev/rw
rw accepting requests since Jul 1 10:56:20 1994
printer rw is idle. enabled since Jul 4 14:32:52 1994
fence priority : 0
```

2. Send the file named **funfile** to the line printer. Make a note of the request ID that is displayed on your terminal.

Answer:

```
$ lp funfile
request id is rw-58 (1 file)
```

3. Verify that your requests are queued to be printed.

Answer:

```
$ lpstat
rw-58 ralph   3967  Jul 4 16:57:25 1994
rw-59 ralph   1331  Jul 6 13:01:19 1994
```

4. How can you tell what files other users are printing? Try it.

Answer:

You can tell by using **lpstat -t**.

5. Use the **cancel** command to remove your requests from the line printer system queue. Confirm that they were canceled.

Answer:

```
$ cancel rw-58 rw-59
request "rw-58" canceled
request "rw-59" canceled
$ lpstat
$
```

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 identity.

Overview of Module 6

Audience

general user — General system users

Product Family Type

open sys — Open systems environment

Abstract

This module is designed to teach the students about permissions on files and directories. We will also discuss the ownership and group access attributes. The associated commands—**chmod**, **chown** , and **chgrp**—are also discussed.

Time

Lab — 30 minutes

Lecture — 45 minutes

Prerequisites

m47m — Managing Files

In order to successfully complete this module, the student must be able to log in and navigate the file system.

Instructor Profile

UX — General UX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000 — Logon access to an HP-UX system

Software Requirements

UX11 — HP-UX release 11.0

Material List

P/N B2355-90033(T)	*HP-UX Reference Manual* , one per terminal

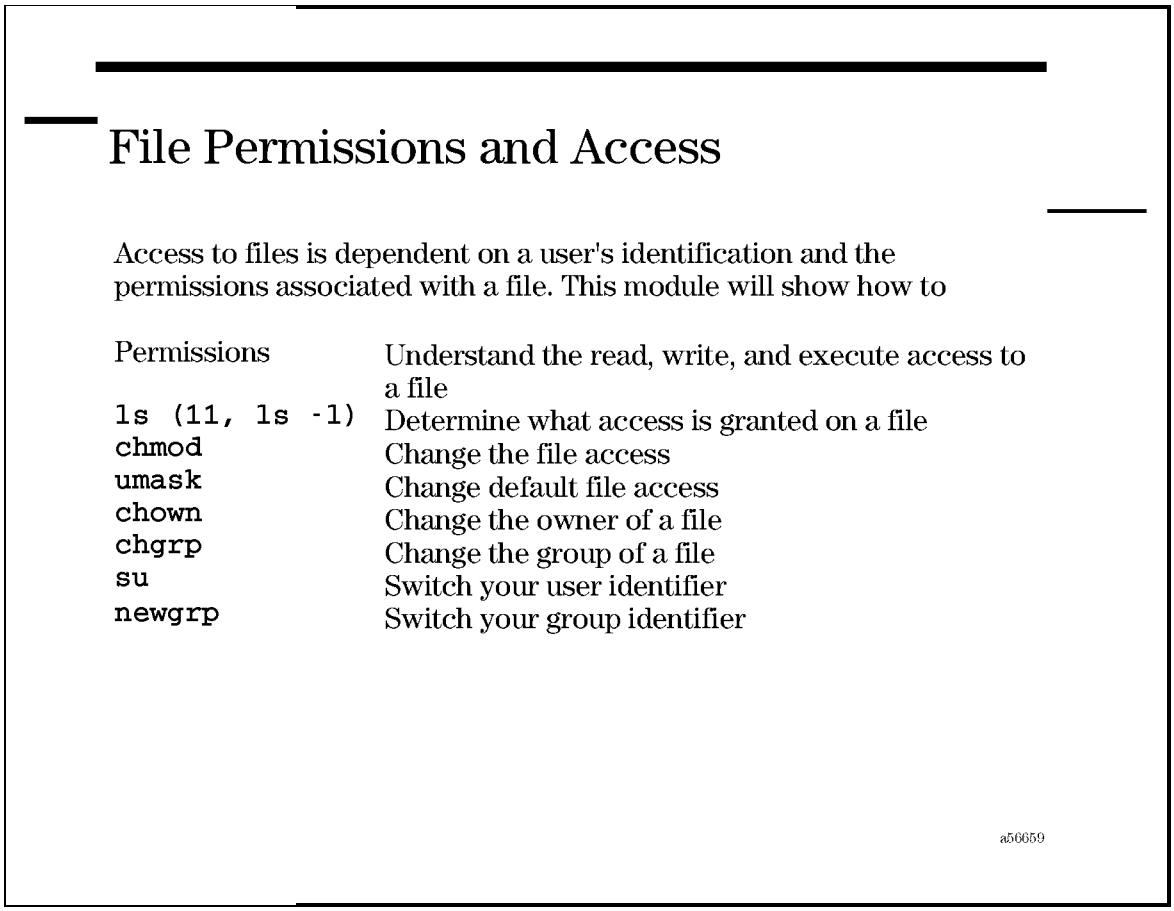
Lab Instructions

setup1	Create user logon of **user1**, **user2**, ... **user***n*, where *n* is the number of students in the class. Set up one user per student.
copyfiles	Copy the lab files to the users' home directories.

Lab Files

```
-rw-r--r--  1 karenk   users     20 May 28 16:12 logname
-rw-r--r--  1 karenk   users     20 May 28 16:12 mod5.1
-rw-r--r--  1 root     other     77 May 28 16:12 root_file
```

6-1. SLIDE: File Permissions and Access



Student Notes

Every file is owned by a user on the system. The owner of a file has the 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.

6-1. SLIDE: File Permissions and Access Instructor Notes

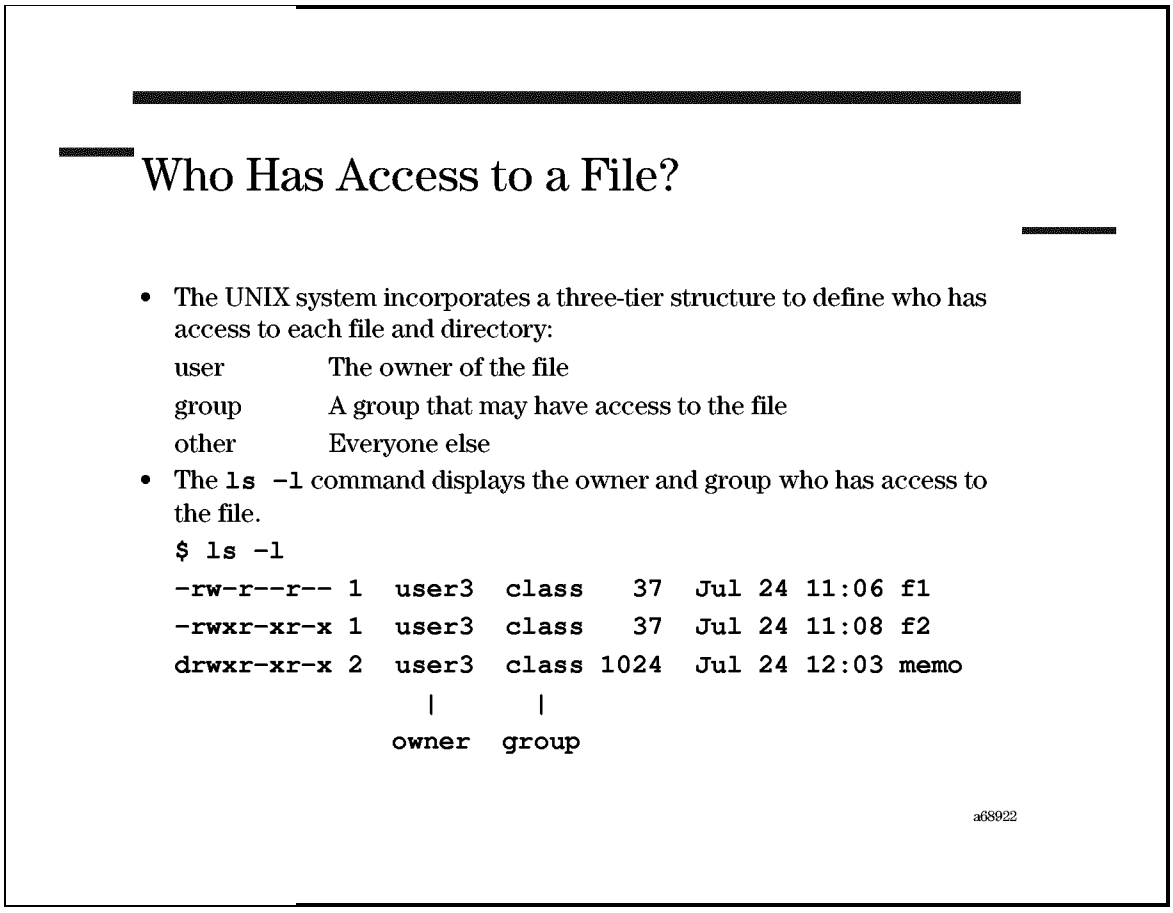
Teaching Tips

It is important that the students understand that the owner of a file has complete control over who has what access to the file. This module will show how to determine what type of access you have to various files and directories in the system.

In addition, the owner of the file can give up ownership or allow a different group access to the file.

Access Control Lists (ACLs) are included as a topic that you may or may not want to present. Many students find the standard UNIX system permission structure not specific enough, and ACLs may satisfy some of their access issues.

6-2. SLIDE: Who Has Access to a File?



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 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 will 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 will no longer have 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.

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.

6-2. SLIDE: Who Has Access to a File? Instructor Notes

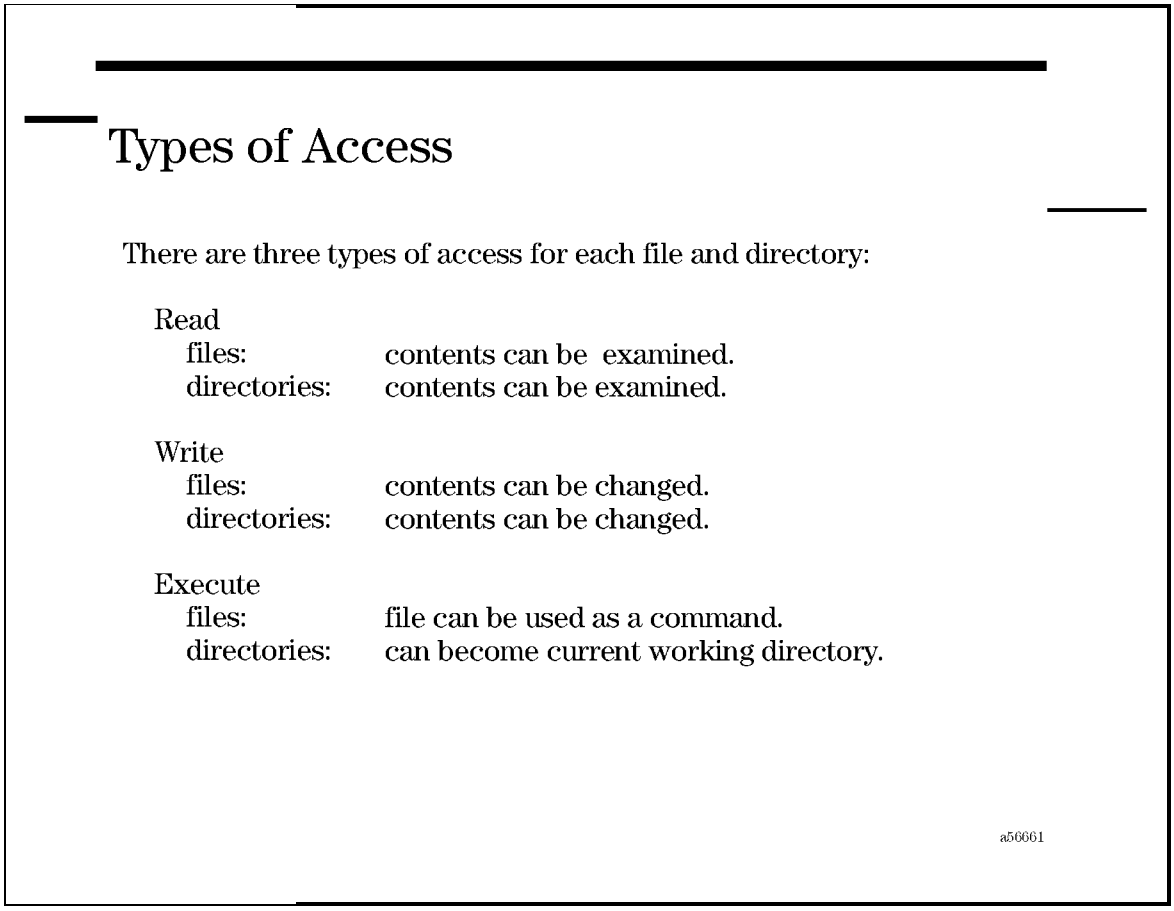
Teaching Tips

Owners and groups are introduced here as a precursor to permissions. Make sure that your students understand the three tiers: user, group, and other.

Key Points

- Every file will be owned by a user on the system.
- On the slide, the files are owned by *user3*.
- Every file will be associated with a specific group on the system.
- On the slide, the files are associated with the group *class* .
- The owner of a file may allow access to other users on the system.

6-3. SLIDE: Types of Access



Student Notes

There are three types of access available for each file and directory:

- read
- write
- execute

Different UNIX system commands will require certain permissions in order to access a program or file. For example, to **`cat`** a file it 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 can not be changed, and therefore files could not be deleted, added or renamed.

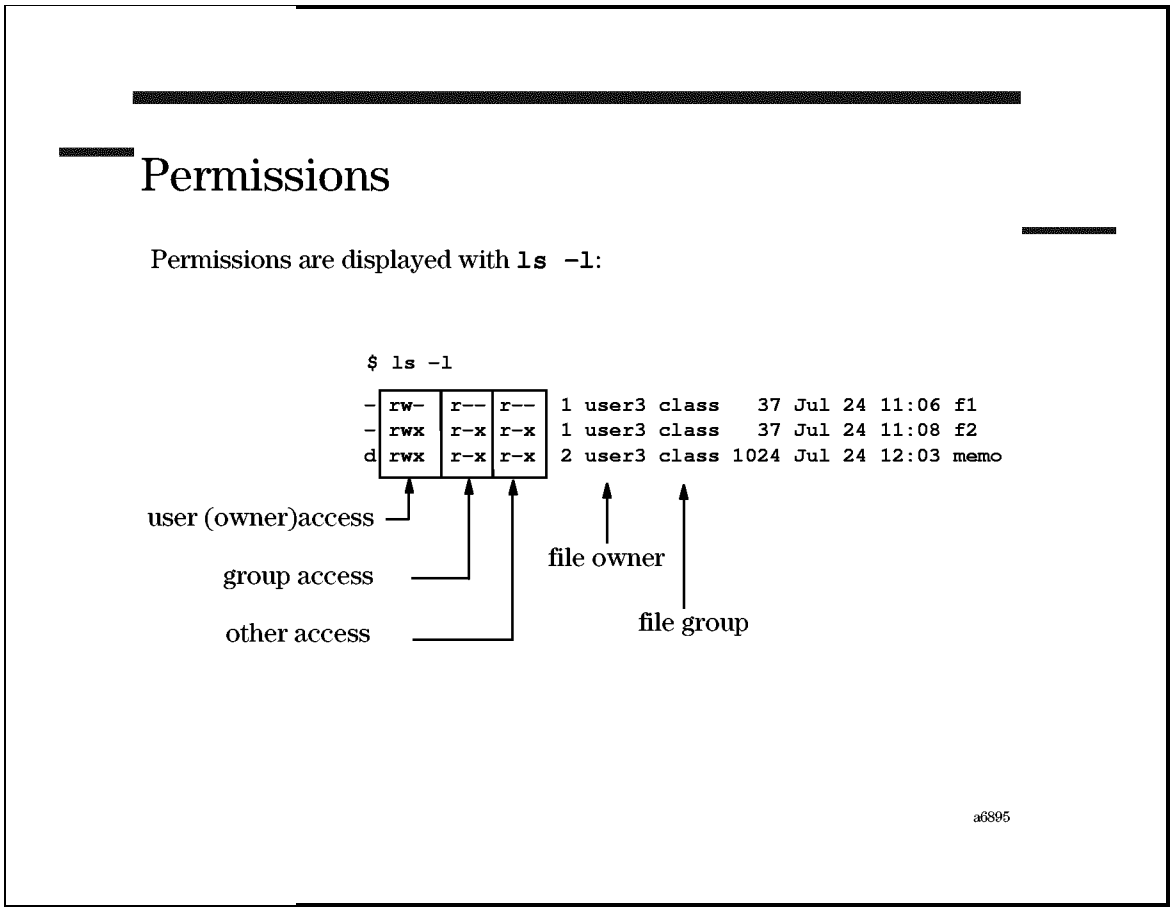
NOTE: In order to run a file as a program, both *read* and *execute* permissions are required.

6-3. SLIDE: Types of Access — Instructor Notes

Key Points

- There are three types of access: *read, write, execute.*
- Different commands require different access of target files or directories.
- Review the table on the slide and which commands require which accesses.
- File deletion is controlled at the directory level, not the file level.
- To protect a file from deletion, remove the *write* permission on the directory holding the file.
- Programs require both read and execute access.

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.

Examples

Referring to the files listed on the slide, access would be as follows:

Filename	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

6-4. SLIDE: Permissions Instructor Notes

Key Points

- Review the examples in the table in the student notes.

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
```

a56663

Student Notes

The **chmod** command is used to change the permissions of a file or directory. Permissions can *only* be changed by the file's owner (or *root*—the system administrator). 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. The write permission to a file would allow a user to change (or write over) the contents of the file, while write permission to a directory would allow 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

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's a commonly used form.

1. To change file permissions you have to convert each group of permissions into the appropriate numeric representation. There will be access 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. Just add together the 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            convert to numeric values
rw-     r--      ---
4+2+0   4+0+0    0+0+0
6       4        0
```

Thus the **chmod** command would be:

```
chmod 640 filename
```

NOTE: To disable all permissions on a file, issue the following command: **chmod 000 file**

6-5. SLIDE: `chmod` — Change Permissions of a File

Instructor Notes

Key Points

- File permissions are generally the *user's* responsibility, not the system manager's.
- The octal form of the mode option is now listed as obsolete. It is still supported, but symbolic modes should be used instead of octal modes.

Teaching Tips

- Review the example on the slide and the examples in the student notes.
- Make sure students understand how to derive the symbolic and octal representation.

Teaching Question

- What access do you have to a file when all of its permissions have been disabled?

  Answer
  You cannot read the contents of the file, you cannot change the contents of the file, you could not execute it if it was a program. You **CAN** delete the file. You can change the permissions to allocate yourself access.

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=
```

a68923

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, you should use the **`umask`** command. This will disable designated default permissions on any *new* file or directory that you create. Write access to the group and all others are probably the most important permissions to disable. The mask that you designate is active until you log out. **`umask`** will have no affect on existing files.

6-6. SLIDE: `umask` — Permission Mask Instructor Notes

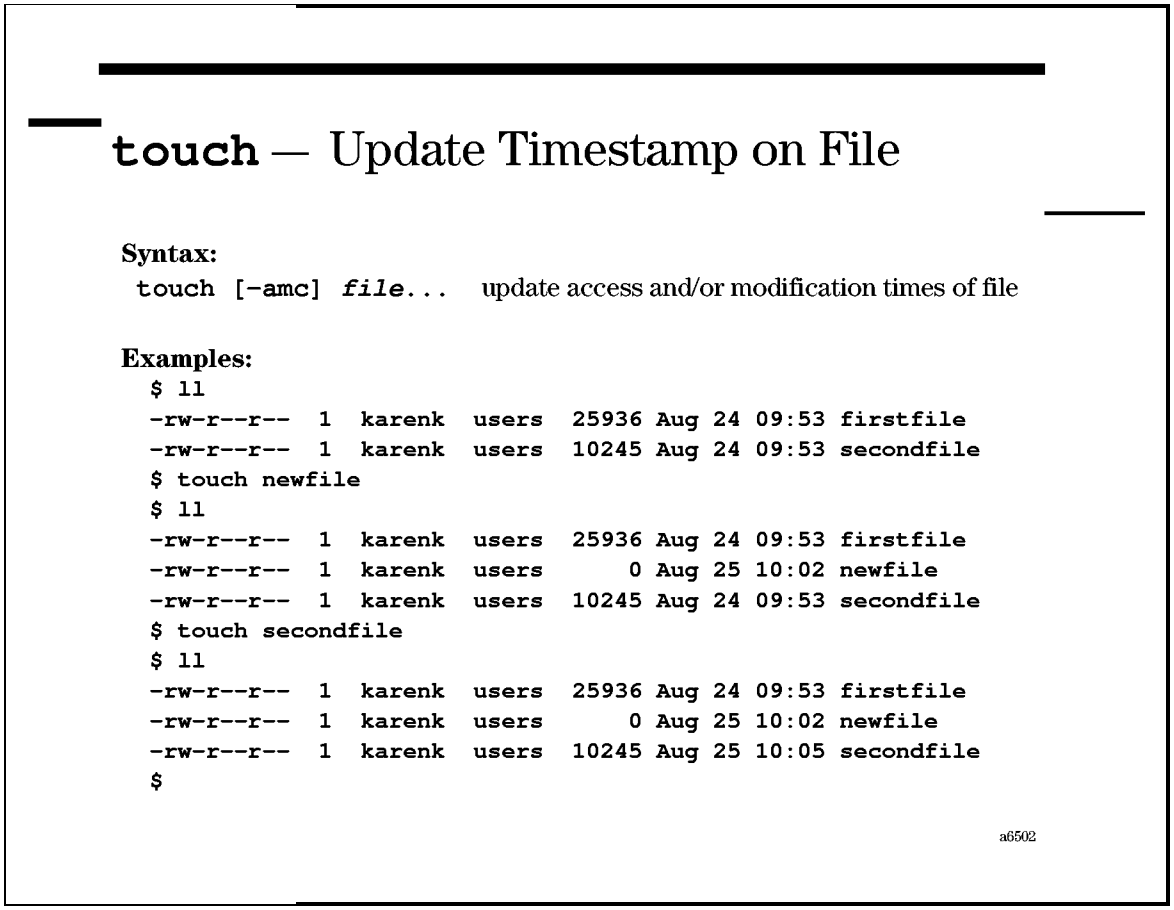
Key Points

- All files and directories are created with some default permission.
- Normally the default permissions give *everyone* access to your files and directories.
- The **`umask`** command can be used to disable some of these default permissions.
- **`umask 000`** will return the permission assignment to the system defined defaults.
- Students can set umask as part of their login process in **`.profile`**.

Teaching Tips

Review the examples that are presented in the student notes.

6-7. SLIDE: `touch` — Update Timestamp on File



Student Notes

The **touch** command allows you to create a new, empty file. If the designated file already exists, **touch** will just update the time stamp on the file. It will have 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.

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
```

6-7. SLIDE: `touch` — Update Timestamp on File — Instructor Notes

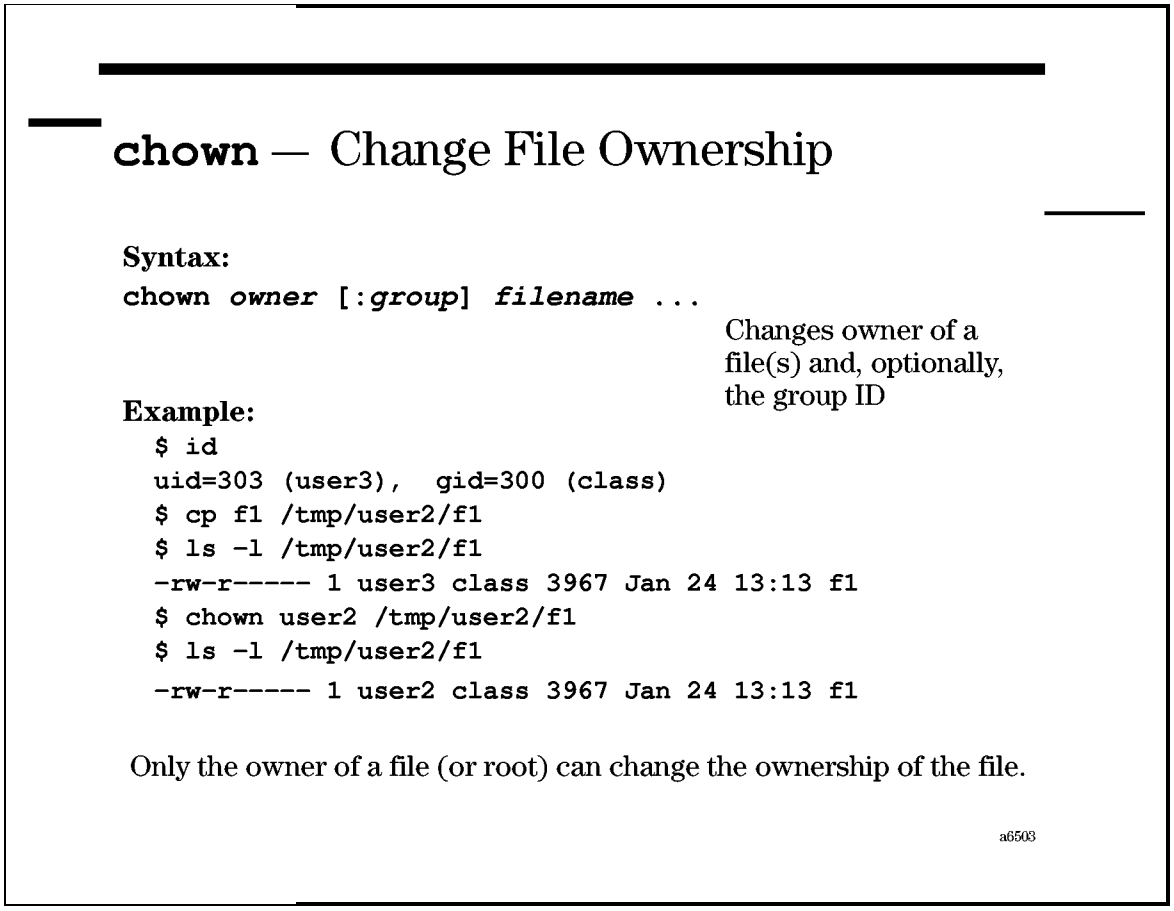
Key Points

- The **`touch`** command will create new files or update the time stamp on existing files.

Teaching Tips

- Have the students create an empty file using **`touch`**.
- Have students change the modify time on **`funfile`**.

6-8. SLIDE: chown — Change File Ownership



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, you 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.

Example

Looking at the example on the slide, after *user3* has transferred ownership of **`/tmp/user2/f1`** to *user2* , he will still have read access, since the file allows read access to all users who are a member of *class*.

6-8. SLIDE: `chown` — Change File Ownership Instructor Notes

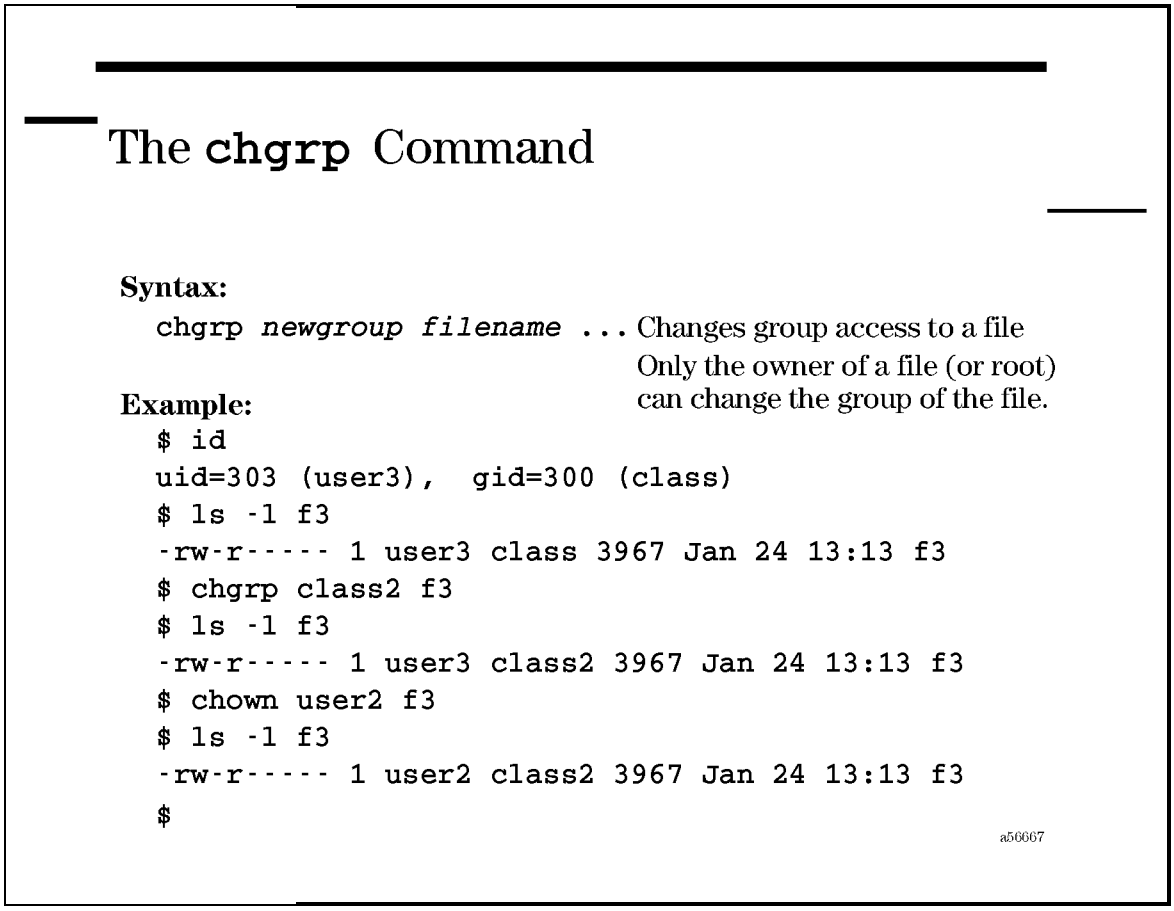
Teaching Tips

- Discuss what access *user3* has to the file **`/tmp/user2/f1`** before and after the **`chown`** command.
- Discuss the fact that the user can change owner *and* group simultaneously.

Key Points

- Only the owner of a file can run **`chmod`**, **`chown`**, and **`chgrp`**.
- One user can own a directory, and another can own a file within the directory.

6-9. SLIDE: The `chgrp` Command



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. Group existence and membership is controlled by the system administrator.

NOTE: Only the *owner* of a file (or *root*) can change the group identifier associated with a file.

Example

Looking at the example on the slide, after *user3* has transferred group access of the file **`f1`** to the group *class2*, her access has not been affected since she still owns the file. After *user3* gives the ownership of the file to *user2*, she will not be able to access it at all, since *user3* is currently associated with the group *class*.

6-9. SLIDE: The `chgrp` Command — Instructor Notes

Key Points

- Only the owner of the file can designate which group has access to a file.
- The owner (usually) does not control the membership in that group.
- You do not have to be a member of the group that you designate in the **`chgrp`** command.

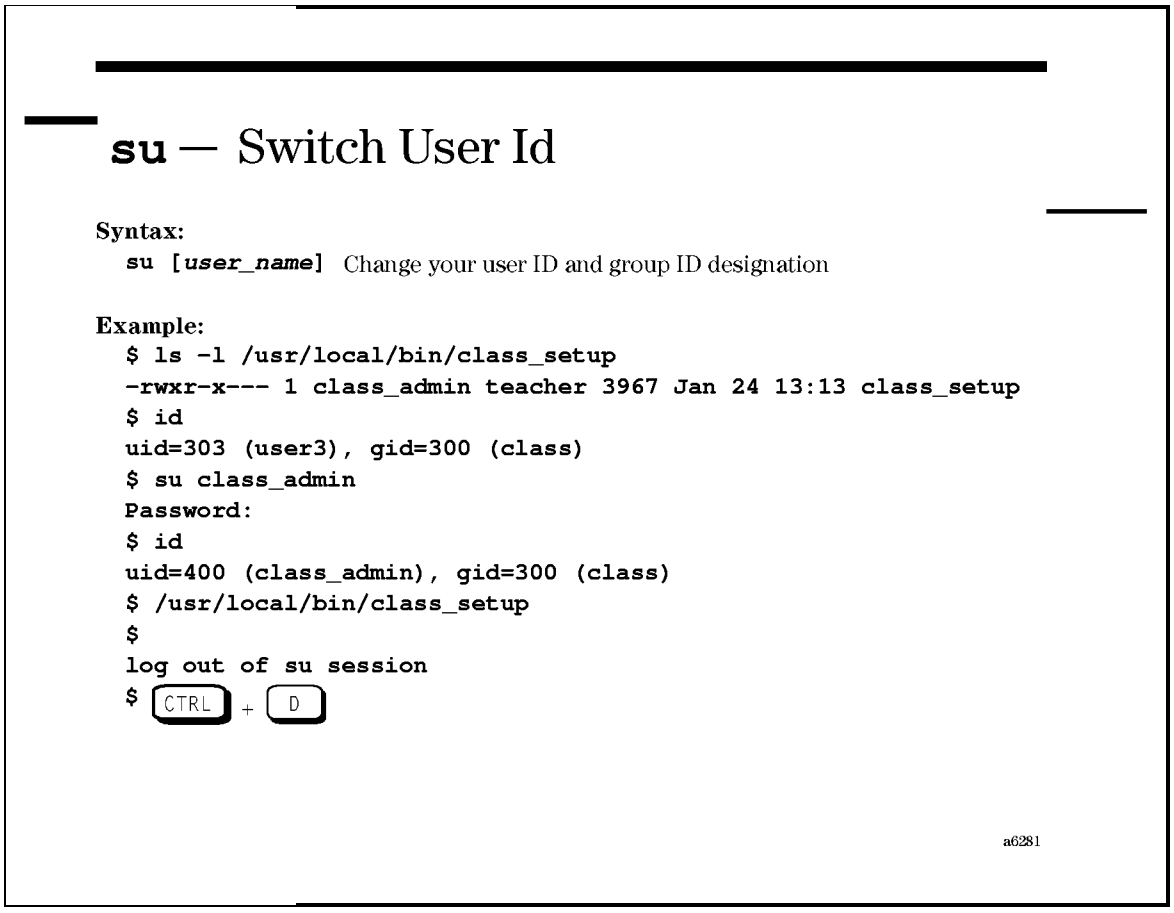
Teaching Tips

Review the example illustrated on the slide. Present what access *user3* will have on file **`f3`** after the first **`chgrp`**, and then again after the second **`chgrp`**.

This example shows how a user can change the group and ownership of a file, such that she cannot access a file at all.

There is the ability to be a member of more than one group at time. The **`/etc/logingroup`** is a file that the system administrator must create. This file has the same form as **`/etc/group`**.

6-10. SLIDE: su — Switch User Id



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 will be 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 *super-user*, 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 *super-user*. 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.

Example

Look at the example on the slide. *user3* does not have access to the program **/usr/local/bin/class_setup** , since she is not a member of the group *teacher*. She can access this program, though, if she enters the command **su class_admin**. As *class_admin*, she can also modify the contents of the program **class_setup**. When she has finished running the program, she resumes her 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 will remain the same as your original login identification. If you would like your session to take on the characteristics associated with the switched user ID, use the dash (**-**) option with the **su** command: **su - username** .

6-10. SLIDE: `su` — Switch User Id — Instructor Notes

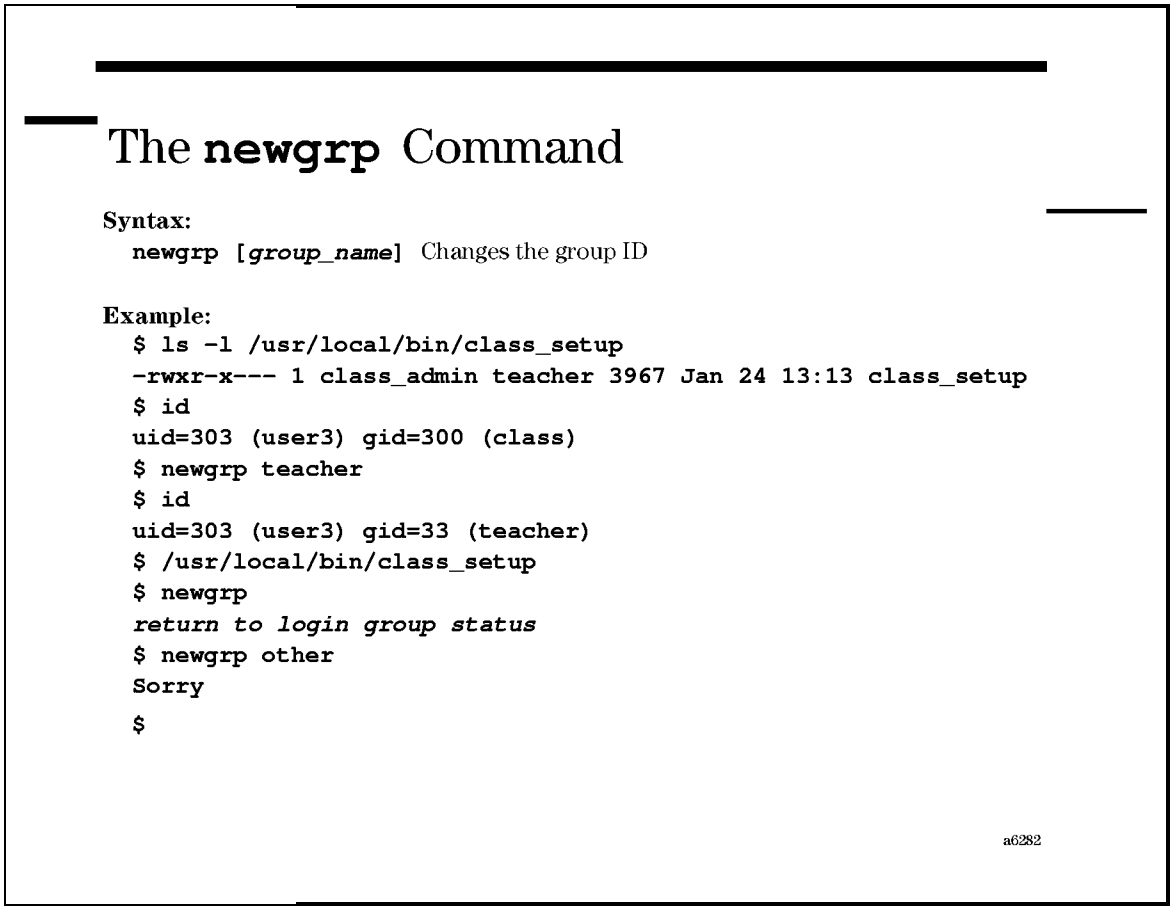
Key Points

- Most commonly used by the system administrator to **`su`** to root.
- Must provide the user ID's password.
- Best reason to have a password assigned to *your* account.
- To return to your previous user ID, a CTRL + d Return or an **`exit`** command will kill the child shell.

Teaching Question

- What would be the user's uid and group ID after she executed the commands on the slide? Answer: **`UID=303`** (user3), **`gid=300`** (class)

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 will define what 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 will get the message: **Sorry** .

Since the **newgrp** command does not start a new subsession, you just 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 he is initially a member of the group *class* . *user3* can **newgrp teacher** to the teacher group because he has been granted access to this group by the system administrator. Now he can run the program, since the program has execute access allowed for anyone in the *teacher* group, but he cannot modify the contents of the

program. Only the user *class_admin* can write to the **class_setup** program. When he has finished, *user3* will **newgrp** to resume his original group status.

Sample /etc/group file

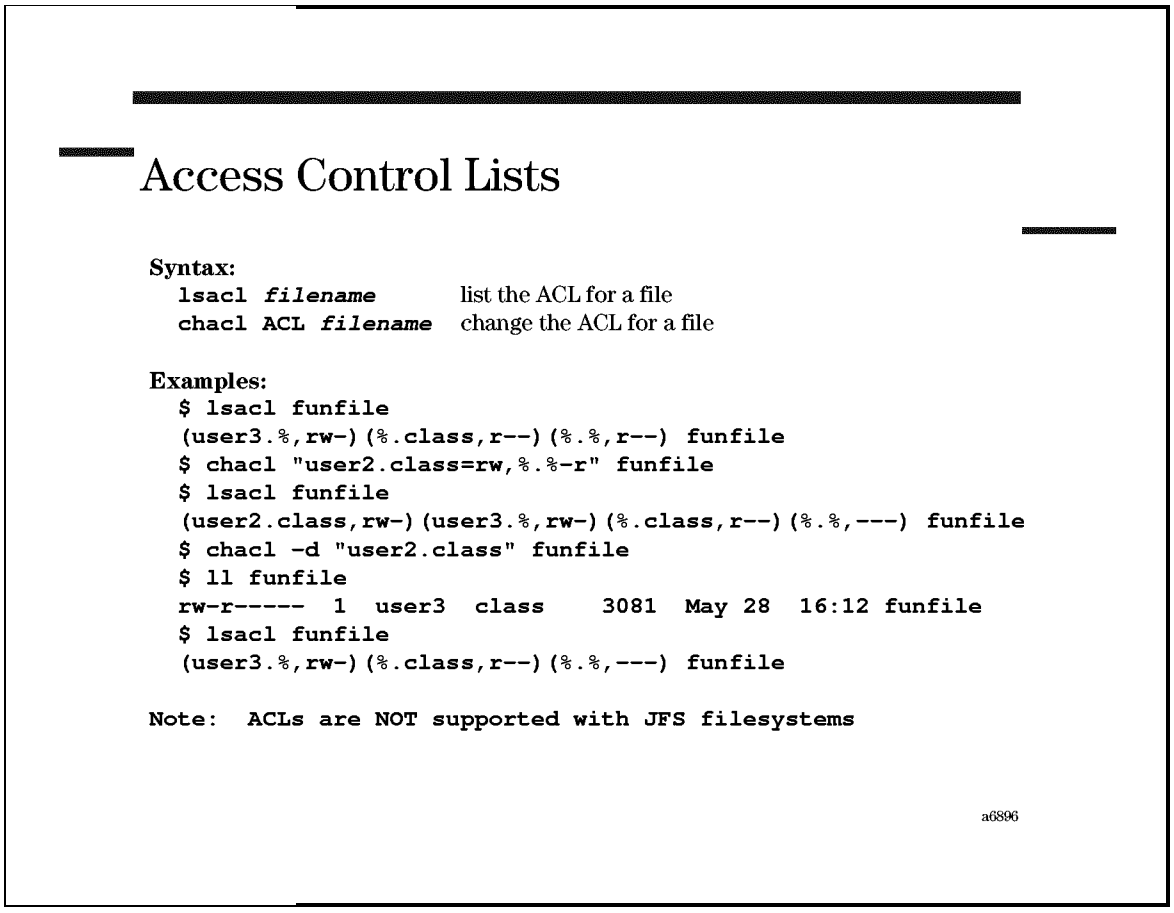
```
teacher::33:class_admin,user3

class::300:user1,user2,user3,user4,user5,user6,class_admin
```

6-11. SLIDE: The `newgrp` Command — Instructor Notes

Teaching Tips

Point out that **`newgrp`** must be used again to become a member of the user's original group.

6-12. SLIDE: Access Control Lists



Student Notes

Access control lists (ACLs) are an enforcement mechanism of discretionary access control (DAC), that allow 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 will take precedence over less specific designations.

Example

```
$ lsacl funfile
(user3.%,rw-)(user2.class,rw-)(user2.%,r--)(%.class,r--)(%.%,---) funfile
```

Users will have the following access to the file **funfile**

Table 6-1.

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 also been implemented 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 the permissions with **chmod** will remove all of the ACLs for that file.

NOTE: ACLs are only supported on **hfs** file systems, which are not the default at HP-UX 11.00.

6-12. SLIDE: Access Control Lists — Instructor Notes

Teaching Tips

Present the access to the file **`funfile`**. Present what access different users would have to the file **`funfile`** before and after the **`chacl`** command. Remember the more specific ACL will take precedence.

Before the **`chacl`** on the slide:

Table 6-2.

User Id	Group Id	Access to File
user3	any	read, write
any user other than user3	class	read
any user other than user3	any group other than class	read

After the first **`chacl`** on the slide:

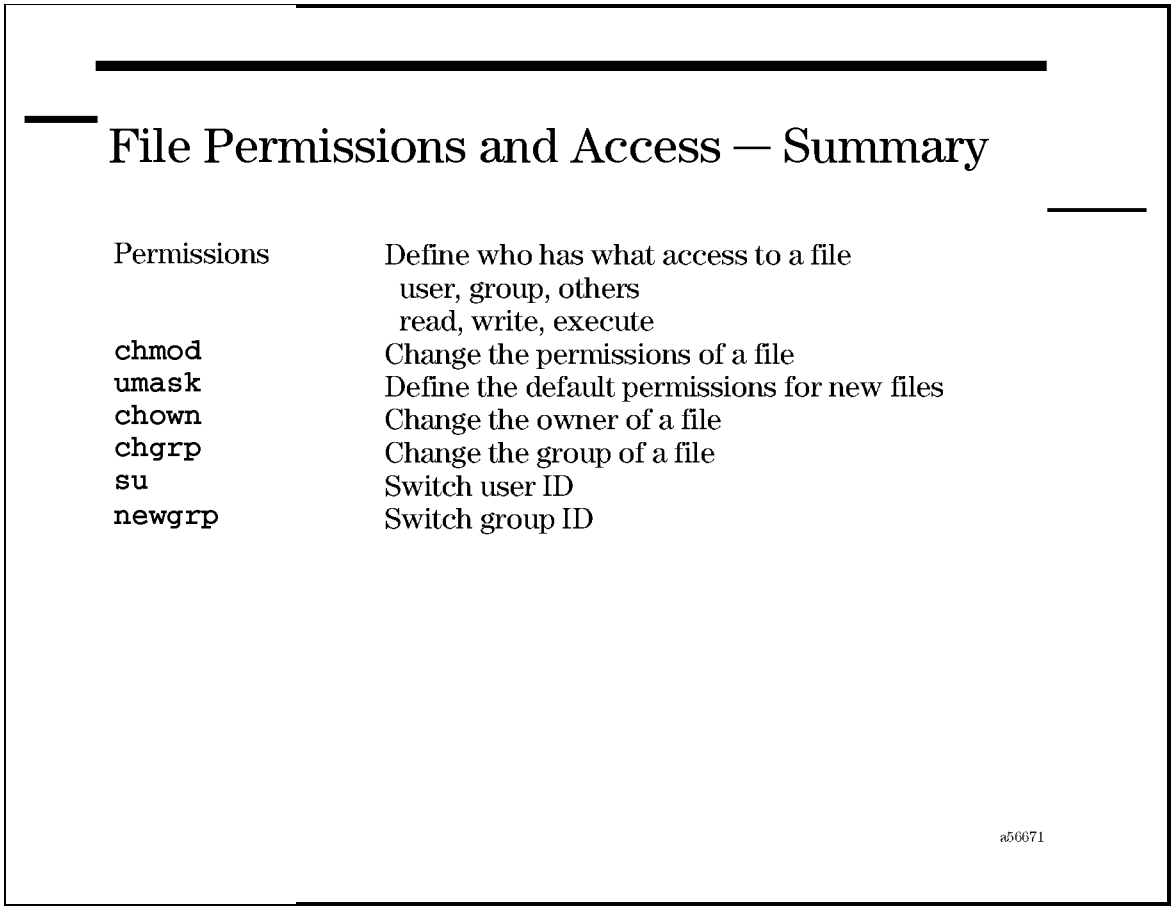
Table 6-3.

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

Other Notes

For customers requiring B1 level of security, Hewlett-Packard will be supporting additional security modules from Secure Ware. Be aware that the ACL implementation from Secure Ware is not compatible with the ACL implementation under HP-UX.

6-13. SLIDE: File Permissions and Access — Summary



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.

6-13. SLIDE: File Permissions and Access — Summary Instructor Notes

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 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 your partner display the contents of your **mod5.1**?

4. How can you modify the permissions on **mod5.1** so that your partner *can* read the file?

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 wanted to make changes to the file **root_file**, how would 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**)

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?

Changing Ownership and Group

1. Look under your *HOME* directory, you should find 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**?

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 permissions when you create a new file? Hint: Create a new file by using the editor, and copy or **touch** an existing file. 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.

6-14. LAB: File Permissions and Access — Instructor Notes

Time: 30 minutes

Purpose

To practice determining and modifying file ownership and permissions.

Notes to the Instructor

As part of the installation procedure, each of the users' *HOME* directory should have the permissions: **rwxr-xr-x** . There is also a file called **root_file** that is owned by root and group other. Students will only have read access to this file. All other files should be owned by the student, and have group class.

It is recommended that students complete the Basic Exercises and complete the Advanced Exercises only if they have time. Most students should complete the Basic Exercises.

Following is a breakdown of the exercises.

File Permissions:	Basic:1-6, Advanced:7-8
Directory Permissions:	Basic:1-4, Advanced:5-6
Ownership and Group:	Basic:1-3, Advanced:4-5
New Files:	Basic:1-2

Solutions

1. Look under your *HOME* directory for a file called **mod5.1**. Who has what access to this file? Can you display the contents of **mod5.1**?

Answer:

```
$ ls -l
-rw-r--r-- 1 YOUR_LOGNAME class     20 Jan 24 13:13 mod5.1
```

YOUR_LOGNAME has read and write access.
Members of group *class* have read access.
All other users have read.

```
$ cat mod5.1
```

This is successful since you have read permission.

2. Modify the permissions on **mod5.1** so that they are: **-w-------**. Can you display the contents of **mod5.1**?

Answer:

```
$ chmod a-rwx,u=w mod5.1
$ cat mod5.1
```

You no longer have read access to the file **mod5.1**, so the **cat** will fail.

3. Modify the permissions on **mod5.1** so that they are: **rw-------**. Can you display the contents of **mod5.1**? Can your partner display the contents of your **mod5.1**?

Answer:

```
$ chmod u=rw mod5.1
```

You can display the contents of **mod5.1**.
Your partner cannot display the contents of **mod5.1**.

4. How can you modify the permissions on **mod5.1** so that your partner *can* read the file?

Answer:

```
$ chmod g+r mod5.1
```

The file allows read access to all members of the group *class*, and your partner is also a member of the group class. Therefore, you provide the group read access to the file.

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?

Answer:

```
$ cp mod5.1 mod5.2
$ chmod -w mod5.2
$ rm mod5.2
mod5.2: 444 mode ? (y/n)
```

mod5.2 is removed!
You would have to remove the write permissions from your *HOME* directory as well.
If you remove write permissions from your *HOME* directory and then try to remove the file, you will get a message "permission denied".

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?

Answer:

The owner is root. The group is other. Only the super-user can change the ownership or group. You have read access only.

7. If you wanted to make changes to the file **root_file**, how would you go about it?

Answer:

Since you have read permission, you can make a copy of **root_file** . You will own the copy, and can therefore, modify the copy's contents.

```
$ cp root_file my_root_file
$ ls -l my_root_file
-rw-r--r-- 1 user3 class   3967 Jan 24 13:13 my_root_file
```

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?

Answer:

```
$ who am i
user3 tty03 Jul 9 11:10
$ mesg y
$ ll /dev/tty03
crw--w--w- 1 user3 class      0 Jan 24 13:13 /dev/tty03
$ mesg n
crw------- 1 user3 class      0 Jan 24 13:13 /dev/tty03
```

You own the device file associated with your terminal connection. The **mesg** command is essentially running a **chmod** on your terminal device file to grant or deny write access of others to your terminal.

Solutions

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**)

Answer:

```
$ cd
$ mkdir mod5.dir
$ cp mod5.1 mod5.dir
$ ls mod5.dir
mod5.1
$ ls -ld mod5.dir
drwxrwxrwx 3 YOUR_LOGNAME class   1024 Jul 24 13:13 mod5.dir
$
```

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**?

Answer:

```
$ chmod a-rwx,u+rw mod5.dir
$ cd mod5.dir
sh: mod5.dir: Permission denied.
$ ls mod5.dir
```

```
mod5.1
$ ls -l mod5.dir/
mod5.dir/mod5.1 not found
total 0
$ cat mod5.dir/mod5.1
cat: cannot open mod5.dir/mod5.1: Permission denied
$
```

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**?

Answer:

```
$ chmod u+wx mod5.dir
$ ls mod5.dir
mod5.dir unreadable
$ cat mod5.dir/mod5.1
This is the contents of mod5.1
$ cd mod5.dir                          cd is successful
$ pwd
/home/user3/mod5.dir
$ ls
. unreadable
```

4. Can other users copy files into your *HOME* directory? How do you display the permissions for your *HOME* directory?

Answer:

```
$ cd
$ ls -ld .
drwxr-xr-x 3 YOUR_USER_NAME class   1024 Jul 24 13:13 .
```

Other users can display the contents of your *HOME* directory, and change to your HOME directory, but they cannot modify the contents of your *HOME* directory. Therefore, other users cannot copy files to 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** ?

Answer:

```
$ ls -ld /usr/bin
dr-xr-xr-x 3 bin other    1024 Jul 24 13:13 /usr/bin
```

Write access for others is not set on **/usr/bin**, so your copy should fail.

6. Can you copy the file **/usr/bin/date** to your *HOME* directory?

Answer:

```
$ cd
$ ls -l /usr/bin/date
-r-xr-xr-x 1 bin bin   16384 Nov 15 13:13 /usr/bin/date
$ cp /usr/bin/date .
```

Since **/usr/bin/date** has read permission for others, you are able to make a copy of the file.

Solutions

1. Look under your *HOME* directory, you should find 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?

Answer:

```
$ ls -l YOUR_LOGNAME
-rw------- 1 YOUR_LOGNAME class   3967 Jan 24 13:13 YOUR_LOGNAME
```

You have read and write access.
Your partner is also in the group *class*.
Your partner has no access 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?

Answer:

```
$ ls -l YOUR_LOGNAME
-rw------- 1 YOUR_LOGNAME class   3967 Jan 24 13:13 YOUR_LOGNAME
You initially have read and write access.
$ chown partner_login_name YOUR_LOGNAME
$ ls -l YOUR_LOGNAME
-rw------- 1 partner class 3967 Jan 24 13:13 YOUR_LOGNAME
```

- You no longer have access to this file.
- You need a minimum of read access to copy a file, so you cannot make a copy with the current permissions.
- You are a member of the group *class*, but the group permissions are disabled.

The only way you can get ownership back, is to have your partner **chown** the file back to you. (Have you been nice to your partner?) You could also **su** to your partner's account (if he or she will share his or her password) and **chown** the file yourself.

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?

Answer:

```
$ cp mod5.1 mod5.3
$ chmod a-rwx mod5.3
$ chown partner mod5.3
```

You can change the ownership because the permissions are associated with your access to the contents of the file, *not* the ownership and group identifiers assigned to the file.

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**?

Answer:

```
$ cp mod5.1 mod5.4
$ chmod a-rwx,u+w,ug+r mod5.4
$ chgrp class2 mod5.4
$ chown root mod5.4
$ cat mod5.4cat: cannot open mod5.4: Permission denied
```

Since you are not currently a member of the group **class2**, you cannot access the file mod5.4.

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.)

Answer:

```
$ newgrp class2
$ cat mod5.4
This is the contents of mod5.1
$ newgrp
```

Once you change your effective group to class2, you can then access the file mod5.4.

Solutions

1. What are the permissions when you create a new file? Hint: Create a new file by using the editor, and copy or **touch** an existing file. Examine the permissions on the new files. How about a new directory? What is your current file creation mask?

Answer:

```
$ touch new_file
$ ls -l new_file
-rw-rw-rw- 1 YOUR_USER_NAME class     0    Jul 24 13:13 new_file
$ mkdir new_dir
$ ls -ld new_dir
drw-r---r-- 3 YOUR_USER_NAME class  1024 Jul 24 13:13 new_dir
```

```
$ umask
000
```

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.

Answer:

```
$ umask a-rwx,u=rw,g=r,o=r
$ touch new_file2
$ ls -l new_file2
-rw-r--r-- 1 YOUR_USER_NAME  class    0    Jul 24 13:13 new_file2
$ mkdir new_dir2
$ ls -ld new_dir2
drw-r--r-- 3 YOUR_USER_NAME  class 1024 Jul 24 13:13 new_dir2
```

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.

Overview of Module 7

Audience

general user — General system users

Product Family Type

open sys — Open systems environment

Abstract

This module is designed to introduce the student to the basic interactive capabilities of the POSIX shell. It describes the basic characteristics of the UNIX user environment. It does not cover all aspects of the environment in detail; however, it does explain it well enough to set some specific environment variables and to intelligently set variables needed by applications.

The details of shell variables are not covered here. This is meant to be an introduction to setting basic environment variables so the user's application will run correctly.

Time

Lab — 30 minutes

Lecture — 45 minutes

Prerequisites

In order to successfully complete this module, the student must be able to navigate the file system.

Instructor Profile

UX — General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000 — Logon access to an HP-UX system

Software Requirements

UX11	HP-UX release 11.0

Material List

P/N B2355-90033(T)	*HP-UX Reference Manual* , one per terminal
P/N B2355-90046(T)	*HP-UX Shells: User's Guide* , one per terminal

Lab Instructions

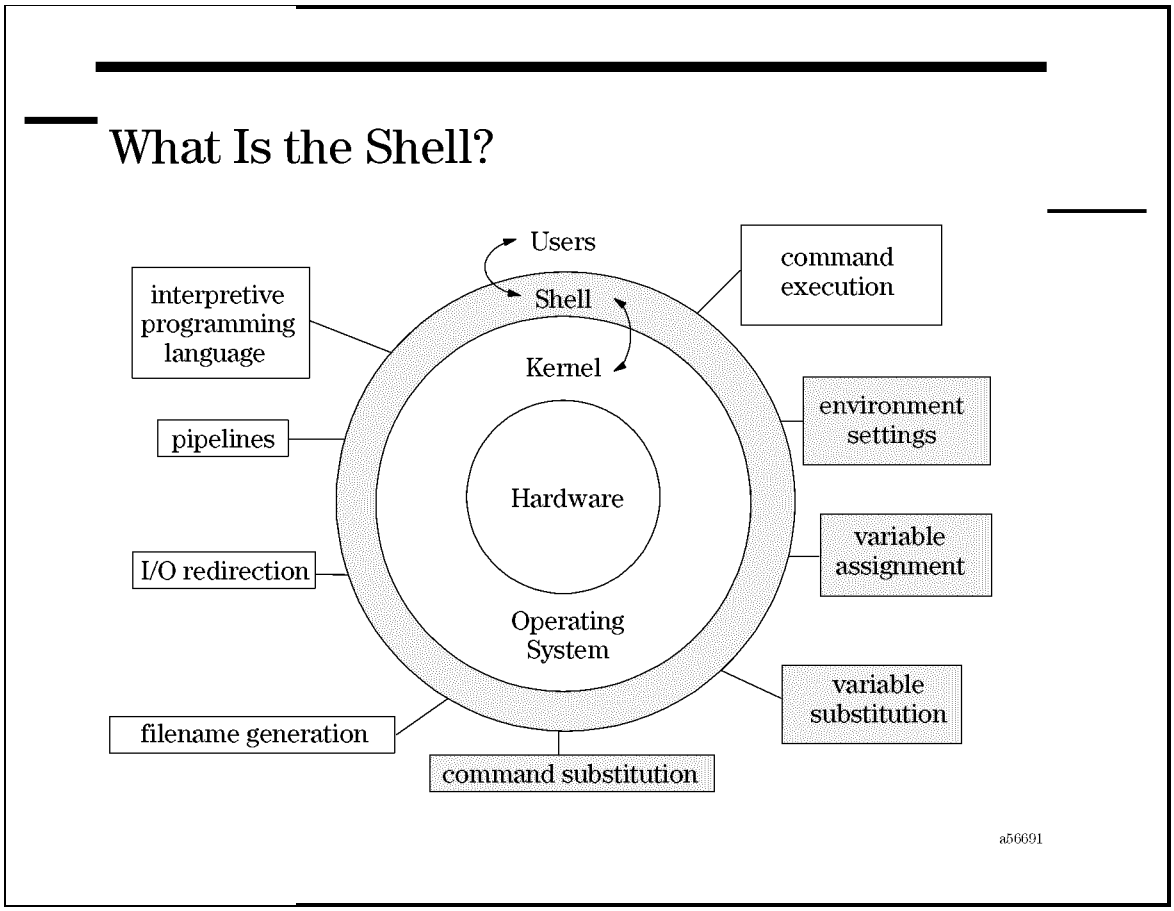
setup1	Create user logon of **user1**, **user2**, ... **user***n*, where *n* is the number of students in the class. Set up one user per student.
copyfiles	Copy the lab files to the users' home directories.
profile	Make sure that the students' **.profile** contains the lines: **EDITOR=/usr/bin/vi; export EDITOR** **FCEDIT=/usr/bin/vi; export FCEDIT**

Lab Files

```
-rw-r--r--   1 karenk   users    1482 May 28 16:12 frankenstein
total 14
drwxr-xr-x   5 karenk   users    1024 May 28 16:12 car.models
-rw-r--r--   1 karenk   users      17 May 28 16:12 cherry
-rw-r--r--   1 karenk   users      17 May 28 16:12 collie
drwxr-xr-x   4 karenk   users    1024 May 28 16:12 dog.breeds
-rw-r--r--   1 karenk   users      17 May 28 16:12 poodle
-rw-r--r--   1 karenk   users      17 May 28 16:12 probe
-rw-r--r--   1 karenk   users      17 May 28 16:12 taurus

tree/car.models:
total 6
drwxr-xr-x   2 karenk   users      24 May 28 16:12 chrysler
drwxr-xr-x   4 karenk   users    1024 May 28 16:12 ford
drwxr-xr-x   2 karenk   users      24 May 28 16:12 gm

tree/car.models/chrysler:
total 0

tree/car.models/ford:
total 4
drwxr-xr-x   2 karenk   users      24 May 28 16:12 sedan
drwxr-xr-x   2 karenk   users    1024 May 28 16:12 sports
```

```
tree/car.models/ford/sedan:
total 0

tree/car.models/ford/sports:
total 2
-rw-r--r--   1 karenk   users      18 May 28 16:12 mustang

tree/car.models/gm:
total 0

tree/dog.breeds:
total 4
drwxr-xr-x   2 karenk   users    1024 May 28 16:12 retriever
drwxr-xr-x   2 karenk   users      24 May 28 16:12 shepherd

tree/dog.breeds/retriever:
total 6
-rw-r--r--   1 karenk   users      27 May 28 16:12 golden
-rw-r--r--   1 karenk   users      29 May 28 16:12 labrador
-rw-r--r--   1 karenk   users      26 May 28 16:12 mixed

tree/dog.breeds/shepherd:
total 0
```

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.
- It handles I/O redirection and pipelines.
- It provides an interpreted programming interface, including tests, branches and loops.

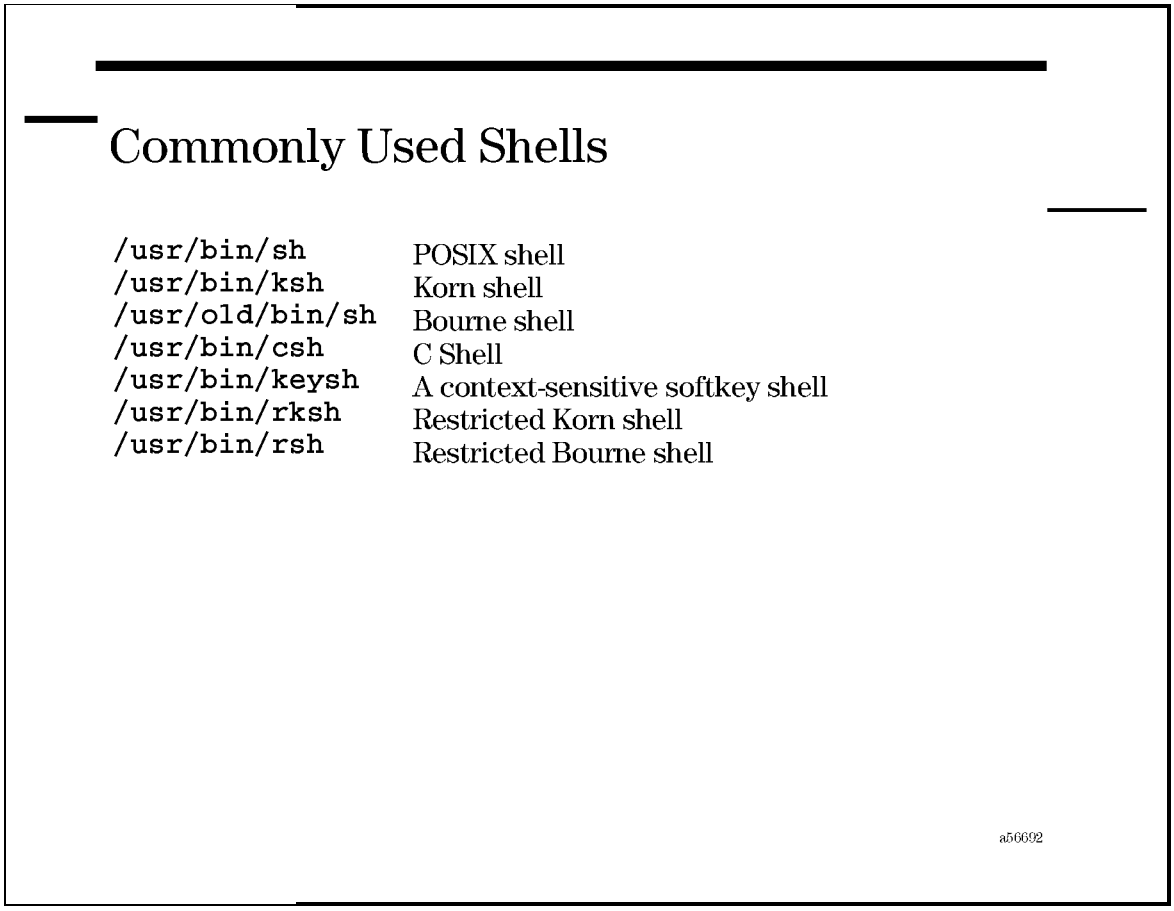
As you log in to a UNIX system, the shell will define certain characteristics for your terminal session, and then issue your prompt. This prompt defaults to a $ symbol in the case of the POSIX, Bourne and K shells. The default prompt for the C shell is the percent sign (%).

7-1. SLIDE: What Is the Shell? Instructor Notes

Key Points

- The shell is a command interpreter.
- The shell is a program *separate* from the operating system, allowing you to easily access your favorite command interpreter.
- It performs functions *in addition to* executing commands.
- Many of these functions are performed *before* the command is executed.
- The shell provides an interpreted programming language.
- The diagram on the slide reviews the shell's position in the UNIX operating environment. The shell has sometimes been compared to an egg shell.

7-2. SLIDE: Commonly Used Shells



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. The Korn shell was developed by David Korn of AT&T Bell Labs.

The Bourne shell is a command programming language and commands 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. The Bourne shell was developed by Stephen R. Bourne and 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. It was developed by William Joy of the University of California at Berkeley.

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 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 7-1. 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

7-2. SLIDE: Commonly Used Shells Instructor Notes

Purpose

To introduce the basic features of the POSIX shell.

Key Points

- The main advantage of HP POSIX Shell is that it has most of the POSIX related features and is compatible with the HP Korn Shell.

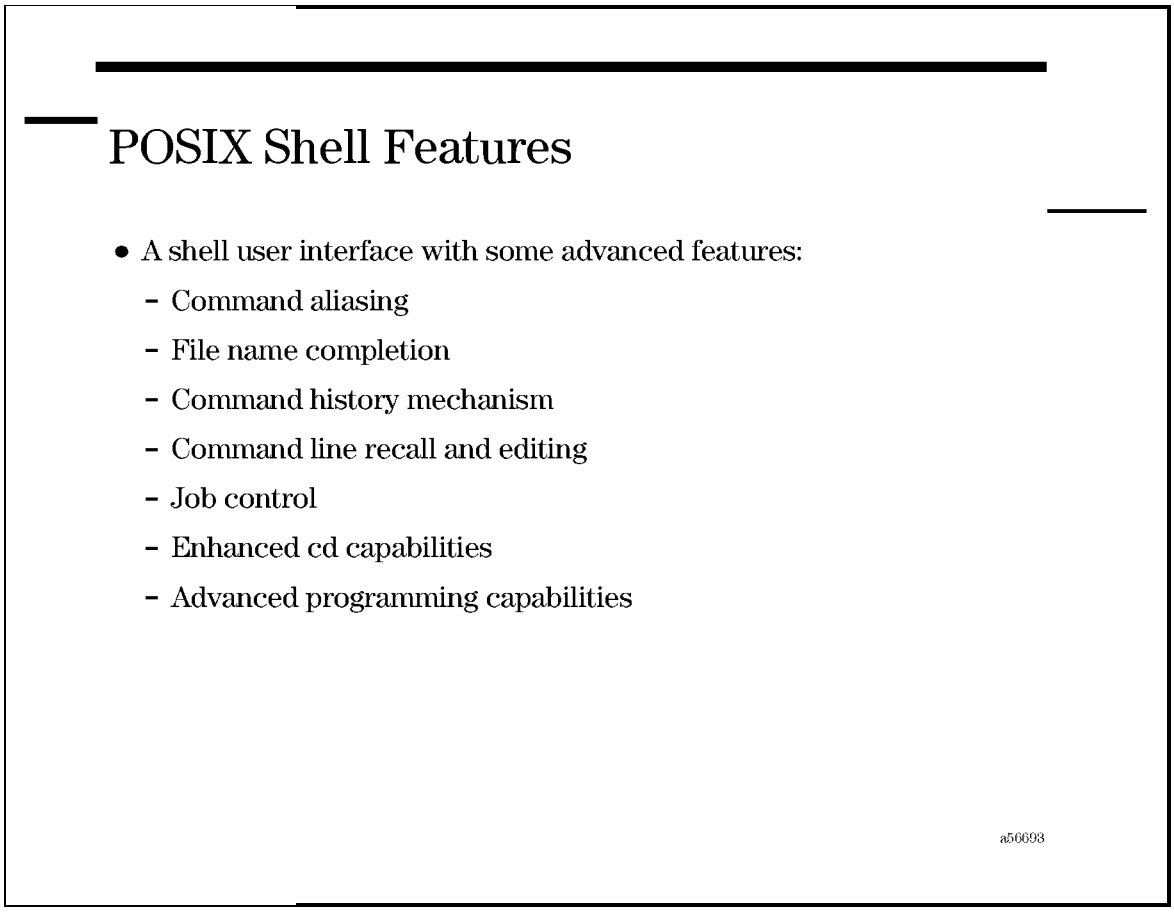
Background

The POSIX shell and the Bourne shell have the same name, **sh**. This was the requirement by the POSIX Committee. However, there are certain areas where these two shells are not compatible. HP has decided to currently have both the shells in the user default path.

Because of name similarities another directory was created and named as **/usr/old/bin/**. This directory contains an executable **sh** (Bourne Shell).

The **/usr/bin** directory is at the *beginning* of the default *PATH* variable. If you execute **sh**, and you are using the value of *$PATH* that is set in **/etc/profile**, you will be executing a POSIX shell, and not a Bourne shell; therefore, the login default shell is **/usr/bin/sh**. That is, if there is no login shell specified in the **/etc/passwd** file, then **login** assumes that you want the POSIX Shell as the login shell.

7-3. SLIDE: POSIX Shell Features



Student Notes

One of the shells provided with UNIX is the **POSIX shell**. This shell has many features that the Korn shell has but that the Bourne shell does not have. 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

7-3. SLIDE: POSIX Shell Features — Instructor Notes

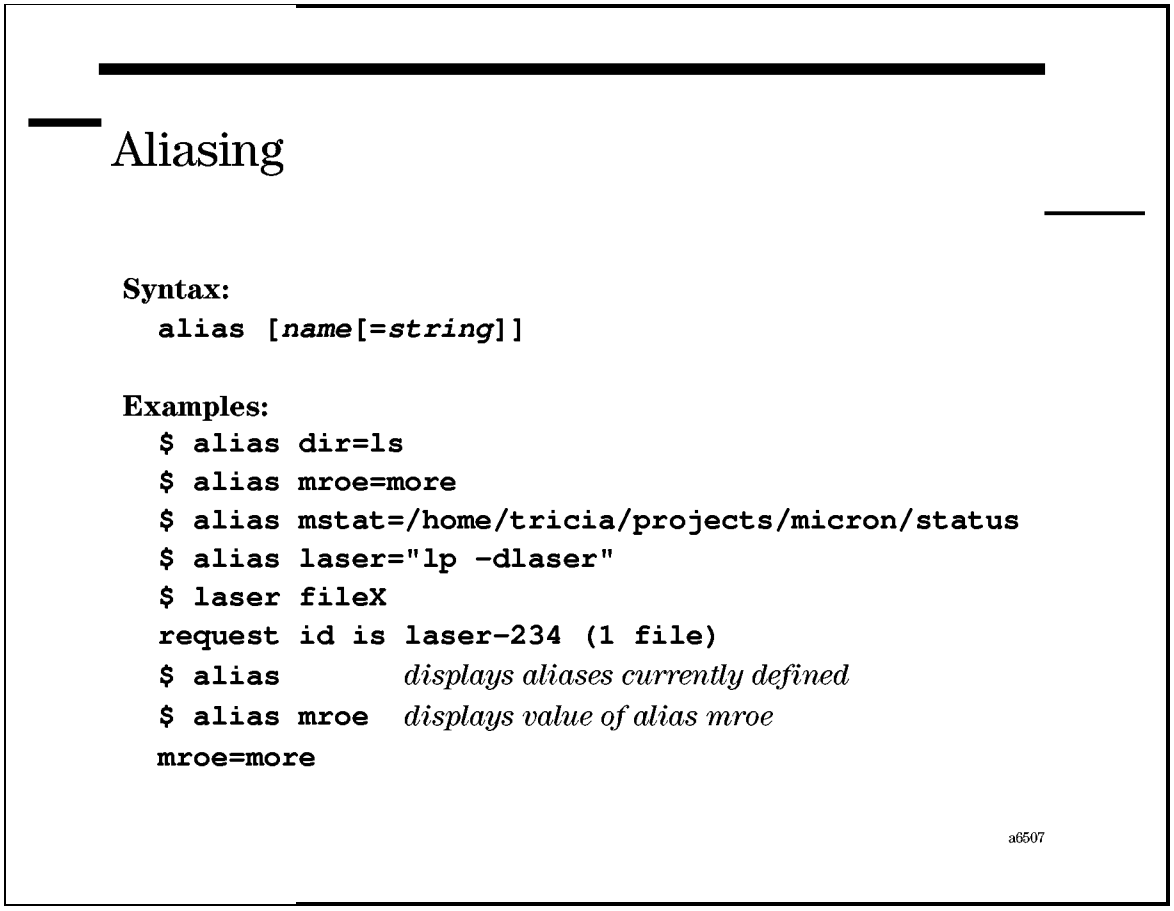
Purpose

To introduce the POSIX shell as a user interface.

Transition

Let's take a closer look at these features of the POSIX shell.

7-4. SLIDE: Aliasing



Student Notes

An **alias** is a new name for a command. Aliasing is a method by which you can abbreviate long command lines, create new commands, or cause standard commands to perform differently by replacing the original command with a new 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 if you could type **psf** instead? You create aliases using 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 *name* is aliased to. If the string contains spaces, you enclose the whole string in quotes. The alias is convenient to save 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.

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 (**alias dir=ls** or **alias copy='cp -i'** for example).

Aliases are also often used as a 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.

7-4. SLIDE: Aliasing — Instructor Notes

Key Points

- Aliases can be used as typing aids and to define new commands.
- Many aliases can be combined on a single command line.
- The **alias** command without arguments will return the values of all aliases that are set in the shell.
- 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.
- If spaces are included in the alias value, the value string must be enclosed in quotes. Quoting is covered in detail in another module.

Teaching Tips

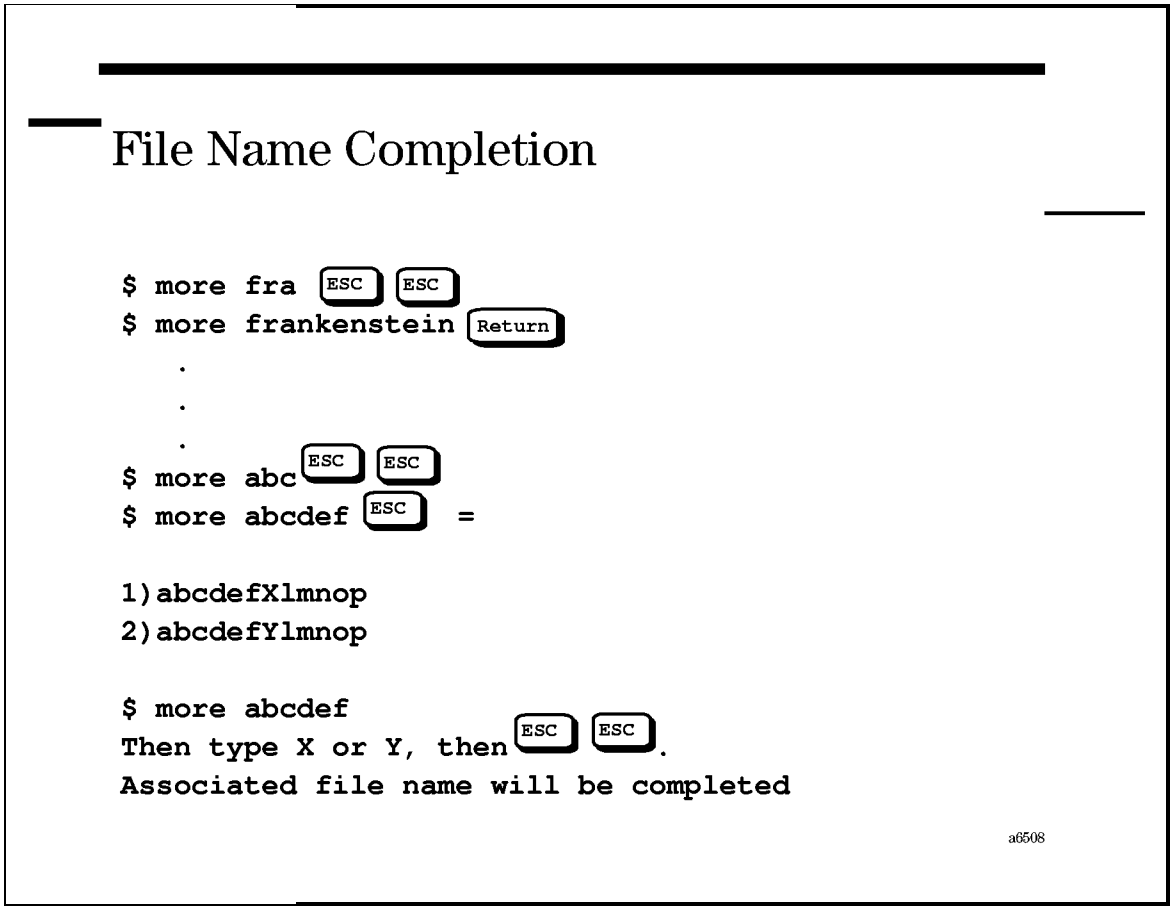
You may want to point out how to combine multiple commands for a single alias, for example,

```
$ alias go='cd /tmp;ls'
```

or

```
$ alias go=cd
$ alias there=/home/user3/tree/ford/sports
$ go there
$ pwd
$ /home/user3/tree/ford/sports
```

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. You 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
```

will cause the following command line to be displayed:

```
$ cd tree/dog.breeds/retriever
```

7-5. SLIDE: File Name Completion — Instructor Notes

Key Points

- Convenient typing aid.
- Used anywhere in a path name.

Teaching Tips

Note that after a list of possible file names has been generated, the command line *must* be changed with the command-line editor, or you can press another ESC and continue typing.

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
```

a56696

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 will display the last 16 commands you have entered and 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.

The *HISTSIZE* variable defines how many previous commands you will be able to access (the default *HISTSIZE* is 128 lines). The *HISTFILE* variable specifies a text file that is created that will 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.

7-6. SLIDE: Command History — Instructor Notes

Purpose

To introduce the concept of the command history stack, and the fact that commands can be recalled, edited, and re-entered.

Key Points

- The **`history`** command displays the last 16 commands.
- The command stack feature requires that certain environment variables be set.
- Using editor commands, you can modify your command line before re-entering the command.
- Using command numbers from the history stack you can re-enter the same command.

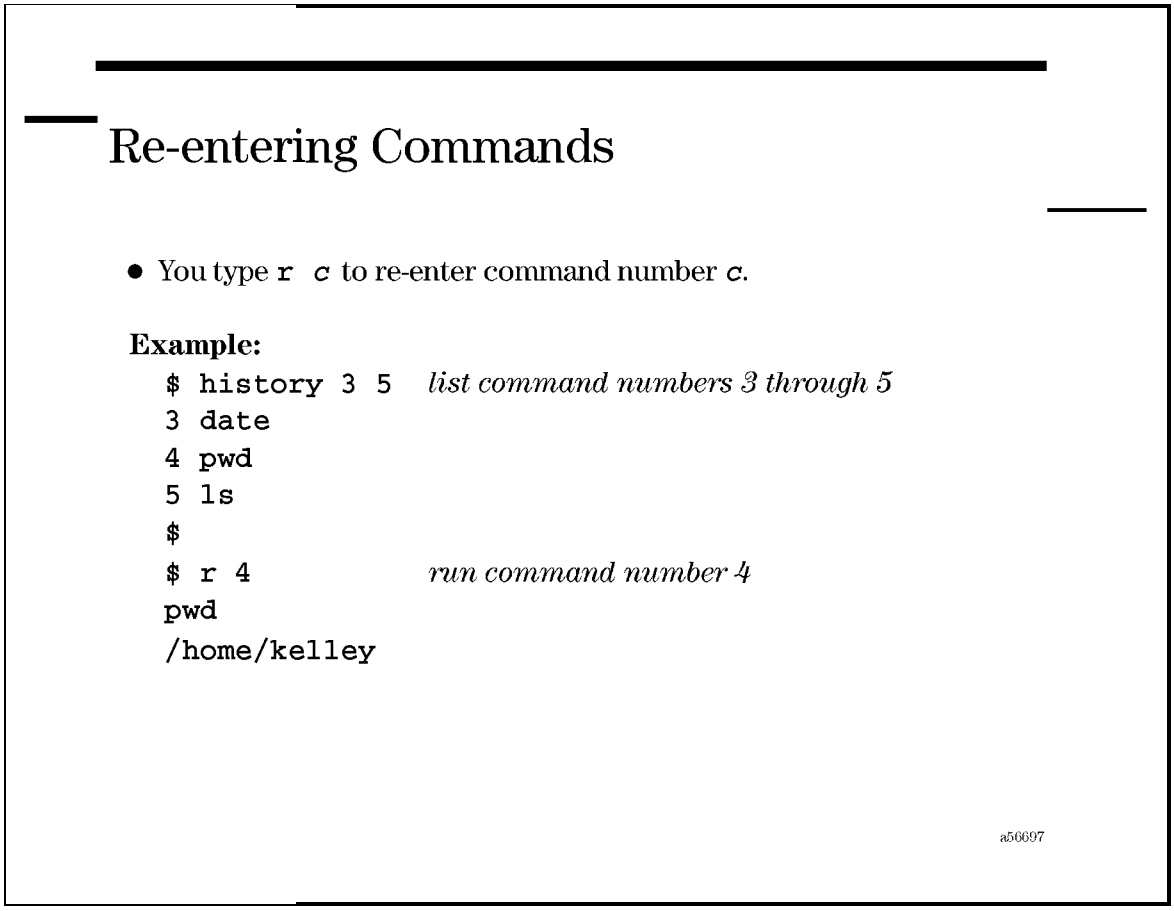
Teaching Tips

You should inform your students that the history file will grow, without bound, from session to session. A sample **`.logout`** is provided, which will rename **`.sh_history`** to **`.sh_hist.old`** when the user logs out, so that he or she does not have to worry about periodically cleaning this file out.

Transition

Now that you have located a command you wish to run again, you can either run it *as is*, or you can edit it before you run it again. First we'll look at how to run it using the command number.

7-7. SLIDE: Re-entering Commands



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,

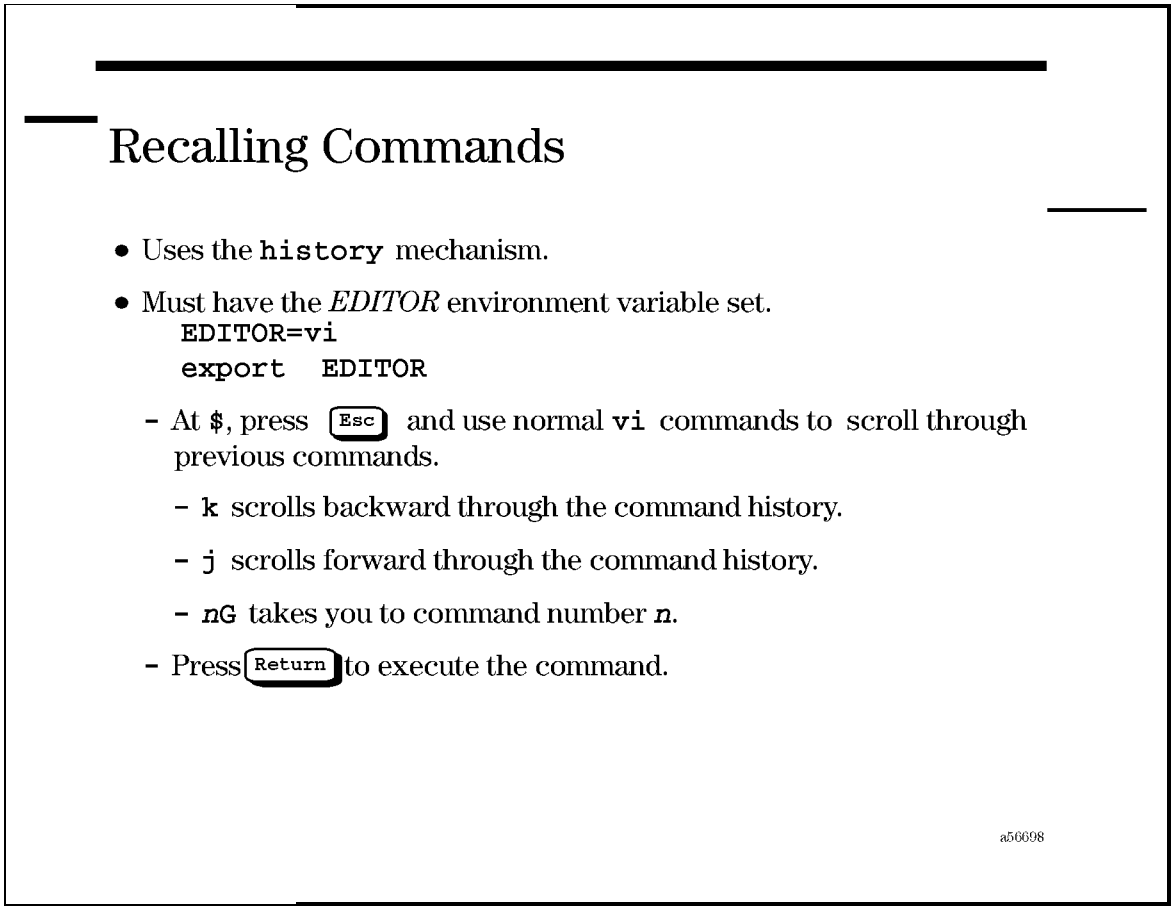
```
$ history
1 date
2 cat file1
3 ls -l
$ r d
Mon Jul 4 10:03:13 1994
```

7-7. SLIDE: Re-entering Commands Instructor Notes

Transition

Now let's look at how to recall and re-enter commands using a specific editor.

7-8. SLIDE: Recalling Commands



Student Notes

The most widely-used editor embedded in the UNIX shell is **vi**, and basic functions of this editor will be used to illustrate command line editing. Detailed use of **vi** will be covered in a later module or in a later course.

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. If you execute the **env** command, you should see this in the listing:

```
$ env
.
.
EDITOR=/usr/bin/vi
```

```
.
.
```

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** 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 to execute again on your command line, just press Return.

You can also use the **history** command to see your last 16 commands. This will list the number of the command with the command line. If you want to execute a particular command, type Esc *n* **G**, where *n* is the command line number from the **history** listing. The **G** command in **vi** moves you to a specific line.

7-8. SLIDE: Recalling Commands — Instructor Notes

Purpose

To introduce the POSIX shell command recall facility.

Key Points

- The *EDITOR* variable must be set in order to use the recall and editing features of **/usr/bin/sh** . This is normally done in the user's **.profile** .
- You must enter ESC to toggle into *command mode* .
- Just use **vi** commands to scroll through the command stack.

NOTE: The information on **vi** has been moved to a later module, so the instructor will have to teach basic **vi** skills here to be successful.

- To execute the displayed command, just enter Return .
- The **history** command displays the last 16 commands.
- Point out that the environment variable *EDITOR* defaults to **/usr/bin/ed** , which should be changed in the user's **.profile** . Also note that the arrow keys cannot be used to edit the command line, even in UNIX.

Activity

1. Enter the following commands:

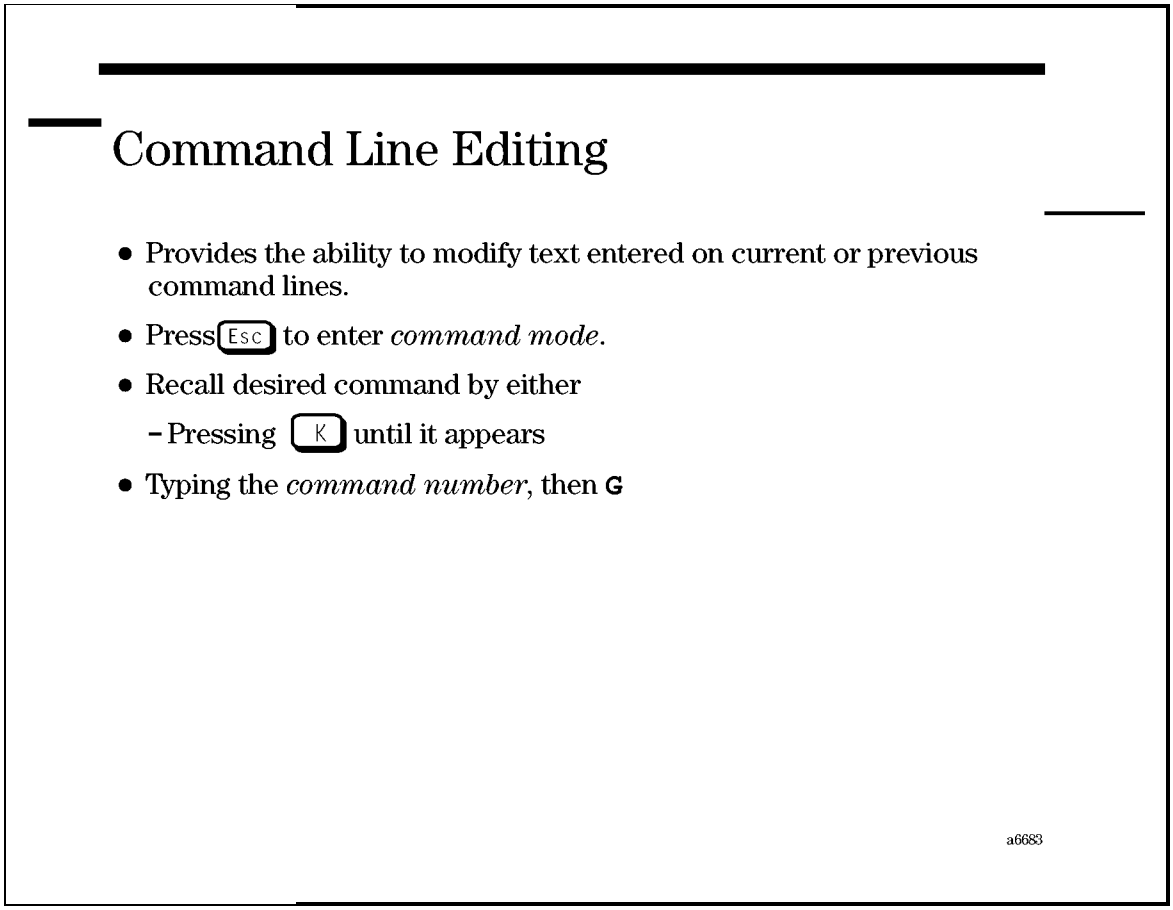
```
$ env                          Note the value of the EDITOR variable
$ ls
$ cd
$ cd /tmp
$ pwd
$ history
```

2. Use the recall feature to re-execute the command **cd**. Confirm that you have changed to that directory by recalling the **pwd** command.

Transition

What if we need to change a previous command before we can re-execute it?

7-9. SLIDE: Command Line Editing



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:

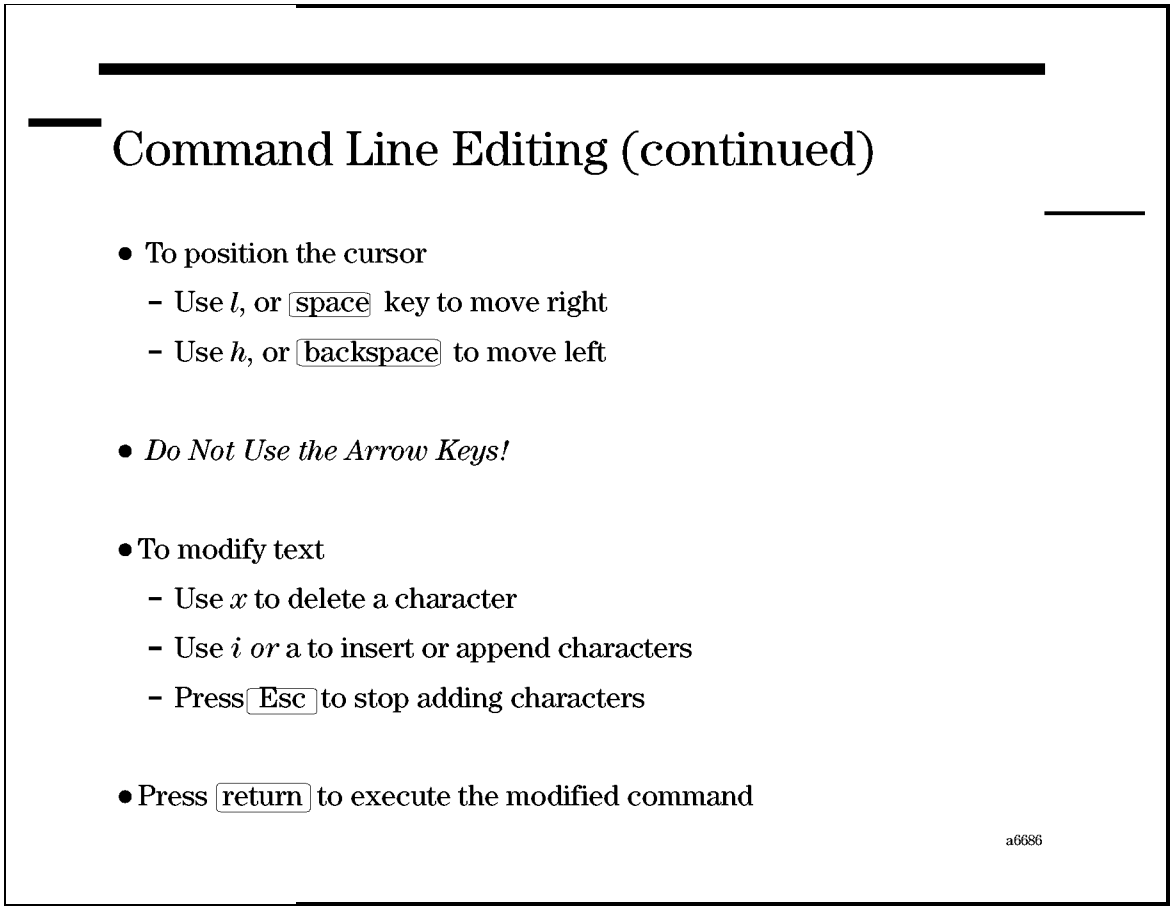
```
cd
```

An alternate way of locating commands in the command stack is to press ESC **k**, as before, and 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 would recall the next to last **cd** command, and so on. Once you have searched for a pattern, typing **n** will also search for the next occurrence.

At this point, you could 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.

7-9. SLIDE: Command Line Editing — Instructor Notes

7-10. SLIDE: Command Line Editing (continued)



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. Once again, this is done with the **vi** editing commands.

To change a command line, you must press Esc to enter the **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, the **vi** commands work. 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 utilize 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:

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, you need to 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—k. As you continue to enter k's, you will step back through your previous commands. When the command is displayed that you wish to run, just press the Return key, 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 using **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**. 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 just press Return and the **cd** command will execute.

If you had problems editing the line and want to try again, just press Break to cancel editing, and 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.

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.

7-10. SLIDE: Command Line Editing (continued) — Instructor Notes

Key Points

- Whether you are typing a command line or you have just recalled a command line, you can edit it using the **vi** editing commands.
- Pressing Break cancels the editing and puts you back to normal POSIX shell mode.
- The cursor can be *anywhere* in the command line when the Return is pressed.
- In addition to editing previously entered commands, you can use the **vi** commands to modify the current line prior to entering it.
- The plus (+) and minus (-) can be used instead of **j** and **k** .

Evaluation Question

Will **vi** commands such as **G**, **D**, **p**, and **s** work? Sure! The results may be a little strange, however, because you can only see one line at a time.

Teaching Tips

If you have students running under windows, and they press the **v** command to edit a previous line, and *close* the window without properly concluding the **vi** session (they do not enter **:wq** or **ZZ**), newly opened terminal emulator windows can hang in infinite **ex** command sessions.

If you do a **ps -ef | grep vi**, you will probably find a **vi** session still running in the background. You will have to kill this process in order to open any new terminal emulator windows.

The fc Command

The POSIX shell supports an alternative command line recall mechanism known as **fc**, which stands for fix command. It can be used to list the contents of the history stack, or re-execute previously entered commands. Command lines can be referenced either through their command line number or mnemonically through the command name. Enough text needs to be provided to uniquely identify the command string you want to re-execute.

The **history** command is an alias that is compiled into the shell. It executes **fc -l**, but you can reassign the value of the history alias.

fc is not used as often as the **vi** feature.

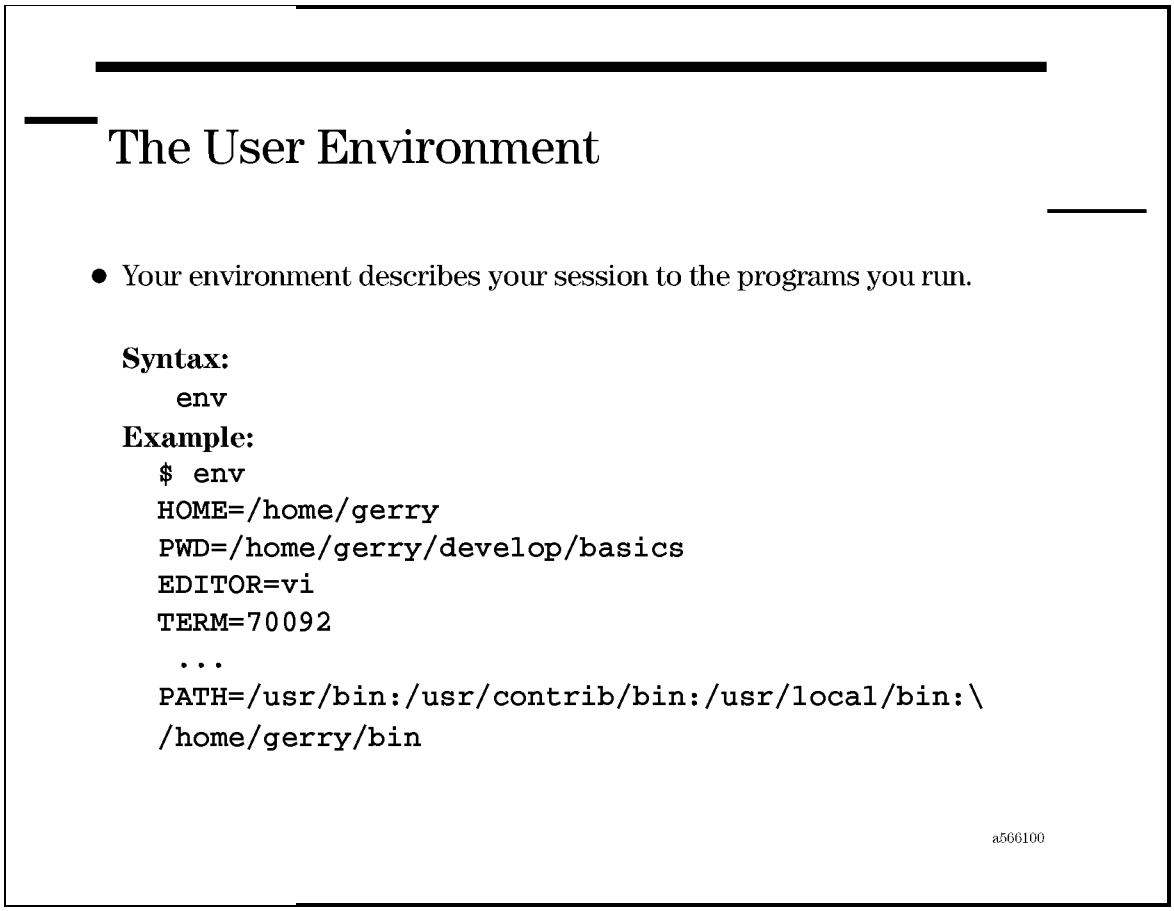
Syntax:

fc -l	List commands
fc -e - X	Fix commands

Examples:

```
$ fc -l                          displays history stack
6 more funfile
7 man ls
8 ls -F tree
$ fc -e - 8                      execute command line number 8

car.models/ ...
$ fc -e - m                      execute last command that starts with m

man ls
$ fc -e - mo                     execute last command that starts with mo

more funfile
```

7-11. SLIDE: The User Environment



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 you are working in
- Who 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.

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 like your user environment is unique.

Many applications require you to customize your environment in some way. This is done 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** .

7-11. SLIDE: The User Environment — Instructor Notes

Purpose

To show a sample user environment using **env** and describe the purpose of some of the more important variables.

Key Points

- **env** displays all environment variables set by the system and the two login files.

Presentation Suggestions

If you have an advanced group, you might go into some detail about the other environment variables that students may see in an **env** listing and describe their purpose. For example, the Korn and POSIX shells use the variables *ENV*, *HISTFILE*, and *HISTSIZE*, X11 uses *DISPLAY*, *WMDIR*, and *WINDOWID*, and so on.

Teaching Question

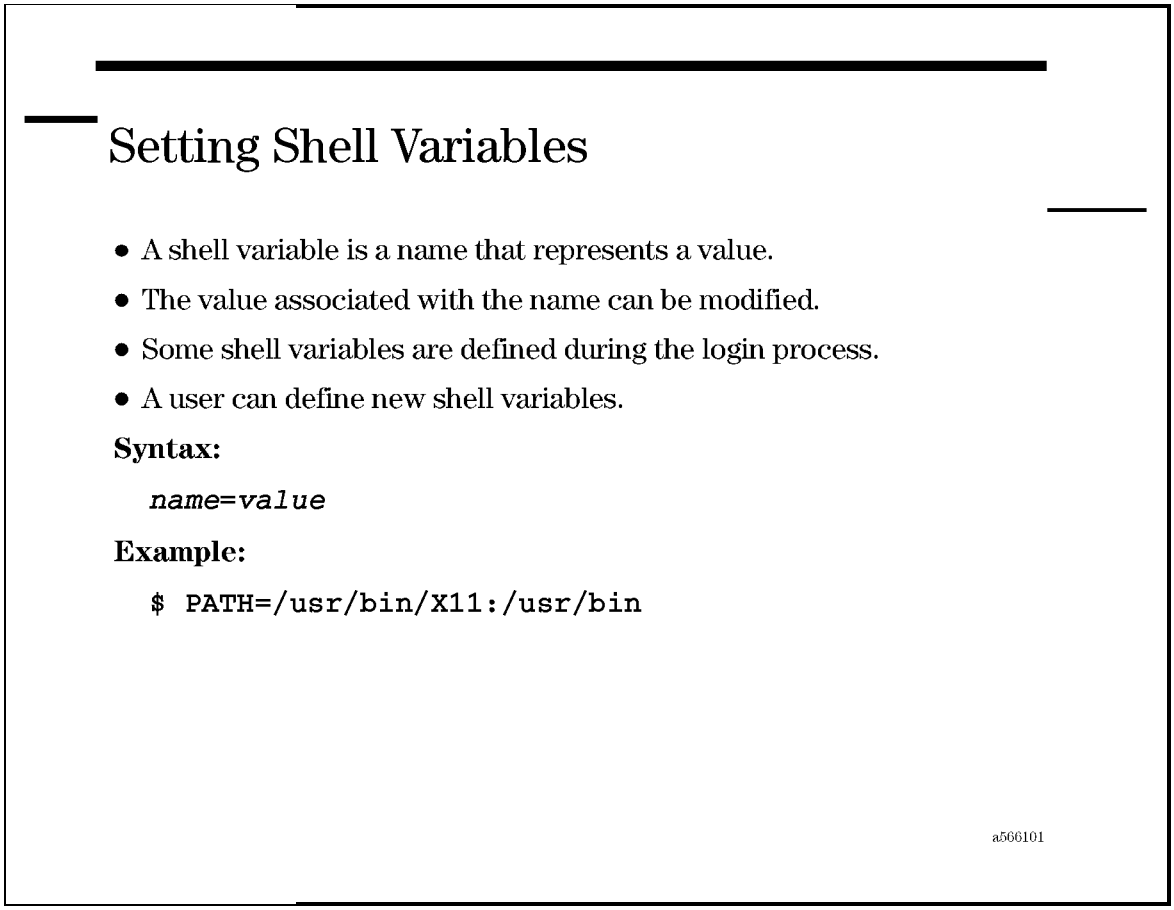
Ask what the other environment variables that are listed might do.

Transition

There are three especially important items in your environment that may have to be customized:

- The *PATH* variable (where to find commands).
- The *TERM* variable (your terminal type).
- The special environment variables for applications.

7-12. SLIDE: Setting Shell Variables



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. When the variable value is set by performing an assignment statement, such as

```
TERM=70092
```

This tells the shell to remember the name *TERM*, and that when the value of the variable *TERM* is requested, respond with *70092*.

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.

7-12. SLIDE: Setting Shell Variables — Instructor Notes

Purpose

To present the basics of setting shell variables as they relate to the user environment. The details of setting, exporting, and dereferencing shell variables are presented in the module "Shell Advanced Features."

Key Points

- A value is assigned to a variable using: **`name=value`**
- No spaces around the equal sign

Teaching Tips

The purpose of this topic is to just introduce the concept of setting environment variables. The details of the shell environment are discussed in another chapter.

The shell does not support variable *data types* in the sense that a programming language does. All variable values are stored as strings of characters. However, the POSIX (and Korn) shell built-in command **`typeset -i`** may be used to tell the shell that the variable is an integer; this will make arithmetic faster. See **`sh-posix(1)`** for more information.

You may want to have the students execute the assignment statements presented on the slide.

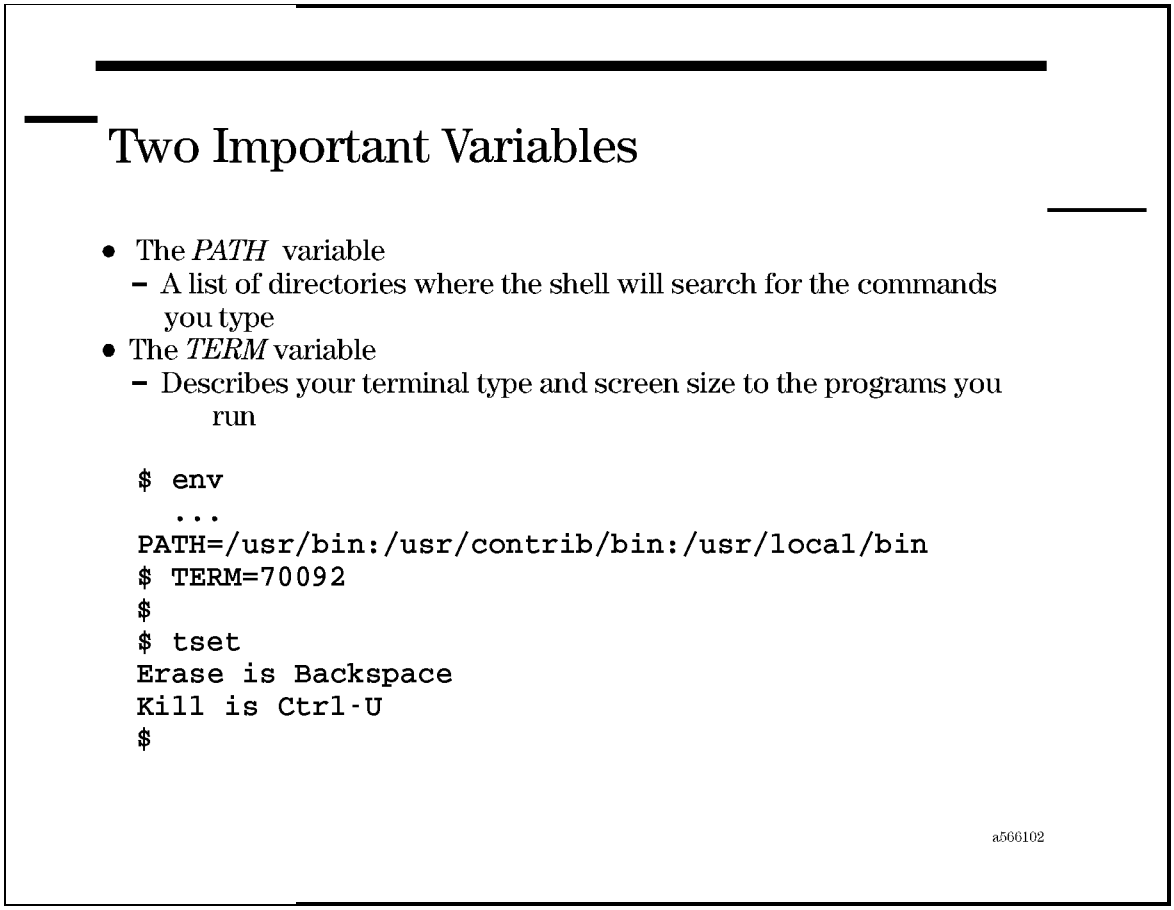
Teaching Question

Ask students what the outcome of the following command would be:

```
$ my_cp=cp f1 f2 f3 /tmp
```

The shell would see the blank spaces, and generate a syntax error message. The blank is the delimiter between arguments, and the assignment expression does not accept any arguments.

7-13. SLIDE: Two Important Variables



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** 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 of 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 so it never asks you about your terminal type. In this case you should 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 should not have 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. This will reset the terminal characteristics using the terminal type found in the *TERM* variable.

7-13. SLIDE: Two Important Variables — Instructor Notes

Purpose

To describe the importance of the *PATH* and *TERM* variables and how they can be changed. The *TERM* is quite often incorrect unless the system administrator has set it up already. Users need to know what the correct terminal types are for their systems.

Key Points

- The **`command: not found`** error normally occurs because an improper command name was entered or the *PATH* was not set correctly.
- If you are on a workstation or PC and the *TERM* variable is correct, you will probably never have to change it.
- The command **`eval 'tset -s type '`** will set the terminal type in one step.

Teaching Tips

If you feel it is appropriate, you might wish to have the students **`echo $TERM`** to see the value of the *TERM* variable. Variable substitution is discussed in the module "Shell Advanced Features."

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
TERM=2392a	designates the terminal type of your terminal use the command: **eval 'tset -s -Q -h'** During startup, this 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 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 on 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, you can 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 1024 pixels)	98730, hp98730, 98730a, hp98730a, 98731, hp98731, 98731a, or hp98731a

7-14. TEXT PAGE: Common Variable Assignments — Instructor Notes

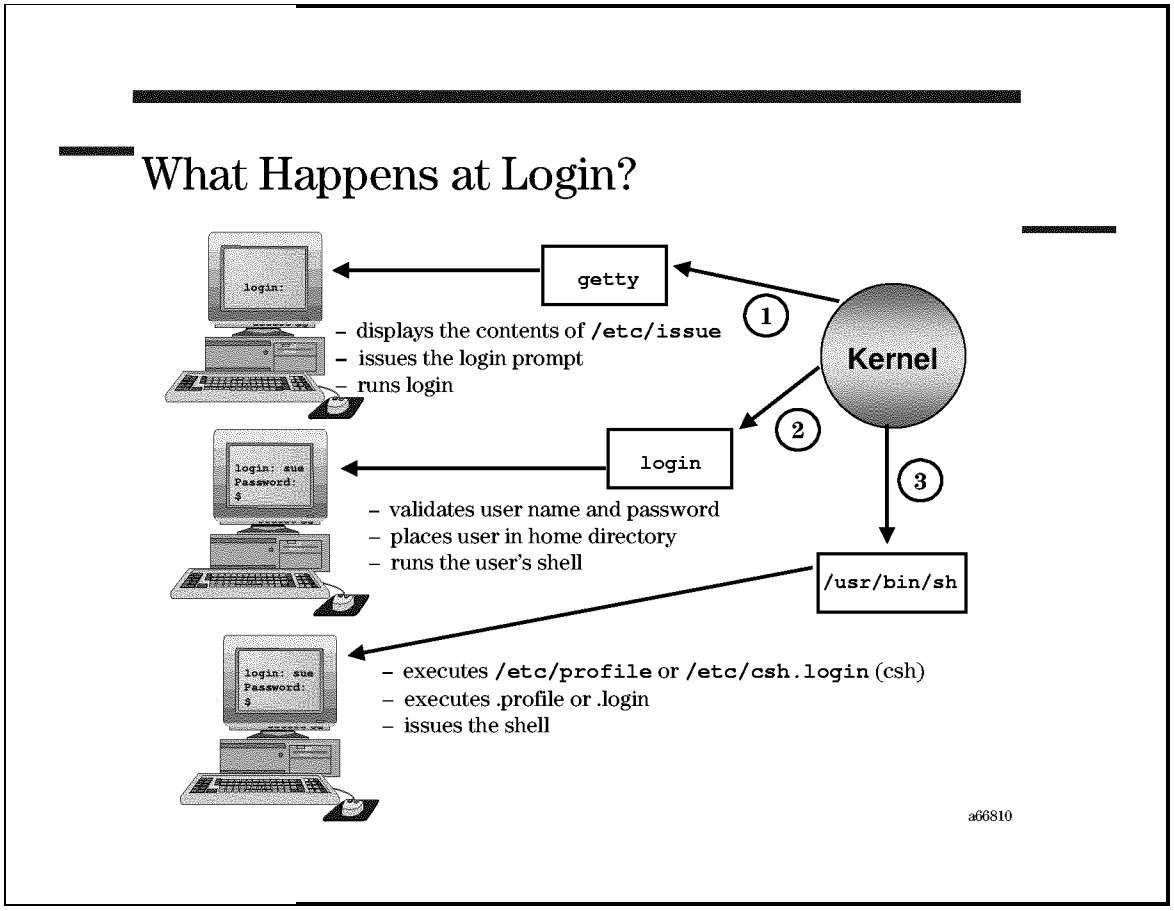
Teaching Tips

Depending on the level of the students, you might want to point out some other variables such as:

- **~** (tilde)—not really a *variable* that can be assigned but is equivalent for $HOME and useful when you need to designate an absolute path name to a directory or file
- TMOUT—allows users to set a timeout (in seconds) on their terminals
- PS1—can be used to display command number and current directory in your prompt. Not usually an environment variable.

You might want to present a brief discussion on the *TERM* variable, since this is crucial to the proper operation of commands such as **`more`** and **`vi`**.

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 will place you in your home directory and start the shell for you. We have seen this happen each time we logged in. Our 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

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 will read commands from a system command file called **/etc/profile**. Whenever someone logs in and starts a shell, this file will be read. There is also a file called **.profile** in your home directory. 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 will take 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**:

```
DTSOURCEPROFILE=true
```

Otherwise, only **.dtprofile** will be executed at login. **.dtprofile** contains commented lines of setup variables you need to set the CDE environment.

7-15. SLIDE: What Happens at Login? Instructor Notes

Purpose

To introduce how the system sets up the initial user environment.

Key Points

- Everyone who logs in reads **`/etc/profile`** when his or her shell is first started.
- Each user has a **`.profile`** to further customize his or her environment beyond what **`/etc/profile`** does.
- **`.dtprofile`** is read *before* any other user's profile files. For more information on **`.dtprofile`** , refer to **`dtlogin(1)`**.

Evaluation Questions

Which users use **`/etc/profile`**?

Answer: Those who log in at the POSIX, Bourne, or Korn Shell.

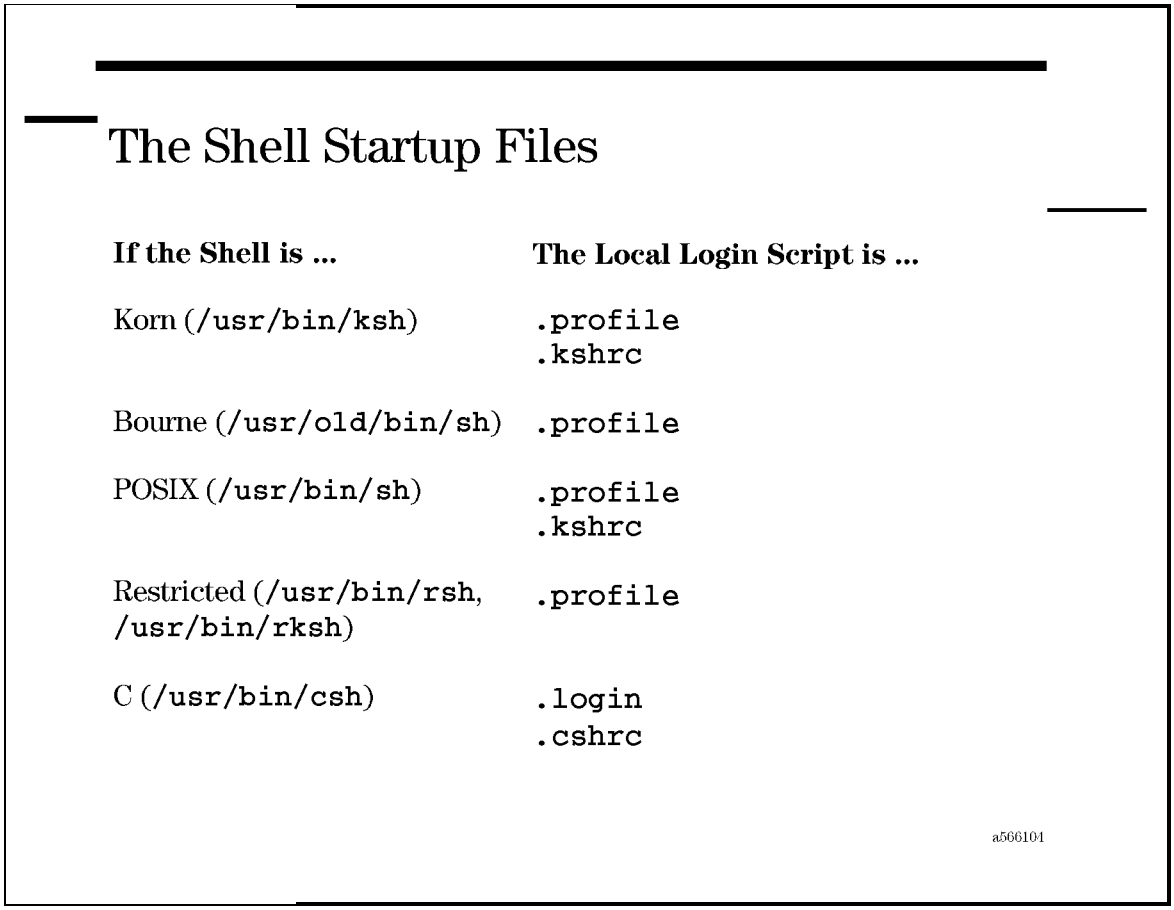
When is **`.profile`** read?

Answer: When a user enters a POSIX, Bourne, or Korn Shell.

Transition

Let's take a more detailed look at the user environment.

7-16. SLIDE: The Shell Startup Files



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 assist you in customizing your session, the files **`.profile`** and **`.kshrc`** are available. These are simple shell scripts that will define environment variables, define aliases, and execute programs upon login. Remember that the POSIX shell originated from the Korn shell, which originated from the Bourne shell. Therefore, it supports the same configuration files in addition to the **`.kshrc`** file.

`.profile`

Any user who wishes to customize the default environment provided by his or her system administrator will create or modify **`.profile`** . This file commonly will define or customize environment variables, set up the user's terminal, and execute programs such as **`date`** during session log in. A user's application can also be initiated from **`.profile`** by **`exec applicationname`** . In this way the user will never have access to a shell prompt, and when the application is exited, the user will be 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 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 **.kshrc**.

.kshrc

The POSIX and K shells have an optional configuration file called **.kshrc**. It is used much like **.profile** to configure your user environment. Unlike **.profile** , however, **.kshrc** is 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 X11 Windows, you may have several shells running at once. You can use the **.kshrc** file so that every one of those shells looks the same.

The file name **.kshrc** is not a required file name. When you invoke the shell, it looks for the file referenced by the *ENV* variable. This file is often named **.kshrc**, but it may be named anything you wish.

To use your **.kshrc** 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
```

This tells the K shell that you want to use the **.kshrc** file in your home directory (**~/.kshrc**). Now just add all of your alias commands to **.kshrc**.

If you are in an environment where you are using the Bourne and the POSIX shells, you might want to store POSIX shell specific variable assignments in this file, since 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 the Korn shells. If found, the commands in the file **.cshrc** are also executed before you get your first C shell prompt.

7-16. SLIDE: The Shell Startup Files — Instructor Notes

Key Points

- Variables defined in **/etc/profile** are overridden by definitions in **.profile** .
- The file name for **.kshrc** is defined as an environment variable, *ENV*.

Teaching Tips

You might want to mention the . (dot) command, that allows you to execute your configuration files such that your current environment is updated, instead of being run in a subshell, which cannot update your environment. Here is an example:

- Edit the **.kshrc** file and uncomment one of the PS1 designations.
- Execute

```
$ . .kshrc (or . ./.kshrc, depending on your PATH)
```

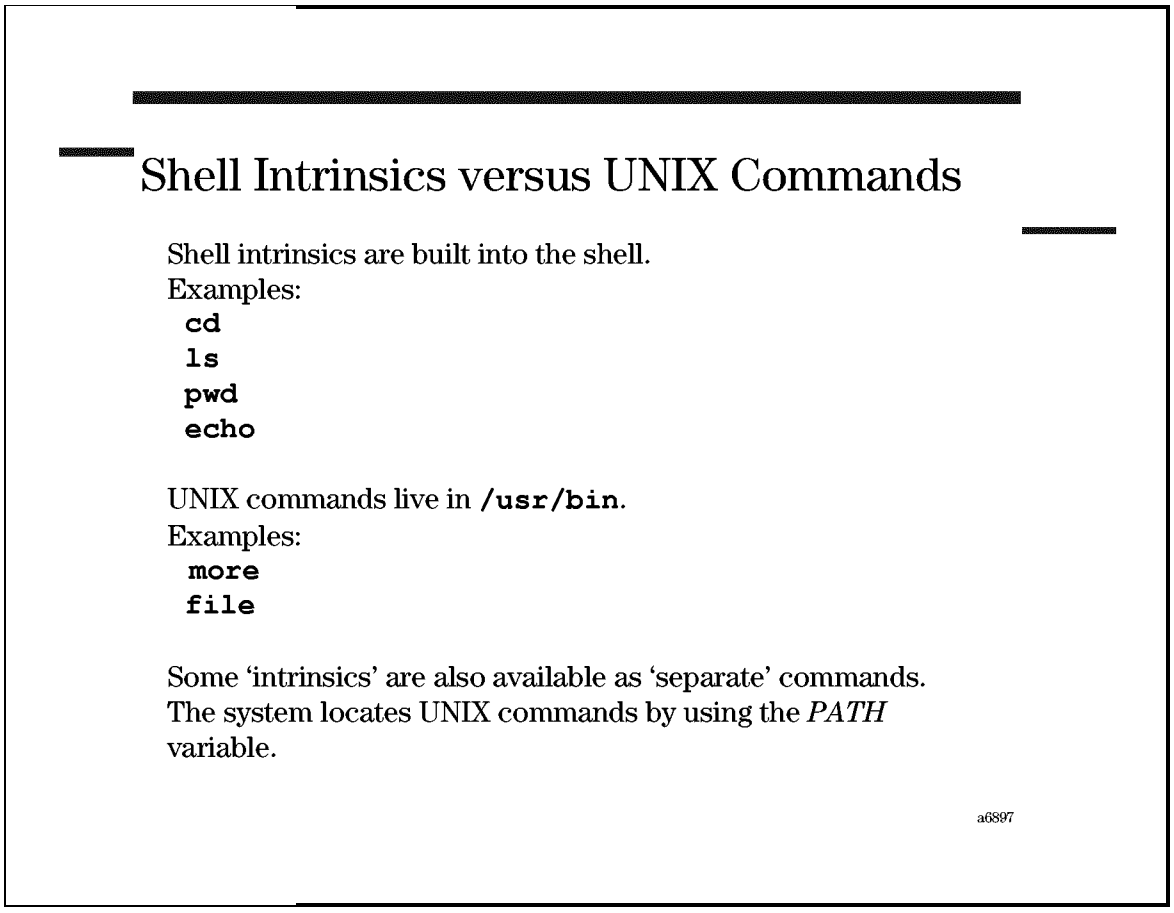
- You will see your prompt updated.

The only other alternative is to log out and log back in to have your modified configuration files take effect.

If there is interest in the C shell, you might want to mention the **.logout** file. The C shell also provides you with a file that is executed when you logout. The file is called **.logout** and is found in your home directory.

The **.logout** file is often used to place clean up commands to be executed when signing off the system. Typical functions are to record the logout time, remove temporary files, and clear the screen. In combination with the shell's history mechanism, the **.logout** file can be used to keep a current log of all your session activities.

7-17. SLIDE: Shell Intrinsics versus UNIX Commands



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.

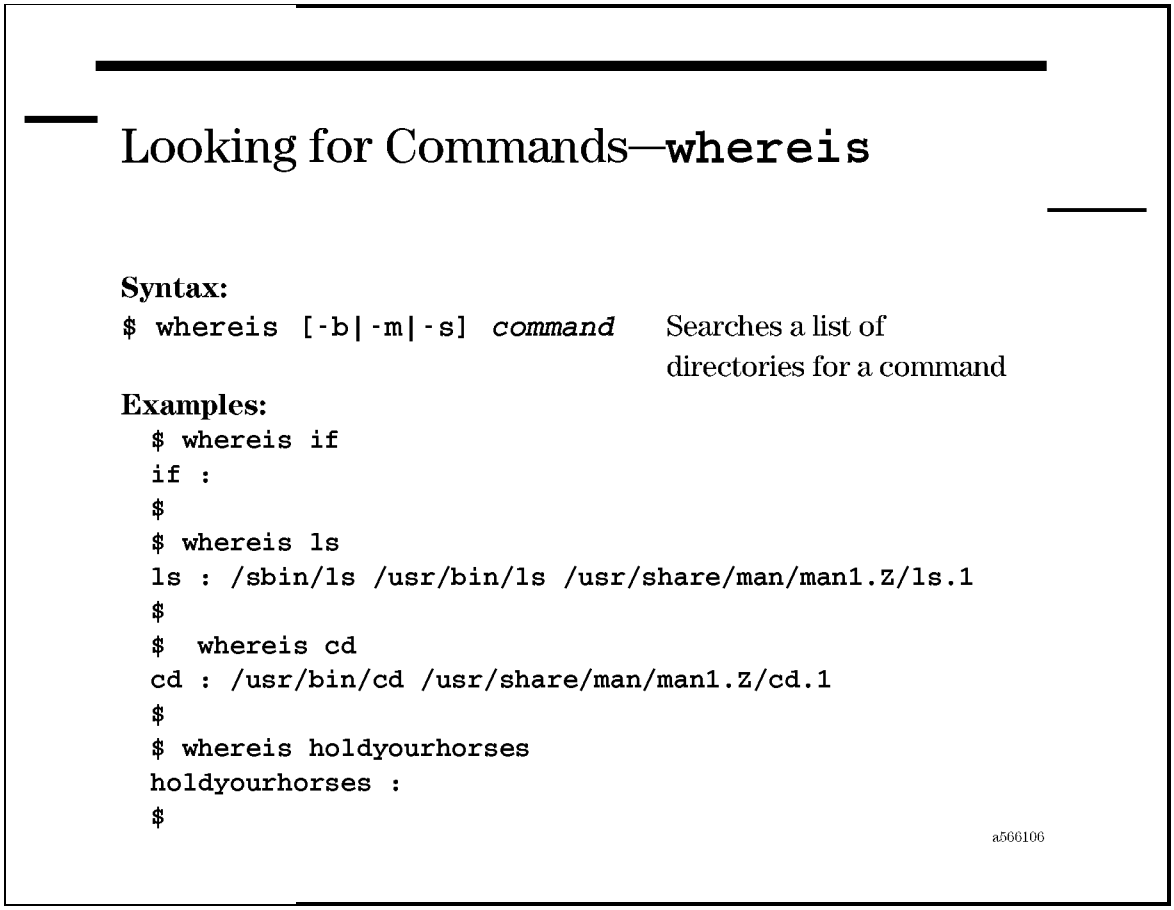
UNIX commands can have the same name as shell intrinsics; however, to access these commands, the user must use the command's absolute *PATH* name to inform the shell to use *it* rather than the intrinsic of the same name.

7-17. SLIDE: Shell Intrinsics versus UNIX Commands — Instructor Notes

Teaching Tips

Have the students look up **`sh-posix(1)`** in the manual. Have them read several of the intrinsic commands to you.

7-18. SLIDE: Looking for Commands — whereis



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** will 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.

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 that the **whereis** command searches, use the flags **-b**, **-m**, or **-s** to limit the search to binary, manual pages, or source code, respectively.

7-18. SLIDE: Looking for Commands — `whereis` Instructor Notes

Teaching Tips

Point out that the **`whereis`** command searches only a small part of the entire file system.

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
```

7-19. TEXT PAGE: Sample .profile — Instructor Notes

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 this file out. Therefore, you might 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 will be 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
```

7-20. TEXT PAGE: Sample `.kshrc` and `.logout` — Instructor Notes

7-21. LAB: Exercises

Directions

Complete the following exercises and answer the associated questions.

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 when separated with a semicolon.)

4. Log out and then log back in to test your aliases. Why did you have to log out?

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`**.

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, and 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.

7-21. LAB: Exercises

Instructor Notes

Time: 30 minutes

Purpose

To practice some of the features of the POSIX shell. These exercises allow your students to experiment with some of the basic features of the shell as a command interpreter. They will set up aliases, practice with the command stack, and practice file name completion.

Solutions

1. Create an alias called **h** that executes the **history** command.

   Answer:

   ```
   $ alias h=history
   ```

2. Check the commands in the **.shrc** file in your home directory. Add your **h** alias to the list.

   Answer:

   ```
   vi .kshrc
   ```

   add the line

   ```
   alias h=history
   ```

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 when separated with a semicolon.)

   Answer:

   ```
   $ alias go="cd /home/user3/tree; ls -F"
   $ go
   car.models/ dog.breeds/ fruit/ horses/
   $ pwd
   /home/user5/tree
   $ cd
   ```

4. Log out and then log back in to test your aliases. Why did you have to log out?

   Answer:

   You had to reread the **.profile** and **.kshrc** files. The easiest way is to log out and then log back in.

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**?

Answer:

Typing the command line given puts **more f** on the command line, and the shell beeps because there is more than one file starting with **f**. If you type an **u** and then Esc Esc again, the file name **funfile** will be completed for you.

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.

Answer:

```
$ cp fr[ESC] [ESC] tree/ca [ESC] [ESC]ford/sports
$ cp frankenstein tree/car.models/ford/sports
```

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).

Answer:

```
$ cd /user/spol/ko/interface[Esc]
```

Using Backspace and the space bar to position the cursor, use **vi** commands **x**, **a**, **cw** to make the appropriate changes. Remember to use Esc whenever you need to leave input mode.

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.

Answer:

```
$ ls -F
$ [Esc] [k]
```

Now use the **r** command to change **ls -F** to **ls -l** and press Return.

9. Using the command stack, recall the previous copy command, and change **frankenstein** to **funfile**.

Answer:

```
[ESC] [k]
$ cp frankenstein tree/car.models/ford/sports
[l] [l] [l]  or  [w]
```

```
 c  w  funfile Return
$ cp funfile tree/car.models/ford/sports/
```

10. Recall the previous copy command, and modify it so that you display the contents of the sports directory.

Answer:

```
ESC  k
$ cp funfile tree/car.models/ford/sports
c  w  ls ESC                          change word cp to ls
w
d  w                                  to delete funfile
Return
$ ls tree/car.models/ford/sports
```

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.

Answer:

```
ESC  k
$ ls tree/car.models/ford/sports
c  w  cd ESC                          change word ls to cd
w  w                                  repeat until cursor is under sports
c  w  sedan ESC                       change word ls to cd
Return
$ cd tree/car.models/ford/sedan
$ pwd
tree/car.models/ford/sedan
```

12. Change back to your *HOME* directory, and 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.

Answer:

```
$ cd
$ history
$ r 'cd t'    or    r cmd_number
$ r 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.

Overview of Module 8

Audience

general user — General system user

Product Family Type

open sys — Open systems environment

Abstract

To teach students to work effectively in the POSIX shell environment, using and setting shell variables, manipulating the environment, and interacting with child processes

Time

Lab — 45 minutes

Lecture — 90 minutes

Prerequisites

m50m — Shell Basics

Instructor Profile

UX — General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000 — Logon access to an HP-UX system

Software Requirements

UX11 — HP-UX release 11.0

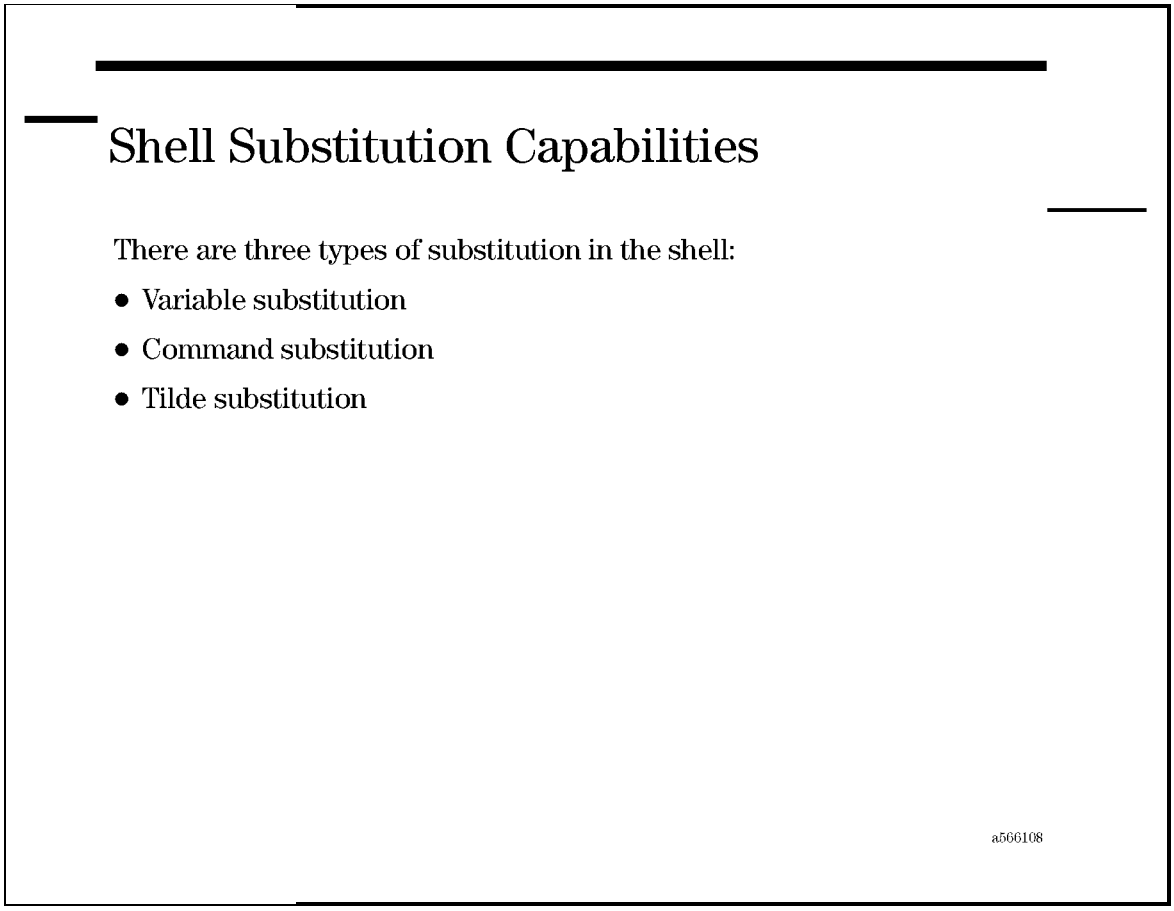
Material List

P/N B2355-90033(T)	*HP-UX Reference Manual* , one per terminal

Lab Instructions

setup1	Create user logon of **`user1`**, **`user2`**, ... **`user`***n*, where *n* is the number of students in the class. Set up one user per student.

8-1. SLIDE: Shell Substitution Capabilities



Student Notes

There are three types of substitution in the shell:

- Variable substitution
- Command substitution
- Tilde substitution

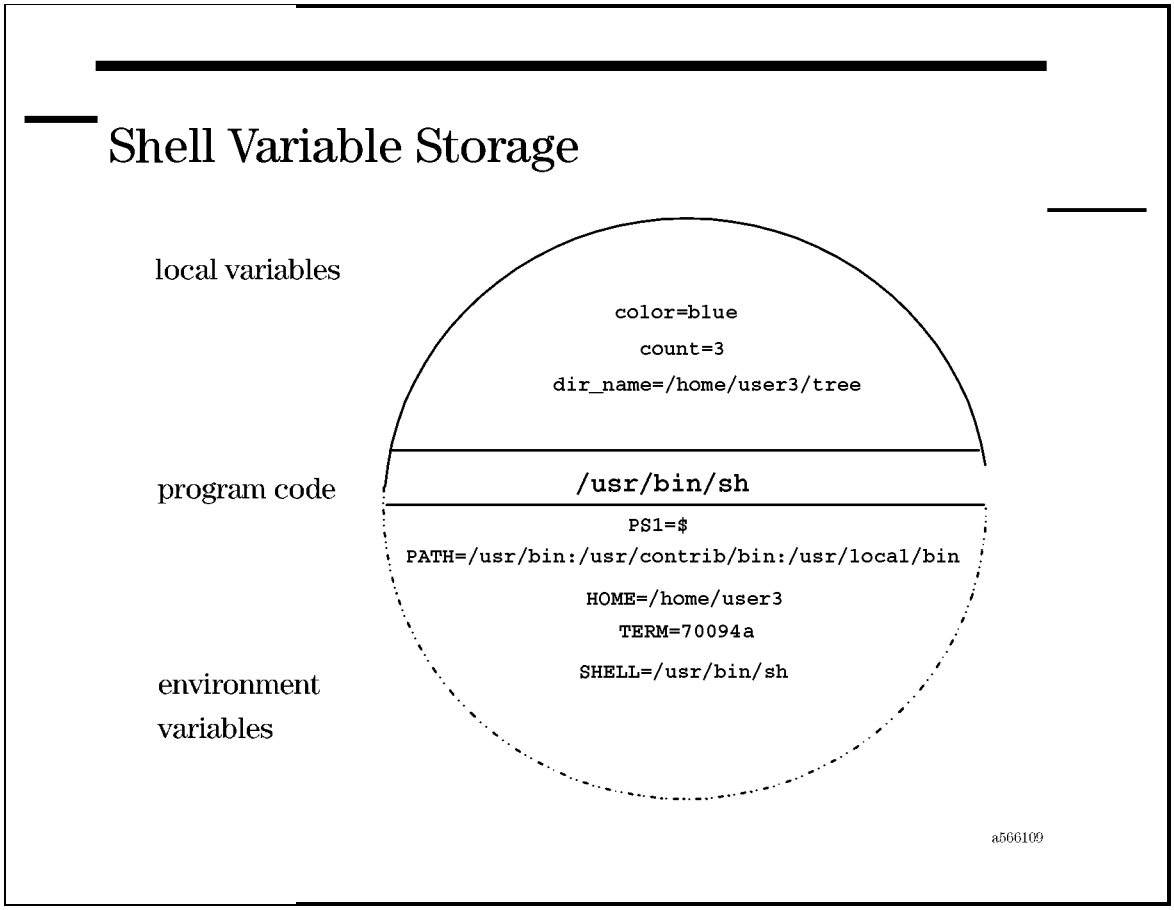
Substitution methods are used to speed up command-line typing and execution.

8-1. SLIDE: Shell Substitution Capabilities Instructor Notes

Teaching Tips

This is just an introductory slide.

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 will be 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, variables that are moved into the environment can be accessed by subprocesses.

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. The values of these variables can be changed to customize characteristics of your terminal session.

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
```

8-2. SLIDE: Shell Variable Storage — Instructor Notes

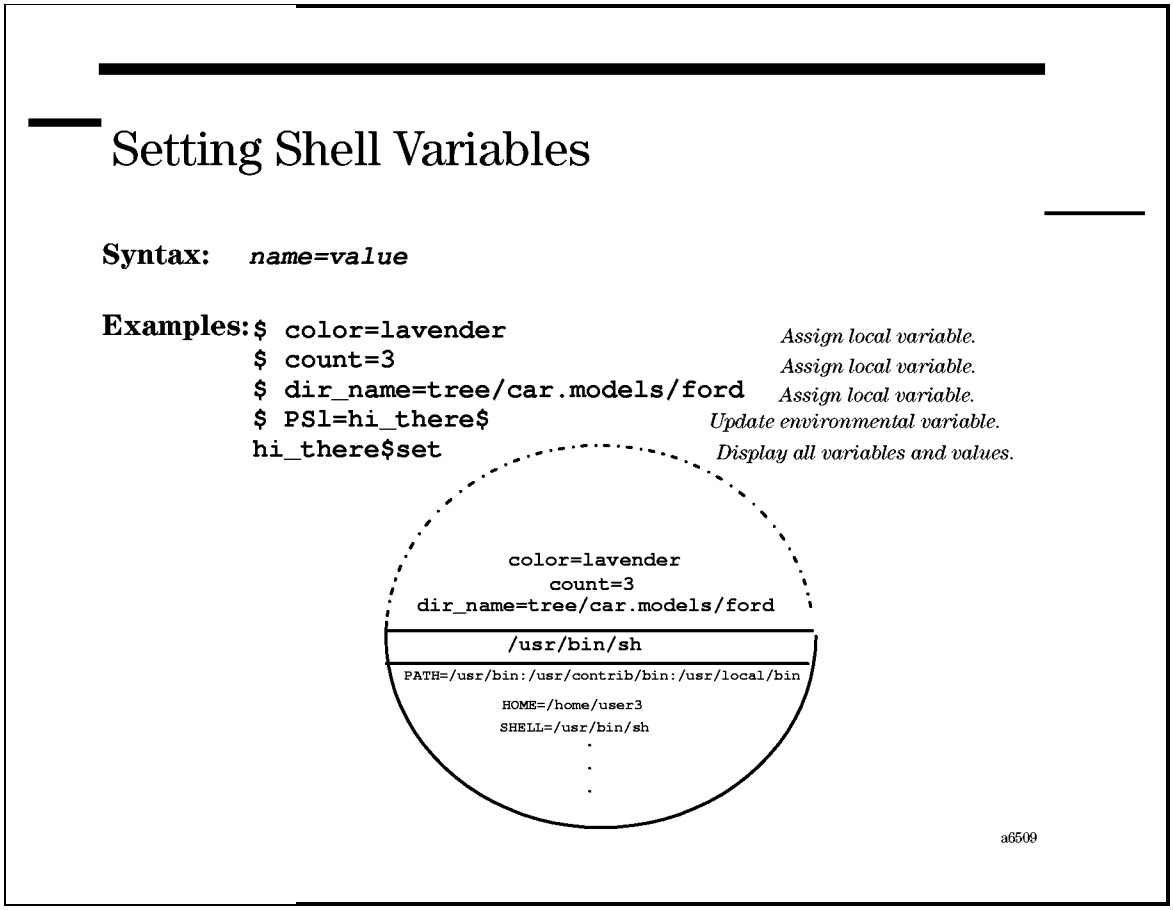
Key Points

- Two areas of memory available to hold shell variables.
- Local variables:
  - — Private variables accessible only by current shell (process)
  - — Can be moved (exported) to the environment
  - — Naming convention — lowercase characters
- Environment variables:
  - — Accessible by subprocesses
  - — Naming convention — uppercase characters
  - — Many are defined during your login process that set up session characteristics
- Because *PS1* is defined through a variable, it can be customized to any character string that you wish.
  - — Note that the prompt variables *PS1* and *PS2* are usually stored locally, *not* in the environment. This makes it easier to differentiate between a parent and child shell process. If a parent shell prompt is customized and a child shell is spawned, the child shell will not inherit the prompt since *PS1* is not in the environment but will use the default prompt instead.

Teaching Tips

Be sure to review the environment variables that are presented in the student notes. These variables are important in defining different characteristics of a terminal session.

8-3. SLIDE: Setting Shell Variables

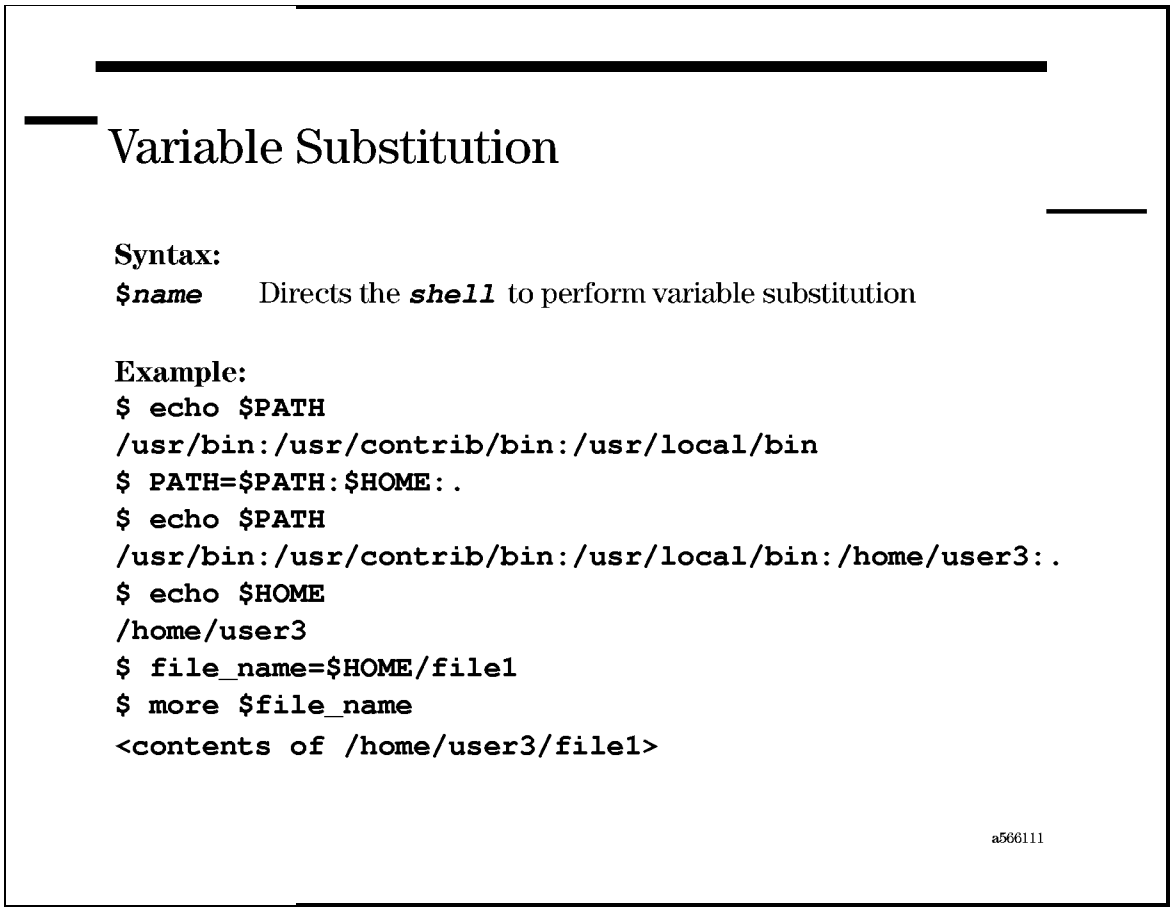


Student Notes

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 will replace the old value in the environment.

8-3. SLIDE: Setting Shell Variables Instructor Notes

8-4. SLIDE: Variable Substitution



Student Notes

Each variable that is defined will have an associated value. When a variable name is immediately preceded by a dollar sign (**$**), the shell will replace the parameter with the value of the variable. This procedure is known as **variable substitution** and 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. This provides 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:.
$ 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       looks for variables file, file1
$ echo ${file}and$file1
thisandthat
```

The curly braces can be used to delimit the variable name from the surrounding text.

8-4. SLIDE: Variable Substitution Instructor Notes

Key Points

- Variable substitution is done *by the shell*, not by the command.
- Variable substitution occurs prior to the execution of the command.
- All dereferenced variables are passed onto the command.
- Variables can be used to assign values to other variables.
- A variable's value can be used to define itself.
- **`echo $name`** is an effective method of displaying a variable value.
- Point out that the PATH variable uses a : delimiter.

Teaching Tips

Review the examples on the slides.

Teaching Questions

What happens when the following are entered?

```
$ HOME
```

The shell tries to execute a command **`HOME`**. The shell will always try to execute the first parameter on the command line as a program. *You will get an error.*

```
$ $HOME
```

The shell tries to execute a command **`/home/user3`**, since *$HOME* is assigned to the value */home/user3* . *You will get an error.*

8-4. 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
```

a566112

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.

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
```

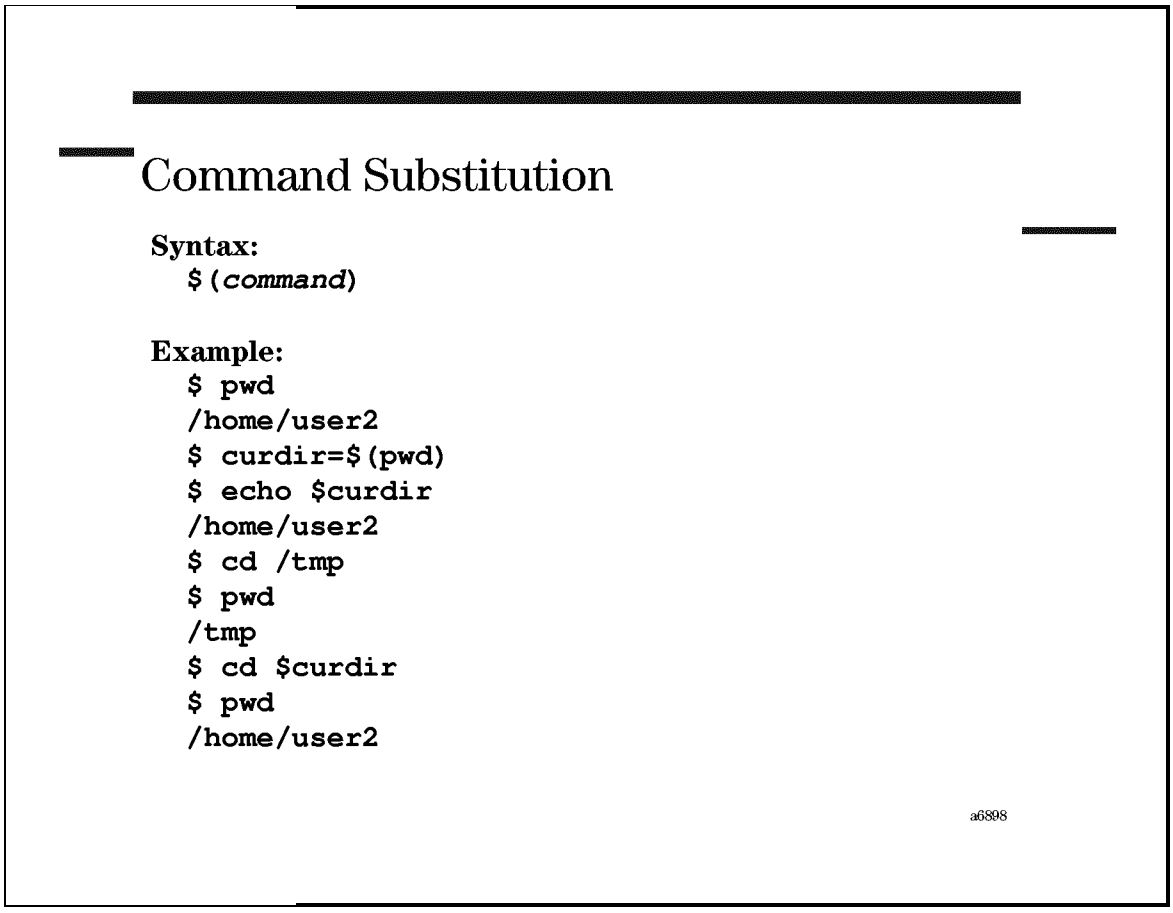
8-4. SLIDE: Variable Substitution (Continued) — Instructor Notes

Teaching Tips

Discuss the advantage of the second assignment of *dir_name* to an absolute path over the first assignment of *dir_name* to a relative path.

The **-C** option with **ls** will cause the **ls** output to be displayed in multicolumn format.

8-5. 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 may 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`
```

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 POSIX, Korn, and Bourne Shell.

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 **banner**ed. In the second example when *date* is used with command substitution, the shell will first execute the **date** command, and replace 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**.

8-5. SLIDE: Command Substitution — Instructor Notes

Key Points

- Command substitution is executed before the leading command on the command line.
- It is useful to assign the output of a command to a variable.

Teaching Tips

Encourage the students to type in the commands on the slide and in the student notes.

For POSIX and Korn Shell, there is another special command substitution for the **cat** command. Normally, you type

```
$ echo "$(cat file)"
```

and the contents of file are displayed. This quicker and shorter form produces the same results:

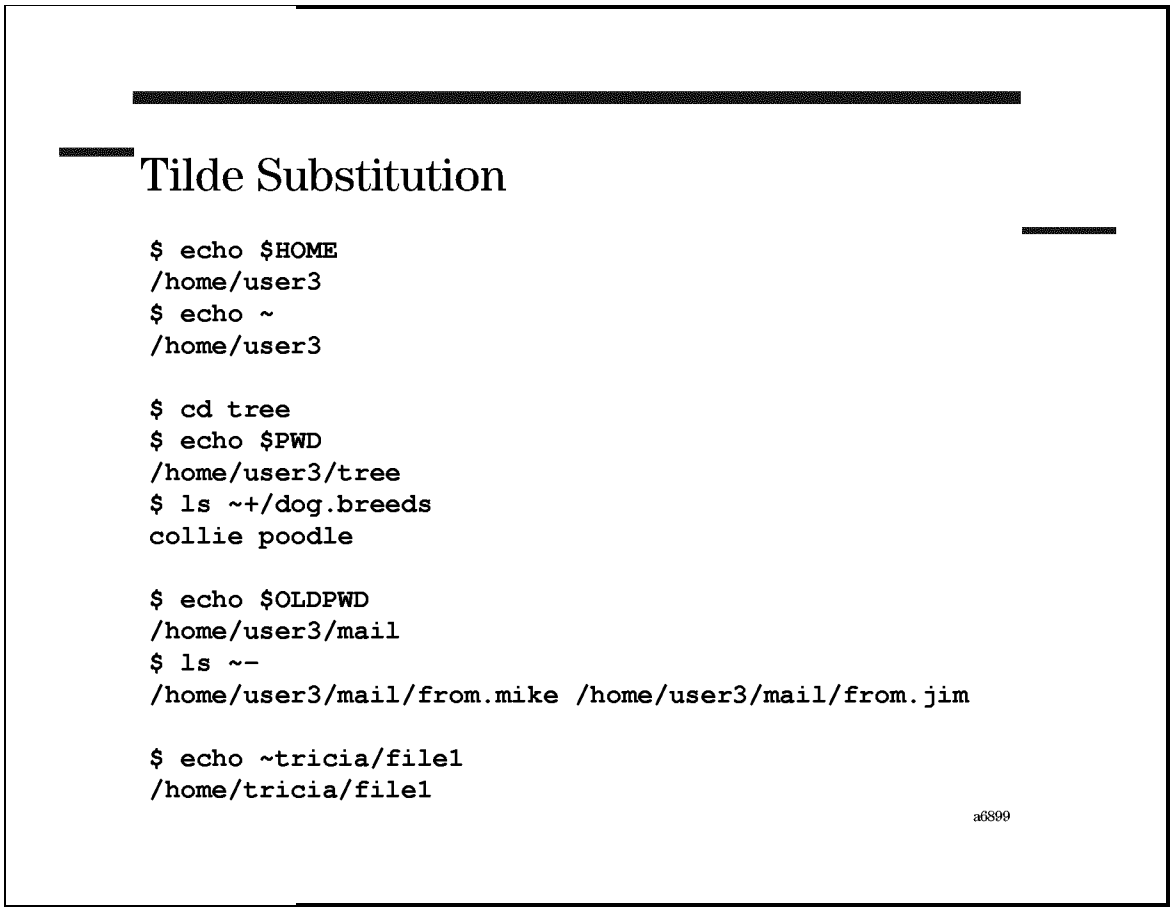
```
$ echo "$(< file)"
```

OR

```
$ echo "`< file`"
```

NOTE: The double quotes are *not* a necessary part of the syntax of the above commands, but they are necessary to quote the newline characters generated by the **cat** command. If the double quotes are omitted, the shell will interpret the newlines as white space and **cat** the contents of the file without breaks between lines.

8-6. 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 **-** 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.

Tildes can be put in aliases:

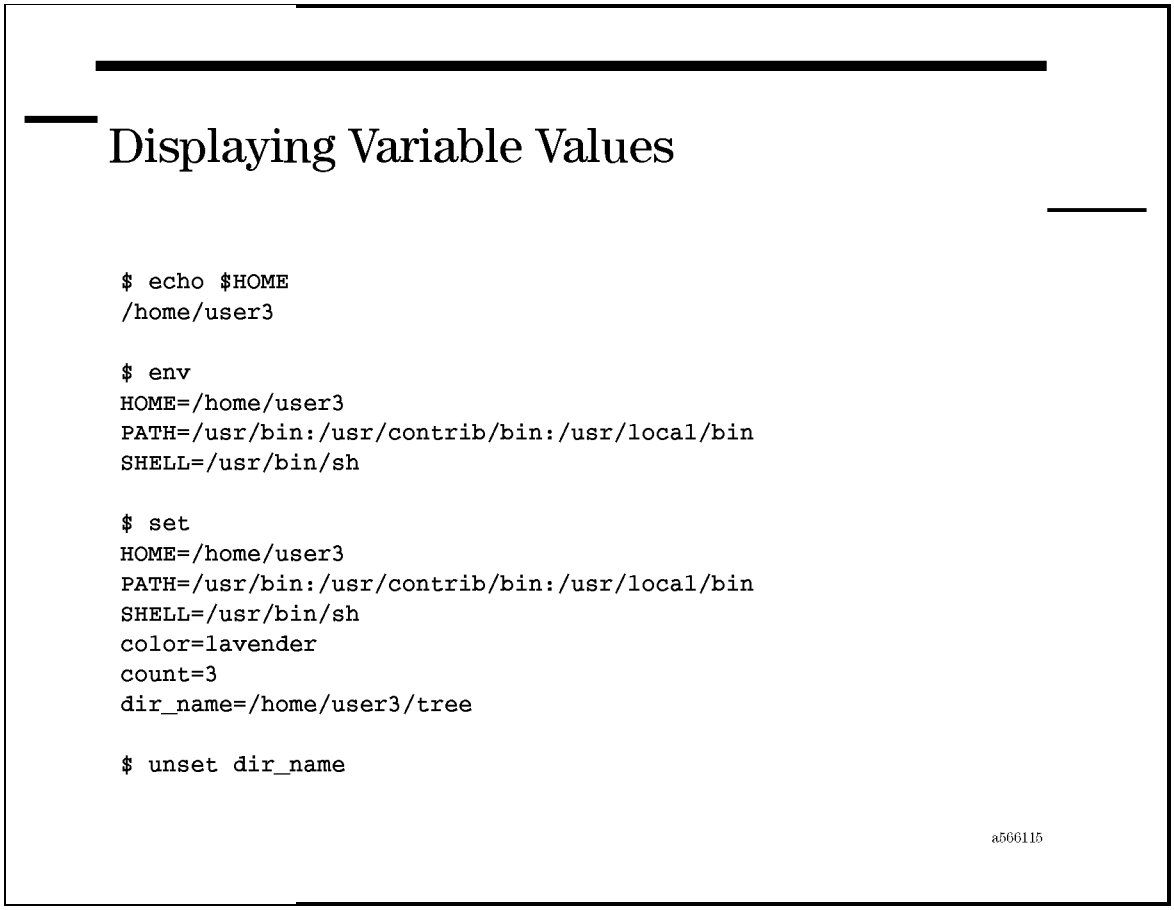
```
$ pwd
/home/user3
$ alias cdn='cd ~/bin'
$ cdn
$ pwd
/home/user3/bin
```

8-6. SLIDE: Tilde Substitution Instructor Notes

Key Points

- Probably the most common use of the tilde is to substitute for a user's home directory.

8-7. SLIDE: Displaying Variable Values



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**.

8-7. SLIDE: Displaying Variable Values Instructor Notes

Key Points

- Variable substitution is the easiest way to display the value of a single variable — local or environment.
- The **env** command displays all of the currently defined variables in the environment only.
- The **set** command displays all of the currently defined variables (local and environment).
- The **unset** command deletes the specified variable.

Note that **env** may be invoked with arguments. The syntax for the command is

```
env [-] [-i] [name=value] ... [command [arguments ...]]
```

env obtains the current environment, modifies it according to its arguments, then executes the command with the modified environment. Arguments of the form *name=value* are merged into the inherited environment before the command is executed. The **-i** option causes the inherited environment to be ignored completely so that the command is executed with exactly the environment specified by the arguments. The **-** option is obsolete and has the same effect as the -i option. See **env(1)** for more information.

The command

```
set -u
```

can be issued to cause an error to be generated when an undefined variable is encountered, as in

```
$ echo $fff
sh: fff: parameter not set
$
```

To just ignore variables that are not set, issue

```
set +u
```

Undefined variables will just be assigned to NULL:

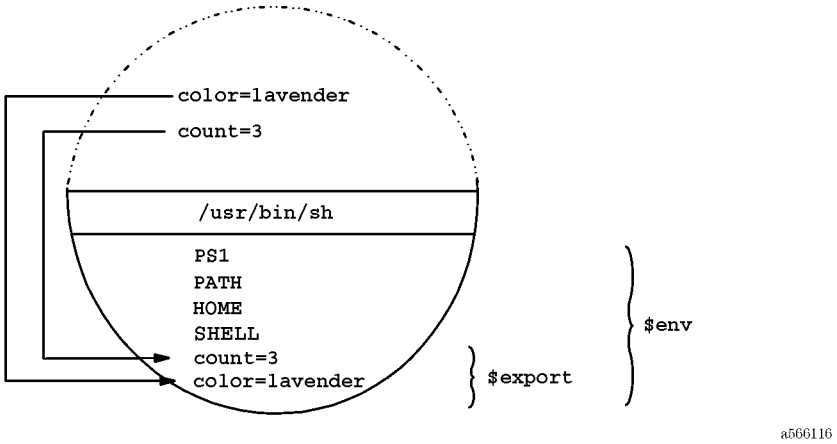
```
$ echo $fff
$
```

8-8. SLIDE: Transferring Local Variables to the Environment

Transferring Local Variables to the Environment

Syntax:

`export` ***`variable`*** Transfer ***`variable`*** to 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 **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.

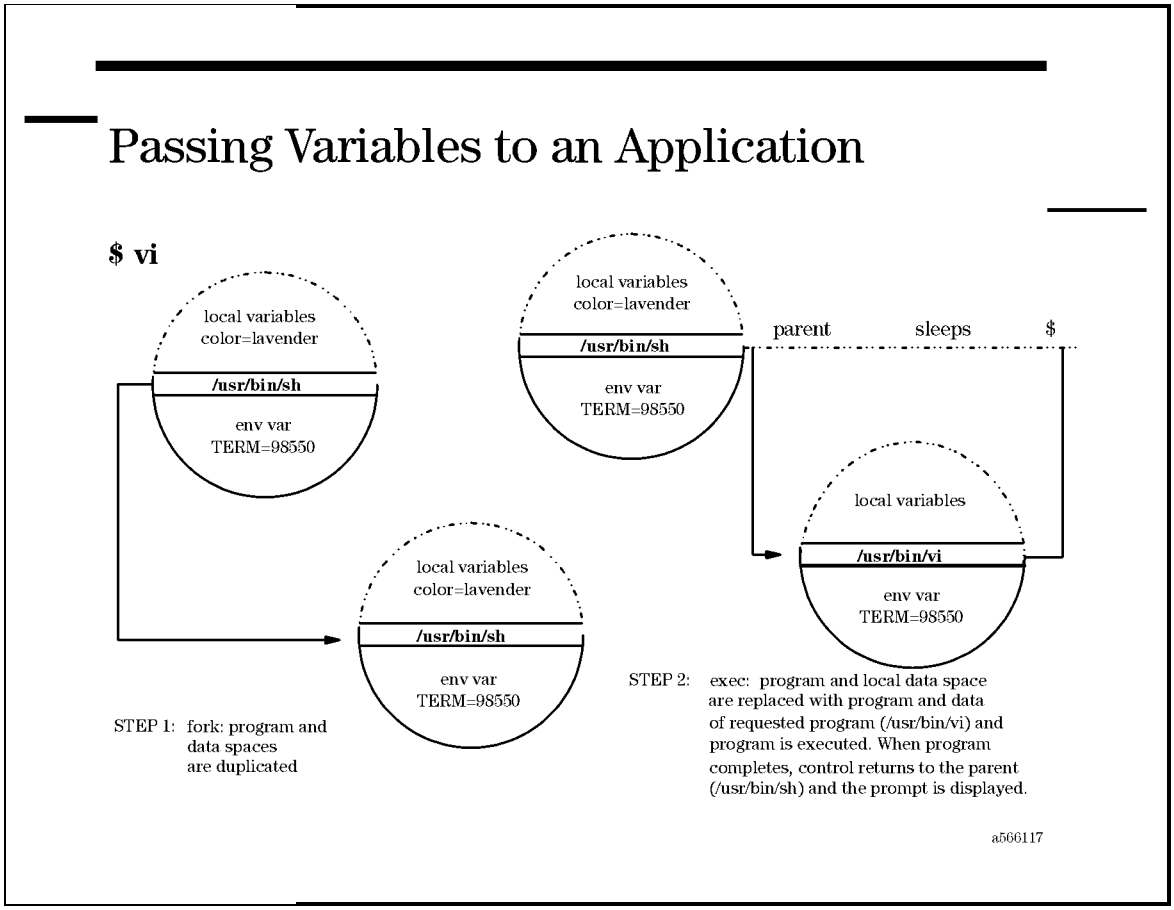
8-8. SLIDE: Transferring Local Variables to the Environment

Instructor Notes

Key Points

- Variables must be in the environment to be made available to child processes.
- The **`export`** *variable* command transfers a local variable to the environment.

8-9. 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 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.

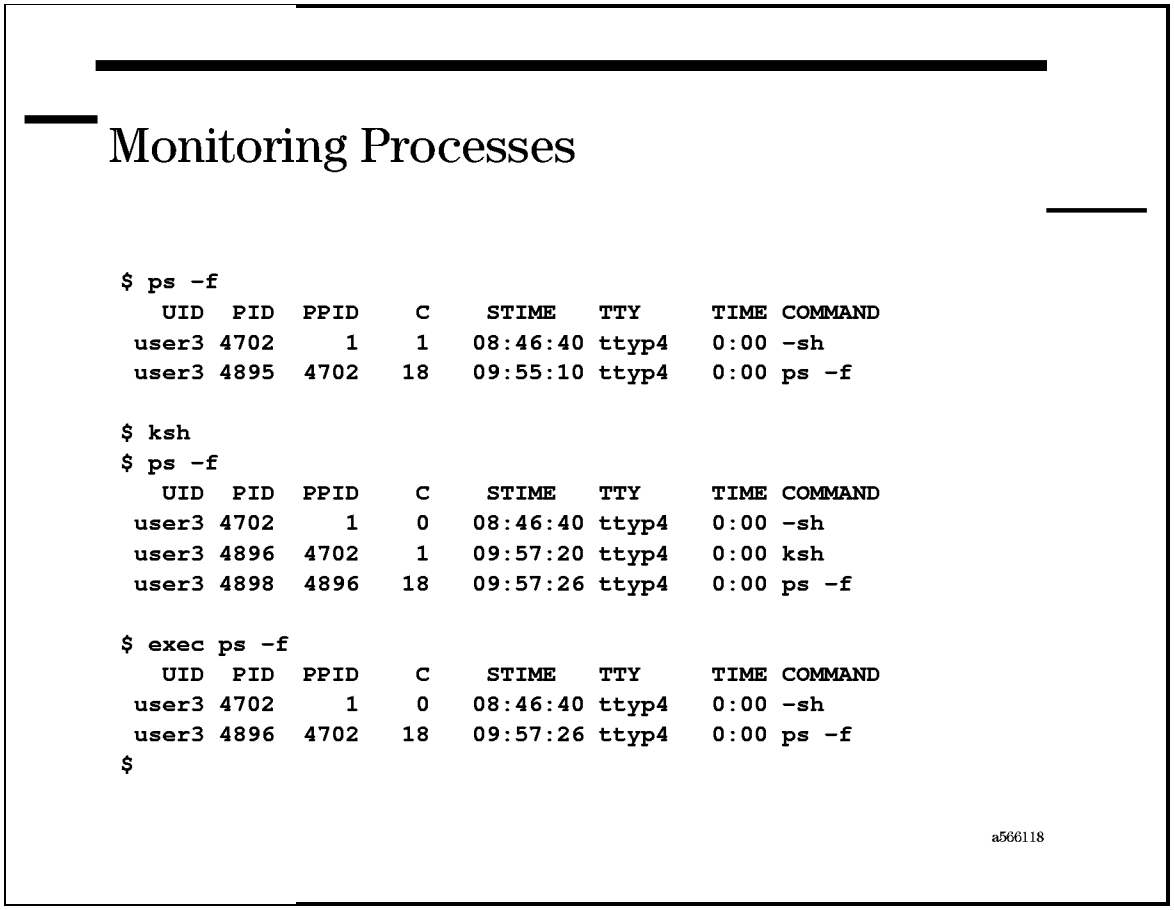
Therefore we see that one way of passing data to (child) processes is through the environment.

8-9. SLIDE: Passing Variables to an Application — Instructor Notes

Key Points

- Discuss the **`fork`** and **`exec`** commands.
- This slide illustrates why environment variables are propagated to child processes, whereas local variables are not.

8-10. SLIDE: Monitoring Processes



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 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.

8-10. SLIDE: Monitoring Processes — Instructor Notes

Key Points

- **ps** output
- PID
- PPID
- Point out the parentage of the **sh** and **ksh** and **ps** commands.

NOTE: If the fair share scheduler has been installed on your system, the following options are also available for the **ps** command:

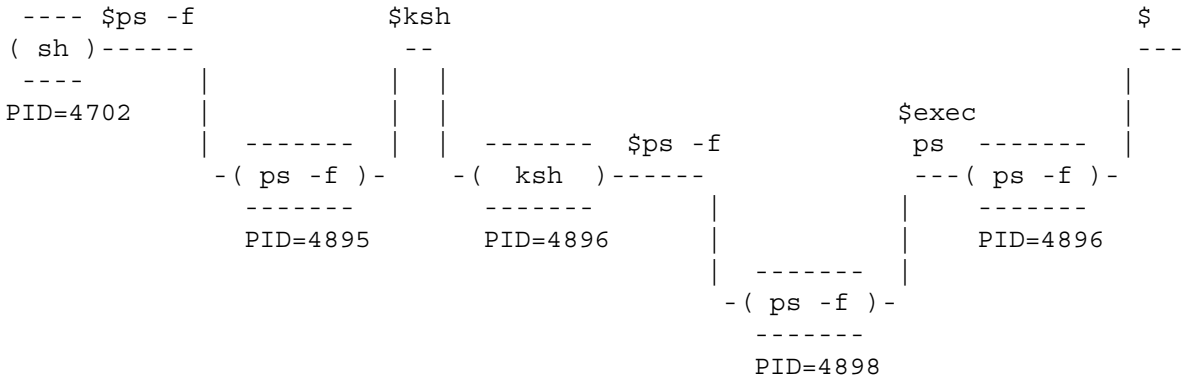
-F	Print the fair share group process association.
-G fglist	Restrict listing to data about processes whose fair share group ID numbers or fair share group names are given in fglist.

Additionally, the following column is added to the standard output:

FSID (f,l)	The fair share group ID of the process; the fair share group ID under the **-l** option, and the fair share group name under the **-f** option. If neither the **-l** option nor the **-f** option are specified when the **-F** option is specified, the fair share group name is printed.

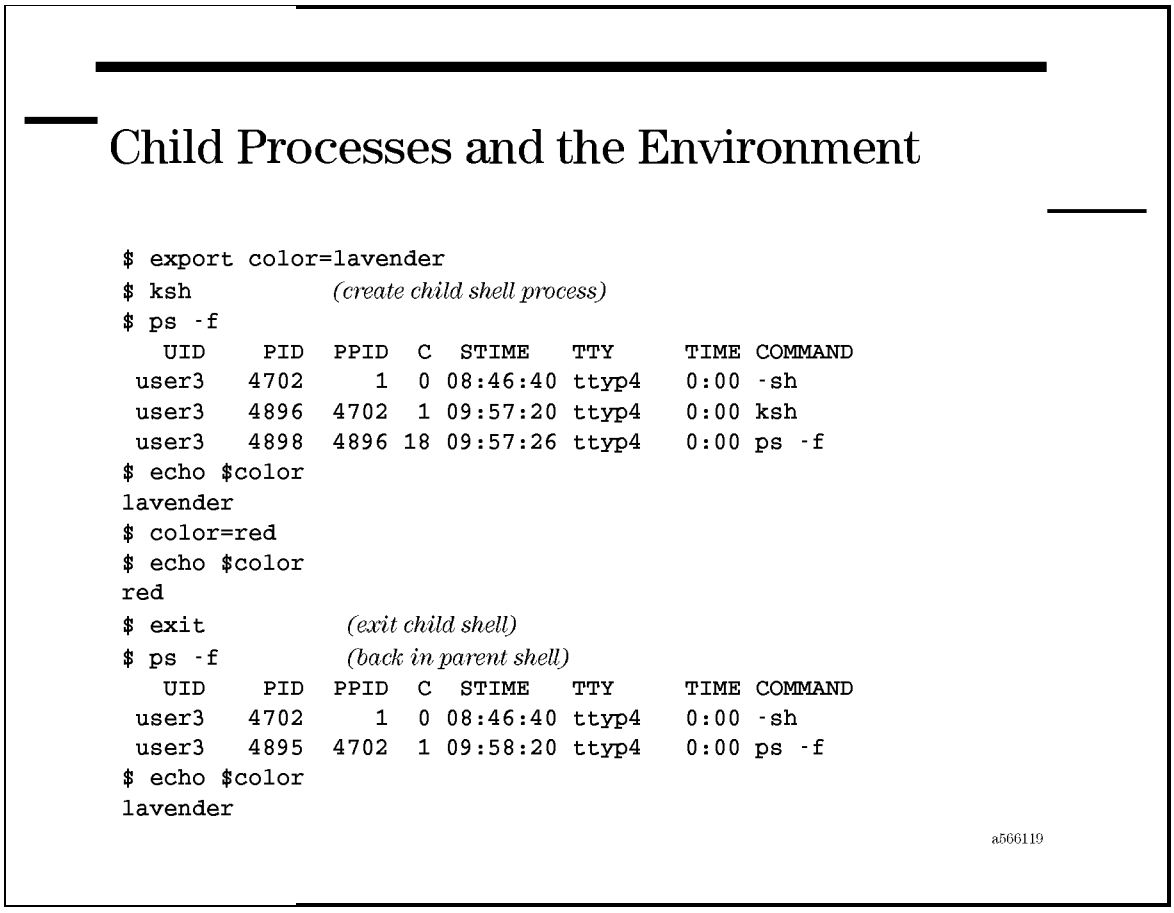
Teaching Tips

You may want to diagram the slide similarly:



Details of the **ps** command are presented in the module "Process Control."

8-11. SLIDE: Child Processes and the Environment



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 which follows confirms this. 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*.

8-11. SLIDE: Child Processes and the Environment — Instructor Notes

Teaching Questions

Have the students execute the commands presented on the slide to prove that a child process cannot alter the environment of its parent.

8-12. 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?

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?)

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 arguments?

8-12. LAB: The Shell Environment — Instructor Notes

Time: 45 minutes

Purpose

To become familiar with the shell as an interpreter of commands. This is achieved by studying shell variables and simple commands.

Notes to the Instructor

Introductory Exercises 1–8

Intermediate Exercises 9–11

The "Introductory" exercises are for students who need additional practice with the basic concepts of the module, such as substitution, assigning variable values, and the effect of the local versus the environment variables.

The "Intermediate" exercises also provide the student with practice assigning and referencing variables, but they are a little more complex. This section will request the student to modify his or her prompt, and *PATH* variables.

Solutions

1. Using command substitution, assign today's date to the variable *today*.

   Answer:

```
$ date
Fri Apr 2 11:57:21 EST 1993
$ today=$(date)
echo $today
Fri Apr 2 11:57:21 EST 1993
```

2. What is an easy way to list the contents of another user's home directory?

   Answer:

   If the other user's name was *mike*, you could get a listing of his home directory using:

```
$ ls ~mike
```

3. Set a shell variable named *MYNAME* equal to your first name. How do you see the contents of that variable?

Answer:

```
$ MYNAME=user3
$ echo $MYNAME
user3
```

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*?

Answer:

The *MYNAME* variable was set in the parent shell's local data area. When the child shell was spawned, it inherited only the parent's environment variables.

When the child shell is dead, the parent wakes up and remembers all that it knew. You can test this by typing

```
$ echo $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?

Answer:

```
$ export MYNAME

$ env
```

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?

Answer:

```
$ MYNAME=user2

$ env
```

MYNAME is still an environment variable in the child shell.

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?

Answer:

```
$ unset MYNAME
```

MYNAME will no longer exist in the child shell because the **unset** command removes it.

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?

Answer:

$ Ctrl + c

Return

The removal of the variable in the child shell does not have an effect on the variable in the parent shell. Therefore, *MYNAME* still exists in the environment of the parent shell. To verify this, you can display the environment variables in the parent shell.

```
$ env
```

9. Modify your shell prompt so that it displays: *good_day$*. What happens to your prompt when you log out and log back in?

Answer:

```
$ PS1=good_day$
good_day$
```

When you log out and log back in the prompt reverts to the default.

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?)

Answer:

```
$ PS1=$LOGNAME   or   $ PS1=$(whoami)
user3                 user3
```

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 arguments?

Answer:

```
$ dir=/usr/bin/ls
$ $dir
```

$dir will be substituted with **/usr/bin/ls** , and then execute the command **/usr/bin/ls** . Yes this will accept command line arguments. Try by executing:

```
$ $dir $HOME /tmp /var
```

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.

Overview of Module 9

Audience

general user — General system user

Product Family Type

open sys — Open systems environment

Abstract

This module is designed to introduce the student to file name generating characters. The student will learn to save typing by using file name expansion instead of complete file names on the command line.

Time

Lab — 30 minutes

Lecture — 30 minutes

Prerequisites

m51m — Shell Advanced Features

Instructor Profile

UX — General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000 — Logon access to an HP-UX system

Software Requirements

UX11 — HP-UX release 11.0

Material List

P/N B2355-90033(T) — *HP-UX Reference Manual* , one per terminal

Lab Instructions

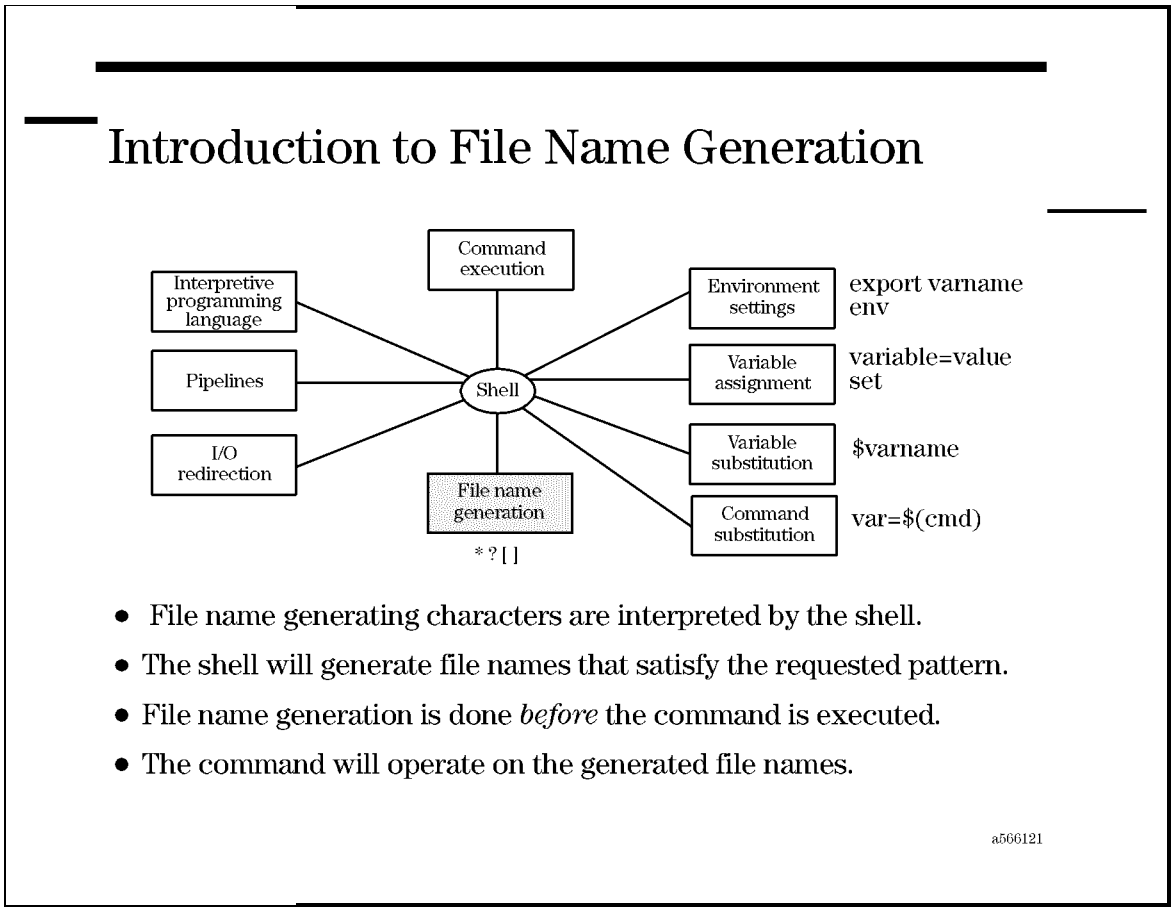
setup1 — Create user logon of **user1**, **user2**, ... **user***n*, where *n* is the number of students in the class. Set up one user per student.

copyfiles — Copy the lab files to the users' home directories.

Lab Files

```
-rw-r--r--   1 karenk  users    20 May 28 16:12 file.1
-rw-r--r--   1 karenk  users    15 May 28 16:12 file.2
-rw-r-----   1 karenk  users  3081 May 28 16:12 funfile
-rw-r--r--   1 karenk  users    61 May 28 16:12 herfile
-rw-r--r--   1 karenk  users   216 May 28 16:12 libtest.c
-rw-r--r--   1 karenk  users   182 May 28 16:12 math.c
-rw-r--r--   1 karenk  users   168 May 28 16:12 math.f
-rw-r--r--   1 karenk  users   189 May 28 16:12 math.p
-rw-r--r--   1 karenk  users    39 May 28 16:12 mod1.c
-rw-r--r--   1 karenk  users    39 May 28 16:12 mod2.c
-rw-r--r--   1 karenk  users    39 May 28 16:12 mod3.c
-rw-r--r--   1 karenk  users    37 May 28 16:12 myfile
-rw-r--r--   1 karenk  users    86 May 28 16:12 myprog.c
-rw-r--r--   1 karenk  users    97 May 28 16:12 myprog.f
-rw-r--r--   1 karenk  users   110 May 28 16:12 myprog.p
-rw-r--r--   1 karenk  users   128 May 28 16:12 part1.c
-rw-r--r--   1 karenk  users    63 May 28 16:12 part2.c
-rw-r--r--   1 karenk  users    23 May 28 16:12 ourfile
-rw-r--r--   1 root    other    77 May 28 16:12 root_file
-rw-r--r--   1 karenk  users   163 May 28 16:12 xdbtest.c
-rw-r--r--   1 karenk  users    29 May 28 16:12 yourfile

filegen:
total 0
-rw-r--r--   1 karenk  users     0 May 28 16:12 Abc
-rw-r--r--   1 karenk  users     0 May 28 16:12 Abcd
-rw-r--r--   1 karenk  users     0 May 28 16:12 abc
-rw-r--r--   1 karenk  users     0 May 28 16:12 abcdemf
-rw-r--r--   1 karenk  users     0 May 28 16:12 e35f
-rw-r--r--   1 karenk  users     0 May 28 16:12 efg
-rw-r--r--   1 karenk  users     0 May 28 16:12 fe3f
-rw-r--r--   1 karenk  users     0 May 28 16:12 fe3fg
```

9-1. SLIDE: Introduction to File Name Generation



Student Notes

The shell provides a time-saving 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.

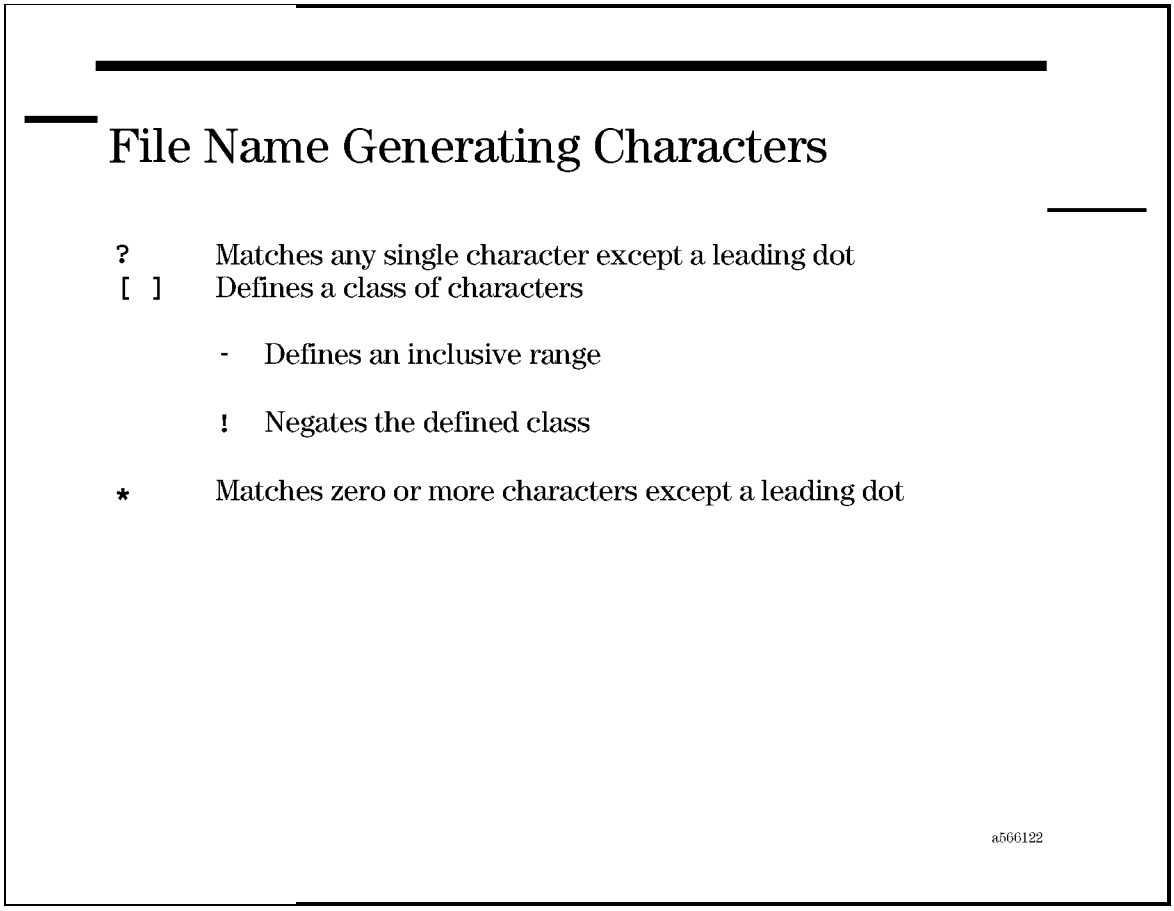
The file name generating feature is useful because most applications will define naming conventions for their files. Once you know what the naming conventions are, you can use file name expansion to access just the files whose names contain the desired pattern. For example, source code for C programs conventionally ends in **`.c`**, and word processors may use **`.doc`** as an extension for document files.

9-1. SLIDE: Introduction to File Name Generation — Instructor Notes

Key Points

- File name generating characters are interpreted by the shell.
- File names are generated *before* the command is executed.
- File name generating characters are *not* wildcards!
- Wildcards are interpreted by the command. This is useful when looking for files that follow a naming convention.

9-2. SLIDE: File Name Generating Characters



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).

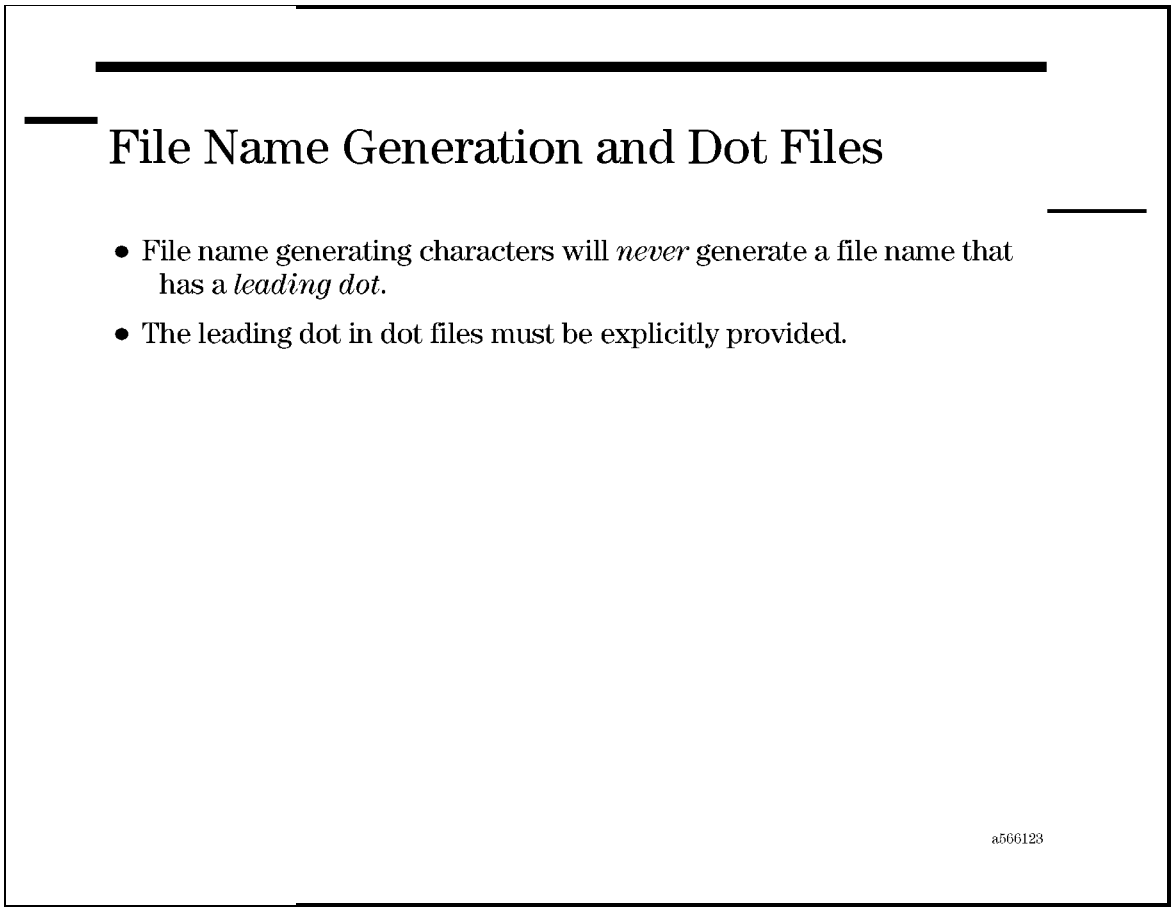
We will see each of these characters in detail.

9-2. SLIDE: File Name Generating Characters — Instructor Notes

Teaching Tip

- Introduce the special characters used. We will define each of them in detail over the next several slides.

9-3. SLIDE: File Name Generation and Dot Files



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.

9-3. SLIDE: File Name Generation and Dot Files — Instructor Notes

Teaching Tips

- Point out that the shell will *not* generate a leading dot. It must be matched explicitly. We will see some examples of this.

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 ?
```

a566121

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.*

9-4. SLIDE: File Name Generation — ? Instructor Notes

Key Points

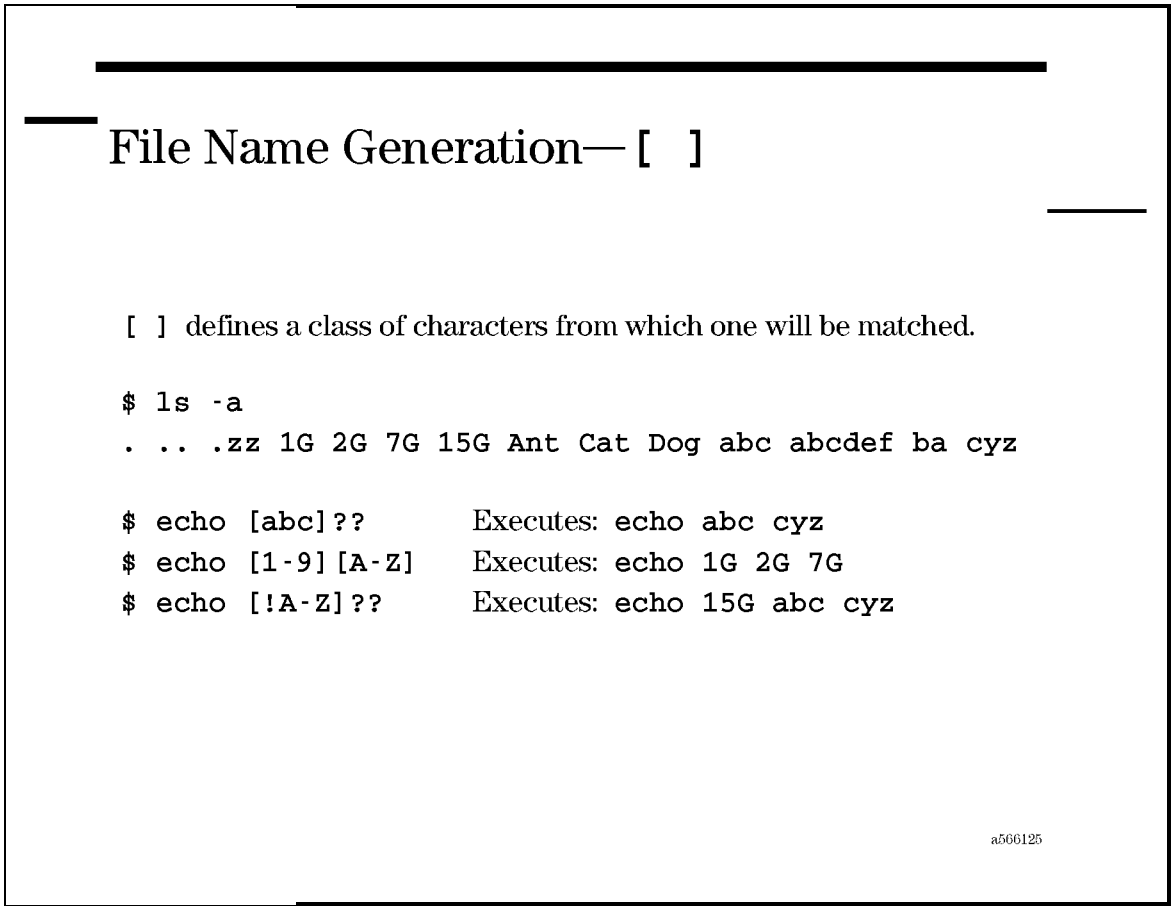
- File name generation is performed *before* the command is executed.
- **?** matches any *single* character; it represents one character position.
- Even though examples use the **echo** command, file name generating is more commonly used with the file manipulation commands.
- Be careful when using file name generation characters with the **rm** command — when a file is removed it is GONE!
- Examples on the slide:

$ echo ???	**???** generates file names that contain three characters.
$ echo abc?	**abc?** generates file names that begin with **abc** and end with any character.
$ echo ??a??	**??a??** generates file names that contain five characters with an **a** in the middle.
$ echo .??	**.??** generates file names that begin with a dot (**.**) followed by two characters.
$ echo ?	**?** generates file names that contain 1 character.

Teaching Tip

- Point out that if no match is found, the pattern is not replaced by a list of file names, and the pattern will be passed as the argument to the command.

9-5. SLIDE: File Name Generation — [ ]



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.

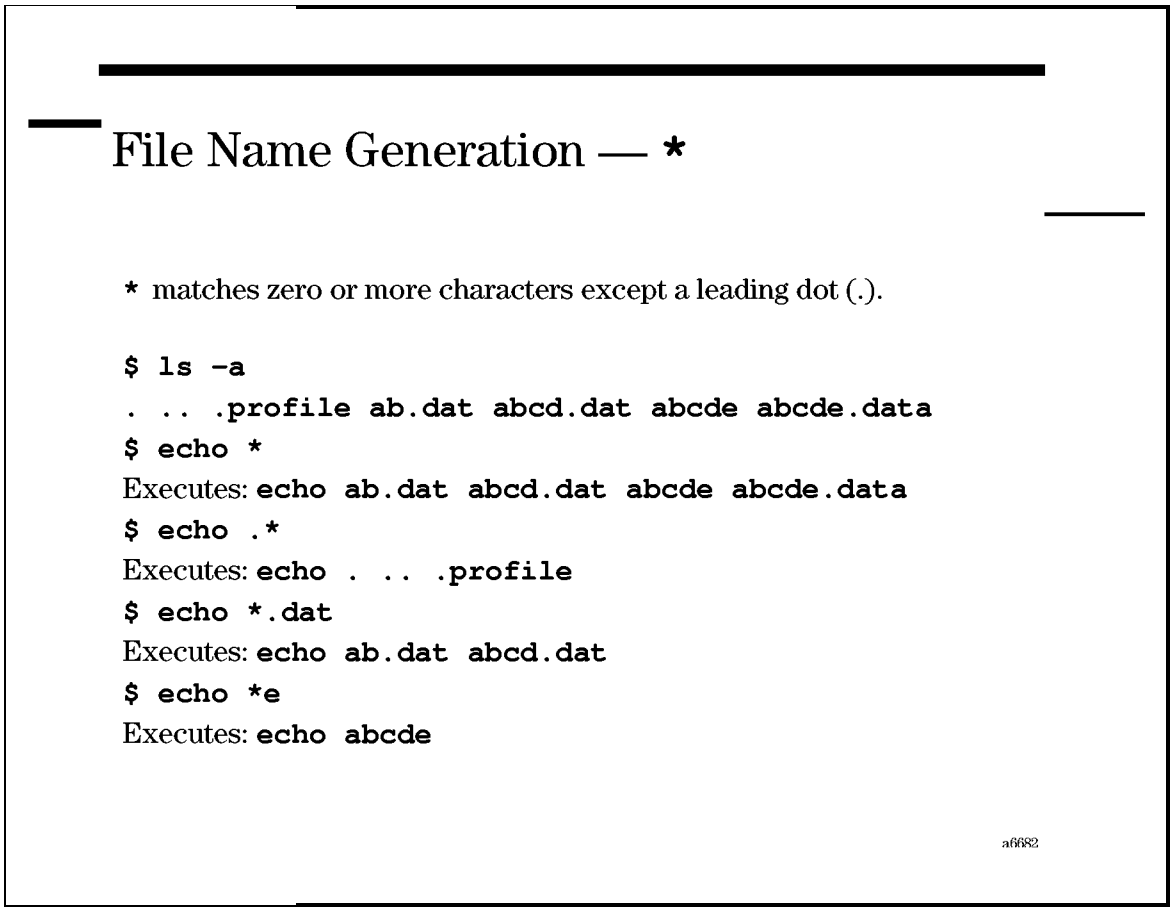
9-5. SLIDE: File Name Generation — [ ] Instructor Notes

Key Points

- Even if a dot is defined in the class, it will not be matched as the first character in a file name.
- There is *not* a character class range [**a-Z**]; you must use [**a-zA-Z**].
- Examples on the slide:

$ echo [abc]??	**[abc]??** generates three-character file names that begin with an **a, b**, or **c**.
$ echo [1-9][A-Z]	**[1-9][A-Z]** generates two-character file names that begin with a digit **1, 2, ...** or **9** and end with an uppercase letter.
$ echo [!A-Z]??	**[!A-Z]??** generates three-character file names that *do not* begin with an uppercase letter.

9-6. SLIDE: File Name Generation — *



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.

9-6. SLIDE: File Name Generation — * Instructor Notes

Key Points

- Examples on the slide:

`$ echo *`	`*` generates all file names *except dot files*.
`$ echo .*`	`.*` generates all dot files.
`$ echo *.dat`	`.dat` generates all file names ending in `.dat` *except dot files*.
`$ echo *e`	`*e` generates all file names ending in `e` *except dot files*.

Teaching Tips

- Again, show that an asterisk (`*`) will *not* match a leading dot (`.`). At this point, you should mention again that file name generating characters can be used with *any* UNIX system command, not just `echo`, as it is the shell that is expanding them and not the command itself.
- You might want to discuss the differences and implications of executing the command `ls .*` as opposed to `echo .*`. Because (`.`) and (`..`) are generated, `ls` will not only list the dot files, but also the complete contents of the current directory *and* the parent directory.

Teaching Questions

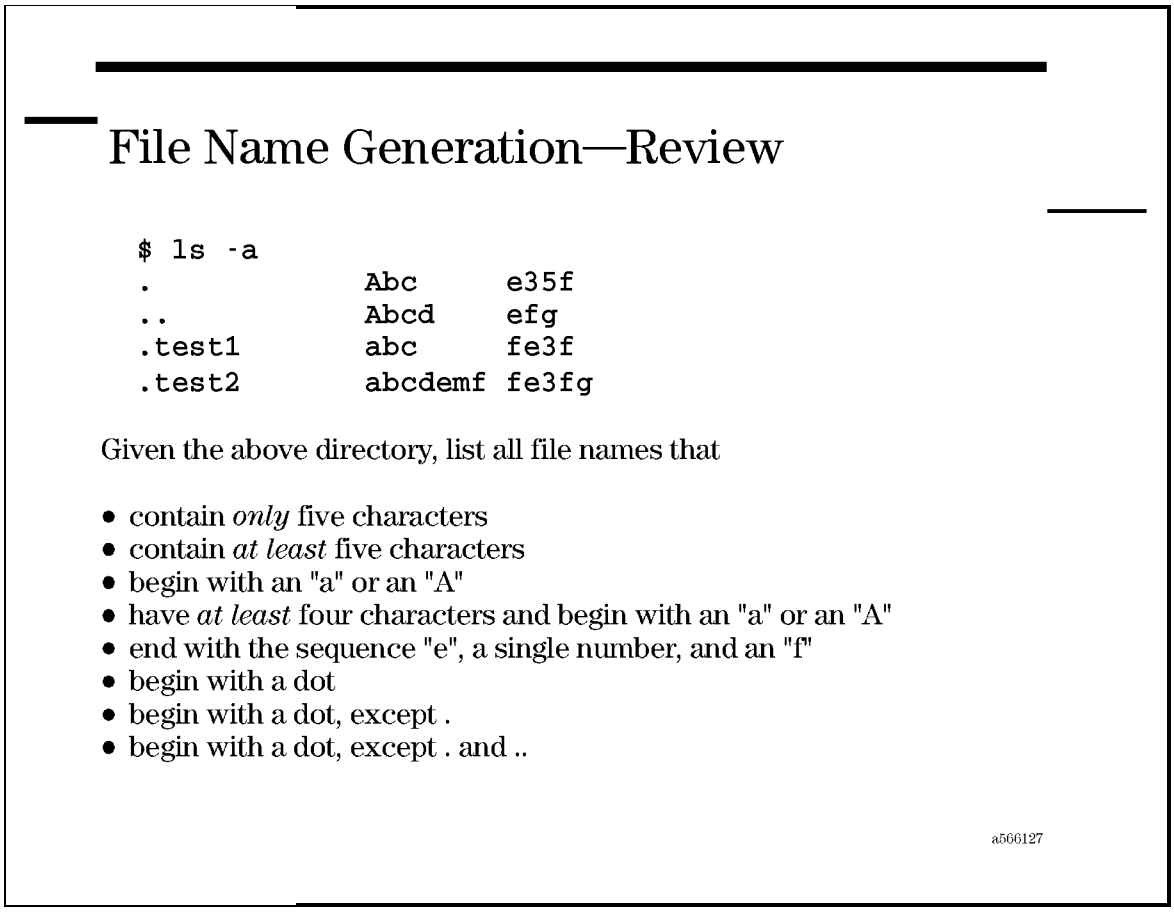
What would be the implication of executing `rm .*`?

Answer:
All dot files would be removed.

What would be the implication of executing `rm -r .*`?

Answer:
All files and directories under the current directory would be removed. Your current directory cannot be removed.

9-7. SLIDE: File Name Generation — Review



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.

9-7. SLIDE: File Name Generation — Review Instructor Notes

Teaching Tips

Get the class to describe the patterns and predict the output of the **echo** commands. The correct answers are as follows:

How do you list all file names that:

Contain only five characters?	**ls ?????**
Contain at least five characters?	**ls ?????*** or **ls *?????**
Begin with an **a** or an **A**?	**ls [aA]***
Have at least four characters and begin with an **a** or an **A**?	**ls [aA]???***
End with the sequence **e**, a single number, and an **f**?	**ls *e[0-9]f**
Begin with a dot?	**ls .*** *what happens?*
Begin with a dot, except **.** ?	**ls .?***
Begin with a dot, except **.** and **..**?	**ls .[!.]***

NOTE: Discuss the effect of:

$ rm .*

$ rm -r .*

Be sure to discuss the teaching question. This really demonstrates the order of operations; the shell generates the file names and then executes the command.

Teaching Question

What's the difference between the following two commands?

```
$ echo .*

$ ls .*
```

The first command will execute the following:

```
$ echo . .. .test1 .test2          Echoes out the arguments
```

```
. .. .test1 .test2
```

The second command will execute

```
$ ls . .. .test1 .test2          Lists contents of directories . and ..
```

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*`** ?

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.)

9-8. LAB: File Name Generation — Instructor Notes

Time: 30 minutes

Purpose

To practice using file name generation characters on the command line to save typing.

Notes to the Instructor

Exercise 4 illustrates a common usage of the file name generating characters, to manipulate all files whose names follow a common pattern.

Your students should complete Exercises 1-6. Exercises 7 and 8 present a puzzle for students to solve, and can be considered optional.

Solutions

1. Change to your *HOME* directory, then type the command **`ls *`** and explain the output.

   Answer:

   Remember that the **`ls`** command can act two different ways. If given a file name as an argument, **`ls`** displays the file's name. If given a directory name, **`ls`** lists the contents of that directory. Thus when given a list of file and directory names, such as that generated by the asterisk (**`*`**), **`ls`** will list the names of all files under the current directory and list the contents of all immediate subdirectories.

2. If the command **`echo ???XX`** produces the output **`???XX`**, what does it mean?

   Answer:

   There are no files in the current working directory that match the pattern **`???XX`**. Therefore the file name generation characters are taken literally, and the echo command echoes them out.

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`**.

Answer:

a. **ls *.c**
libtest.c math.c mod1.c mod2.c mod3.c myprog.c part1.c part2.c xdbtest.c

b. **ls mod*.c** or **ls mod[0-9].c** or **ls mod?.c**
mod1.c mod2.c mod3.c

c. **ls *file***
file.1 file.2 herfile myfile my_root_file ourfile root_file yourfile filegen/: Abc Abcd abc abcdemf e35f efg fe3f fe3fg

d. **ls *.[cfp]**
libtest.c math.c math.f math.p mod1.c mod2.c mod3.c myprog.c myprog.f myprog.p xdbtest.c part1.c part2.c

4. Create a directory called **c_source**. Move all of your **.c** files to the **c_source** directory using the file name generating characters.

Answer:

```
$ mkdir c_source
$ mv *.c c_source
```

5. Create a directory called **dir_1** under your *HOME* directory. What happens when you issue the command: **cd dir*** ?

Answer:

```
$ cd dir*                          Expands to: cd dir_1
```

You will effectively change to the *dir_1* directory.

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.

Answer:

```
$ cd                               Expands to: cd dir_1 dir_2 dir_3 dir_4
$ mkdir dir_2 dir_3 dir_4

$ cd dir*
```

This is not a legal usage for the **cd** command and will fail.

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.

Answer:

```
$ touch AXX
$ touch A.XX
$ touch B.XX
$ touch XX.AB
$ touch XX.A
$ echo ?XX
AXX
$ echo ?.XX
A.XX B.XX
$ echo *XX
A.XX AXX B.XX
$ echo XX.??
XX.AB
$ echo XX.*
XX.A XX.AB
```

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.)

Answer:

```
$ rm *XX*
```

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.

Overview of Module 10

Audience

general user — General system user

Product Family Type

open sys — Open systems environment

Abstract

The purpose of this module is to teach the student how to escape the special meaning of many of the characters on the command line.

Time

Lab — 45 minutes

Lecture — 60 minutes

Prerequisites

m51m — Shell Advanced Features

Instructor Profile

UX — General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000 — Logon access to an HP-UX system

Software Requirements

UX11 — HP-UX release 11.0

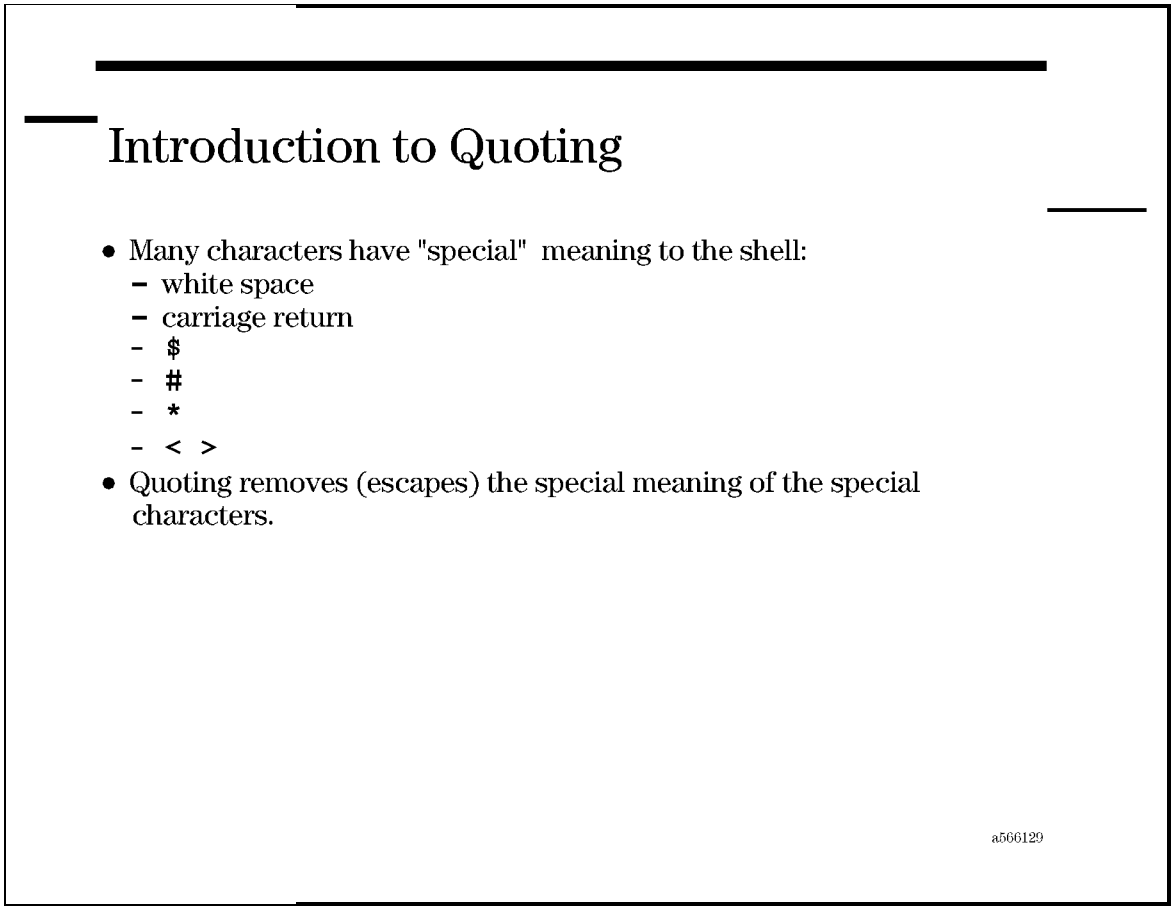
Material List

P/N B2355-90033(T)	*HP-UX Reference Manual* , one per terminal

Lab Instructions

setup1	Create user logon of **`user1`**, **`user2`**, ... **`user`***n*, where *n* is the number of students in the class. Set up one user per student.

10-1. SLIDE: Introduction to Quoting



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.

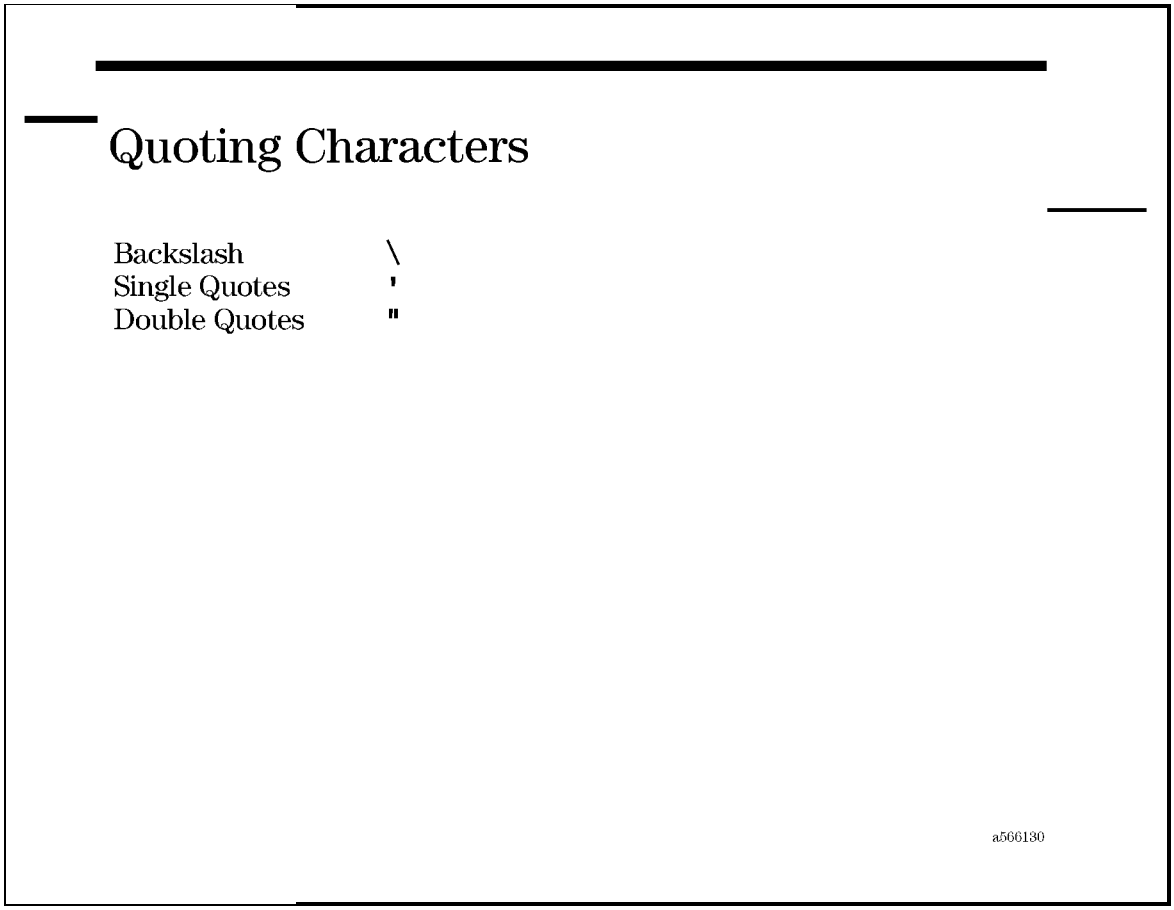
There are situations in which you do not want the shell to interpret the special meaning associated with these characters. You require just the literal character. Therefore, the UNIX system must provide a mechanism to escape or remove the special meaning of a designated character. This mechanism is known as **quoting**.

10-1. SLIDE: Introduction to Quoting — Instructor Notes

Teaching Tips

Point out that there are several special characters in the shell. Ask the students for some examples that they have seen. Stress that because there are times when one would like the special characters to stand for themselves, we need a mechanism for doing this.

10-2. SLIDE: Quoting Characters



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 **"**.

10-2. SLIDE: Quoting Characters — Instructor Notes

Teaching Tips

The definition of the quoting mechanisms are given here in order to be complete. Examples are on the next three slides.

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
```

a566131

Student Notes

The backslash always removes the special meaning of the next character. There are no exceptions.

10-3. SLIDE: Quoting — \ Instructor Notes

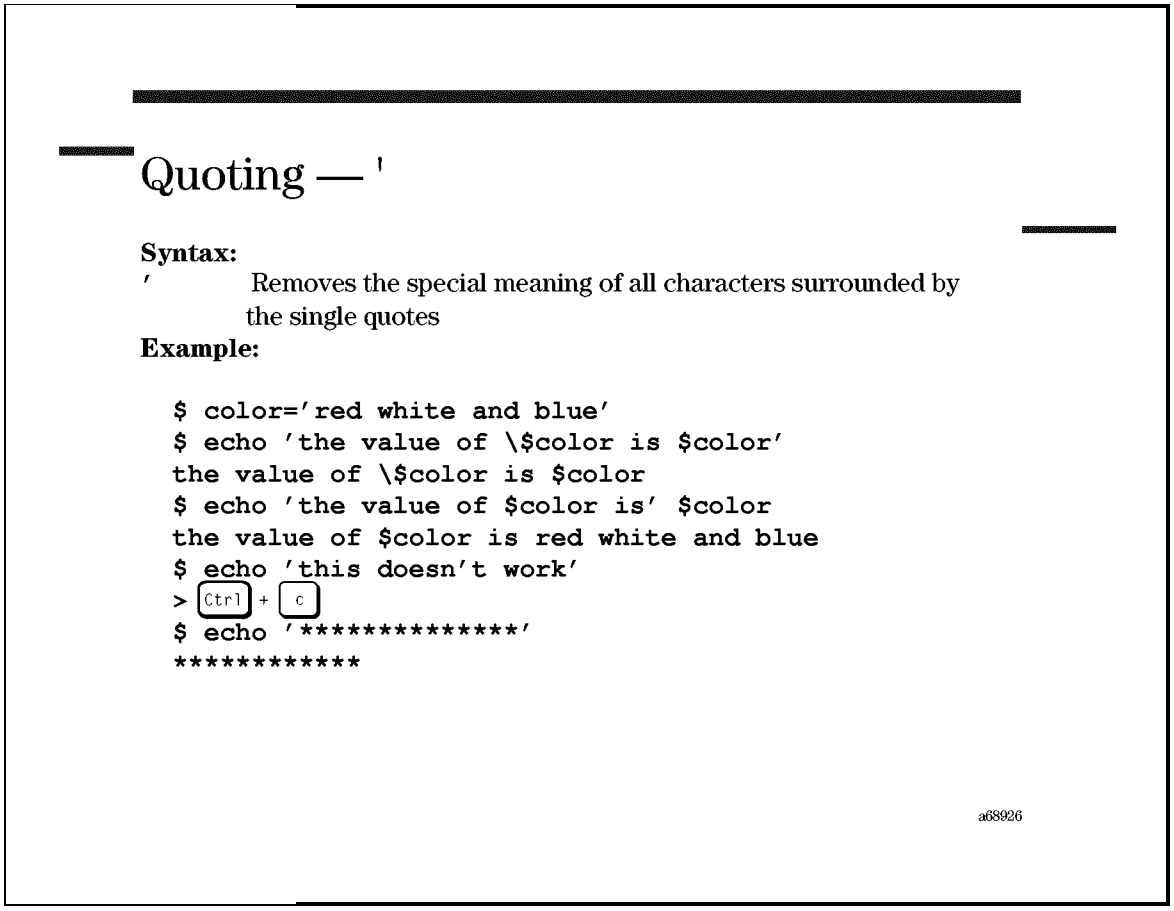
Key Points

- The backslash removes the special meaning of the very next character. There are no exceptions.
- The last example on the slide is escaping the Return. Since the Return is escaped, the shell does not see the command terminator. Therefore, it will display the secondary prompt, signifying that the user must enter more information to complete the command. This is helpful when entering extremely long command lines.

Another Example

```
$ echo 3 \> 2

3 > 2
```

10-4. SLIDE: Quoting — ’



Student Notes

Single quotes will remove the special meaning of *all* of the special characters enclosed by the single quotes.

10-4. SLIDE: Quoting — ’ Instructor Notes

Key Points

- Single quotes will remove the special meaning of *all* of the special characters enclosed by the single quotes.
- The single quote is required to close the quoted string.
- You *do not* have to describe what the ***** does at this point. You should just let the students know that *it is* a special character to the shell, and if it is not escaped, unexpected results will probably occur.

Teaching Tips

Note that the single quote (**’**) is not the same as the grave quote (**‘**). The grave quote is the old (Bourne shell) method used for *command substitution*, as in

```
$ pwd
/home/user3
$ curdir=‘pwd‘
$ echo $curdir
/home/user3
```

The above syntax is valid in the POSIX/Korn shell as well; however, the new POSIX/Korn shell method uses **$(cmd)**, as in

```
$ pwd
/home/user3
$ curdir=$(pwd)
$ echo $curdir
/home/user3
```

This concept is discussed in the module *Shell Advanced Features* .

Teaching Question

- How do you get *this doesn’t work* to print out?

```
echo this doesn\’t work
  OR
echo "this doesn’t work"
```

10-5. SLIDE: Quoting — "

Quoting — "

" 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, \, ', "
```

a566133

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: 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.

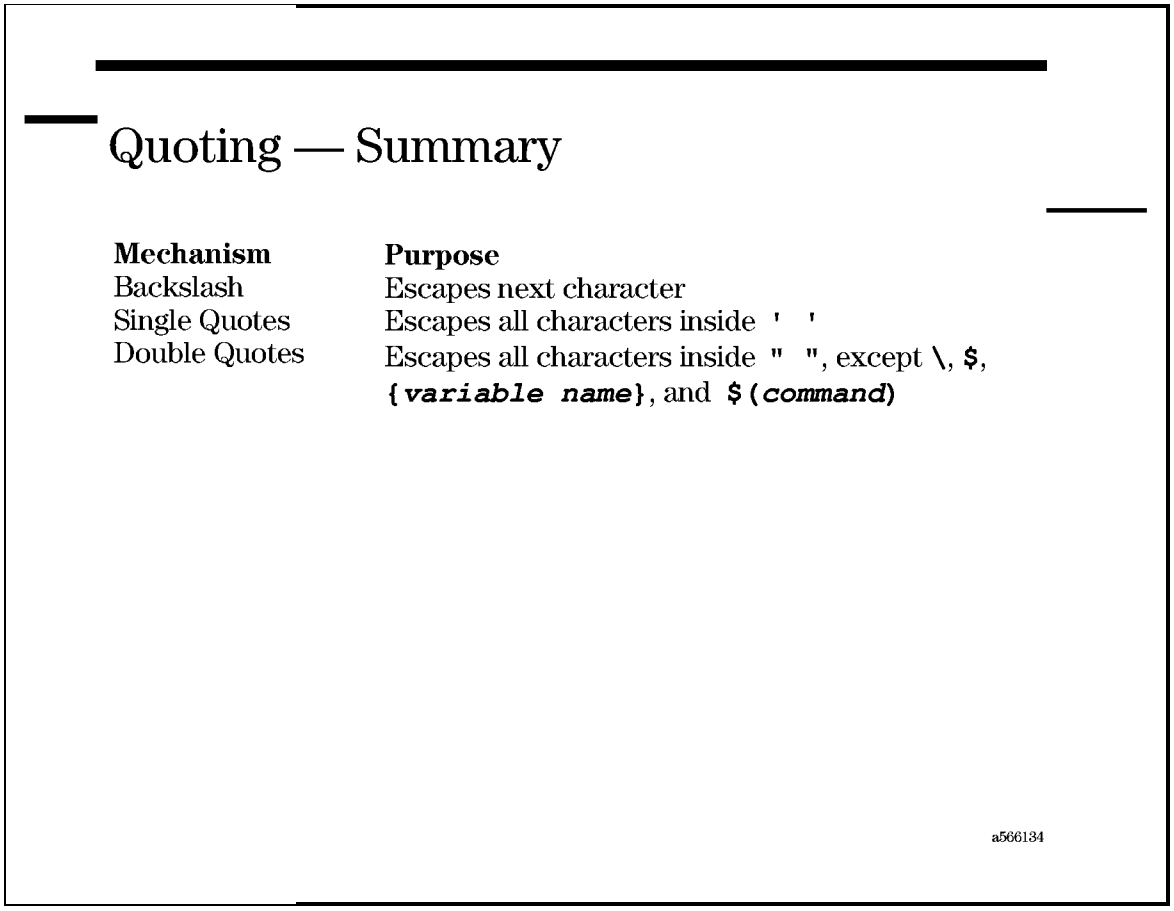
NOTE: All quoting mechanisms can be used in a single command line.

10-5. SLIDE: Quoting — " Instructor Notes

Key Points

- This is often confusing for students, but present the motivation for maintaining the special meaning of `$` and `` ` ``, and that should help in your presentation. The examples on the slide should assist you in this presentation.

10-6. SLIDE: Quoting — Summary



Student Notes

The slide shows a summary of the quoting characters and their actions.

10-6. SLIDE: Quoting — Summary Instructor Notes

Teaching Tips

Point out the summary and review it.

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?

4. Assign to your prompt the string: *Way to go YOUR_USER_NAME $*

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?

10-7. LAB: Quoting — Instructor Notes

Time: 45 minutes

Purpose

To practice using the quoting mechanisms to escape the meaning of special characters.

Solutions

1. Type an echo command that will produce the following output:

```
$1 million dollars   ...   and that's a bargain !
```

Answer:

```
$ echo "\$1 million dollars   ...   and that's a bargain !"
$ echo '$1 million dollars   ...   and that'\''s a bargain !
```

There are several options to echo this string out, since the echo command uses blank spaces as delimiters between the strings. You really only need to escape the **$** and the **'**.

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.

Answer:

```
$ long_string="\$1 million dollars   ...   and that's a bargain !"
$ echo "$long_string"
```

This has fewer options because you must quote all of the blank spaces for the variable assignment to succeed. Note what happens when you don't quote **$long_string**.

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?

Answer:

```
$ banner good day                    2 command line arguments
GOOD
DAY
$ banner "good day"                  1 command line argument
```

```
GOOD DAY
```

The quotation marks escape the space as an argument delimiter, so the second command will see only one command line argument.

4. Assign to your prompt the string: *Way to go YOUR_USER_NAME $*

Answer:

```
$ PS1="Way to go $LOGNAME $ "
Way to go user3 $
```

5. How would you display the following message?

```
Exercises #1, #2, and #3 are now complete.
```

Answer:

```
echo Exercises \#1, \#2, and \#3 are now complete.
```

The **#** symbol precedes comments, and must therefore be escaped.

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"
```

Answer:

```
$ echo '$abc'
$abc
```

The single quotes (**'**) do *not* allow variable substitution to occur, so the literal string *$abc* will be echoed back to your terminal.

```
$ echo "$abc"

sh: abc: parameter not set.
```

When the **$** appears within the double quotes (**"**), the shell will try to de-reference the variable. Since the variable does not hold a value, the shell will generate an error message when it tries to evaluate the variable value.

Note: The POSIX/Korn Shell will generate error messages when referencing a variable that has not been defined and the **set -u** option is set. You can enter **set -o** to view the options configured for your shell. If **-u** is not set, no error will be generated, and the variable value will be substituted with NULL. You can disable this option with: **set +u**. See the man page **sh-posix(1)** for more details.

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?

Answer:

```
$ touch "White Space"
$ touch "  "
$ touch "   "
```

When you do an **ls** command, the file names with the leading blanks will appear at the beginning of your directory listing. One way to determine how many blanks are in the file names is to make the files executable and then execute an **ls -F** and observe how many blanks occur before the asterisk, you can also try an **ls -b** command.

Notice that with the quoting mechanism you can create a file name that contains *any* special character. You will always have to use the escape characters though whenever you want to reference the file name on your command line.

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`**.

Overview of Module 11

Audience

general user — General system user

Product Family Type

open sys — Open systems environment

Abstract

Input and output redirection is an important feature of the UNIX system. This module shows 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.

Time

Lab — 30 minutes

Lecture — 30 minutes

Prerequisites

m1305m — Shell Basics

Instructor Profile

UX — General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000 — Logon access to an HP-UX system

Software Requirements

UX11 — HP-UX release 11.0

Material List

P/N B2355-90033(T)	*HP-UX Reference Manual* , one per terminal

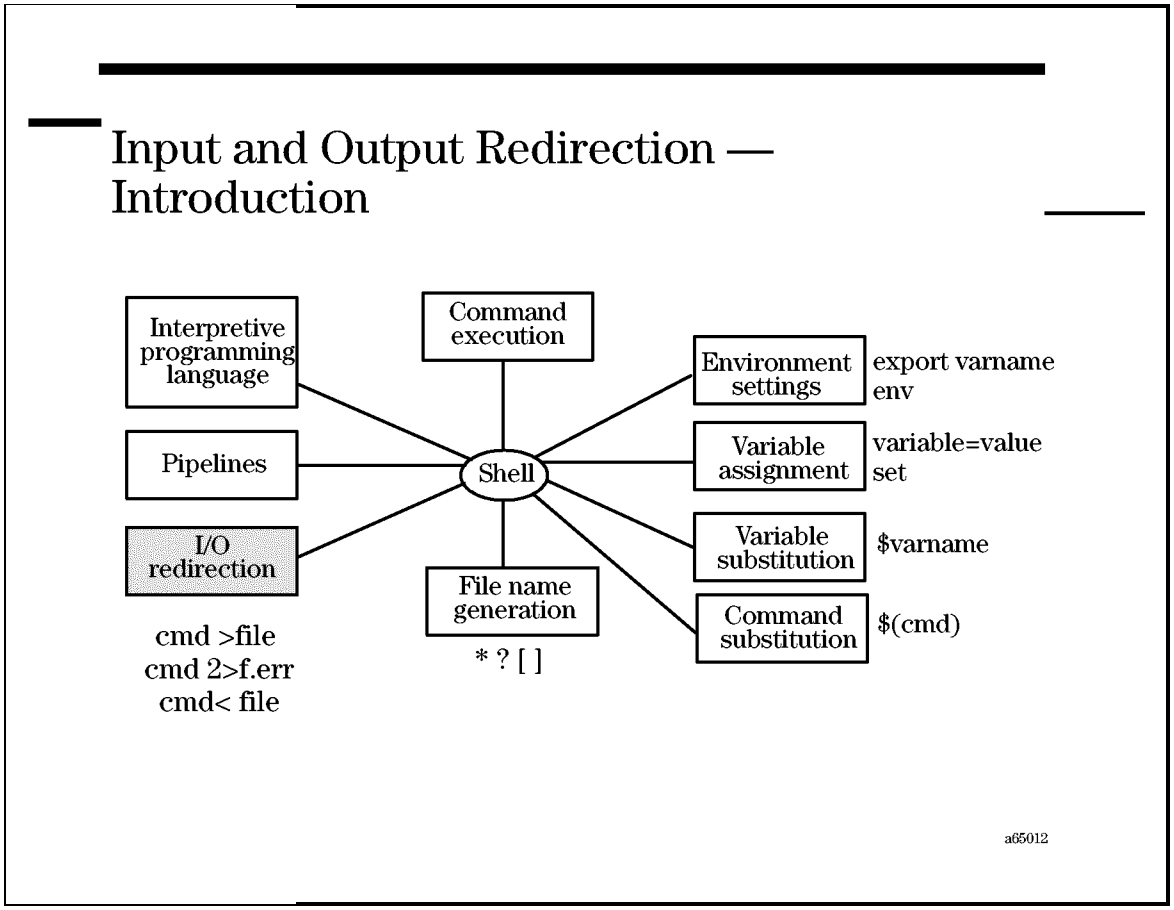
Lab Instructions

setup1	Create user logon of **user1**, **user2**, ... **user***n*, where *n* is the number of students in the class. Set up one user per student.
copyfiles	Copy the lab files to the users' home directories.

Lab Files

```
-rw-r--r-- 1 karenk users   61 May 28 16:12 names
```

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.

11-1. SLIDE: Input and Output Redirection — Introduction Instructor Notes

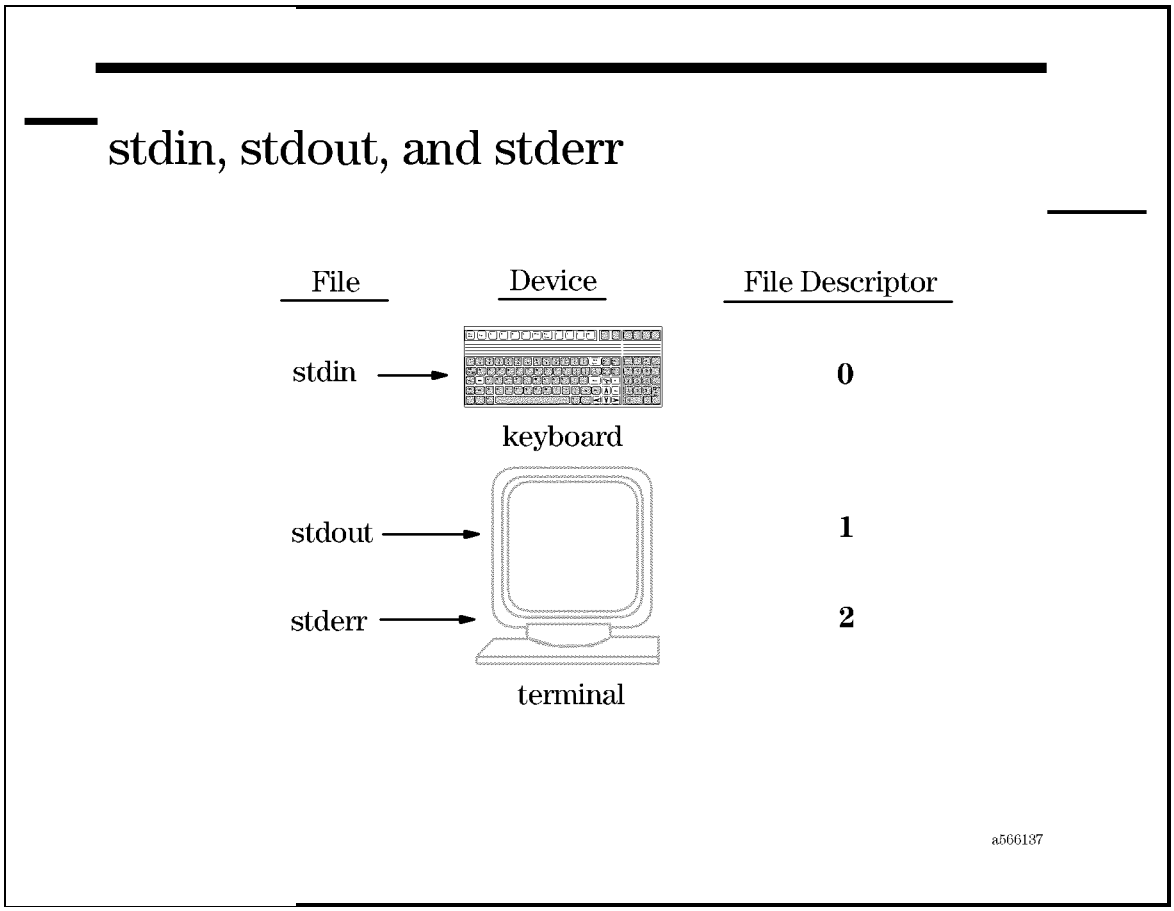
Key Points

- Everything in the UNIX system is a file.
- Commands generate output to the terminal file.
- Commands accept input from the keyboard file.
- Input and output redirection allow you to replace these default devices with text files.
- Output redirection is useful for logging command output, or saving output for future processing (filters).
- Input redirection is useful because you can create a file using an editor and send that file into the command instead of typing it interactively.

Teaching Tips

This slide is just meant as an introduction to the chapter but should also provide some motivation on why input and output redirection are very powerful capabilities.

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.

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.

11-2. SLIDE: stdin, stdout, and stderr — Instructor Notes

Teaching Tips

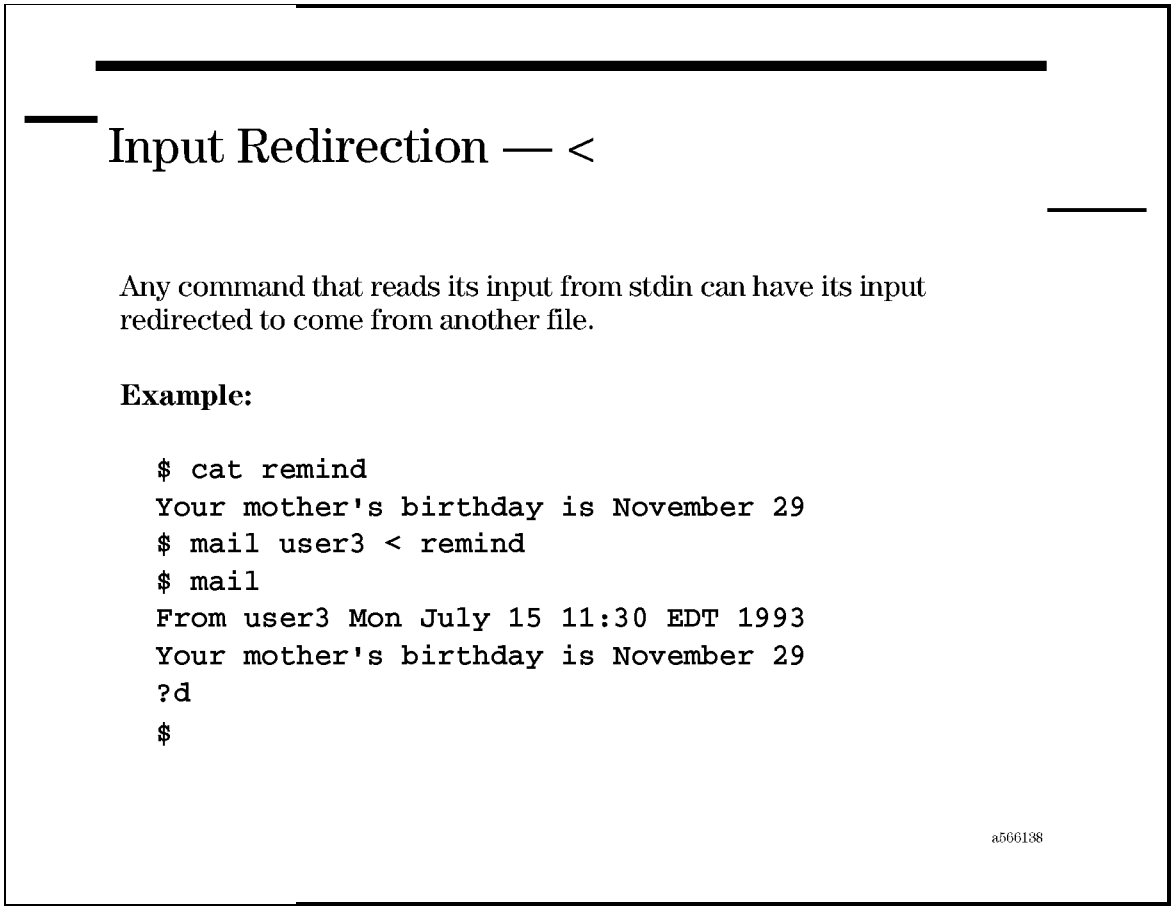
Don't go into details about how to use **`stdin`**, **`stdout`**, and **`stderr`** here. The file descriptor numbers are useful to know about especially when we want to redirect standard error messages with **`2>`**.

Student Exercises

You might want to have the students enter the **`date`** command and explain that the date is transmitted to standard output.

You might also want to have the students enter **`cp`** Return and note that the usage message is transmitted to standard error.

11-3. SLIDE: Input Redirection — <



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.

Input from stdin:	Operate on cmd line arg(s):	Redirect input:
$ **cat** Return *input text here* Ctrl + 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.

11-3. SLIDE: Input Redirection — < Instructor Notes

Key Points

- You can identify commands that accept standard input by entering the command and a Return. If the prompt disappears and the system seems to be waiting, it is waiting for you to provide input from the keyboard. Other commands that accept standard input are presented in this and other modules.
- Distinguish input provided through command line arguments and from input provided through standard input (stdin).
- Using a file as input is non-destructive to the input file.

Teaching Tips

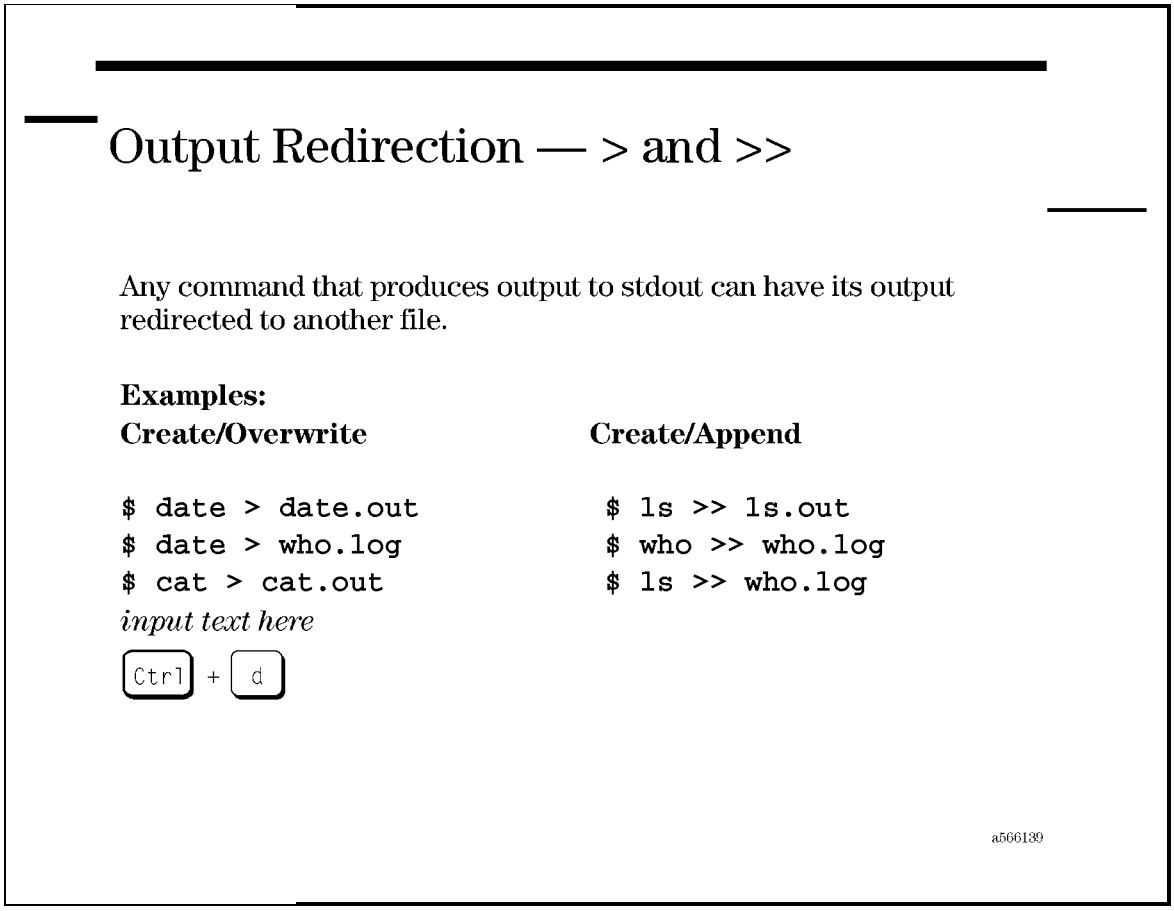
Think of the < as an arrow. The input of the command is coming from the file specified on the line: **command** <— **filename**

Teaching Question

Do the commands **ls, banner,** or **echo** accept input from stdin?

No, they do not. Typing **ls** Return executes the **ls** command. Typing **banner** Return generates a banner error (usage) message. Typing **echo** Return echoes a blank line to your screen.

11-4. SLIDE: Output Redirection — > and >>



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.

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.

11-4. SLIDE: Output Redirection — > and >> Instructor Notes

Key Points

- Output redirection captures messages going to standard output (the screen) and sends them to the specified file instead.
- You will see *no* command output displayed on your terminal, unless an error message is generated.
- The **tee** command is presented in the module *Pipes*, to show how to send output to the screen *and* to a file.
- You can append to a file with **>>**.

Teaching Tips

Think of the **>** as an arrow. The output of the command is going to the file specified on the line: **command** —-> **filename**

Redirecting Standard Output to Standard Error

You might want to mention redirecting standard output to standard error. This is especially useful when developing error messages in your scripts. Remember the only way to generate a message is through the **echo** command, but the **echo** command only sends its output to standard output. You would like the **echo** command to send its output to standard error instead:

```
echo error message >&2
```

This captures the standard output (**>**) and redirects it to the file associated with the standard error stream (**&2**). Remember that standard error is associated with file descriptor **2**.

Redirecting Input and Output on the Same Command Line

Since input and output are distinct streams, you can redirect both on the same command line. The command, of course, must accept standard input AND generate standard output.

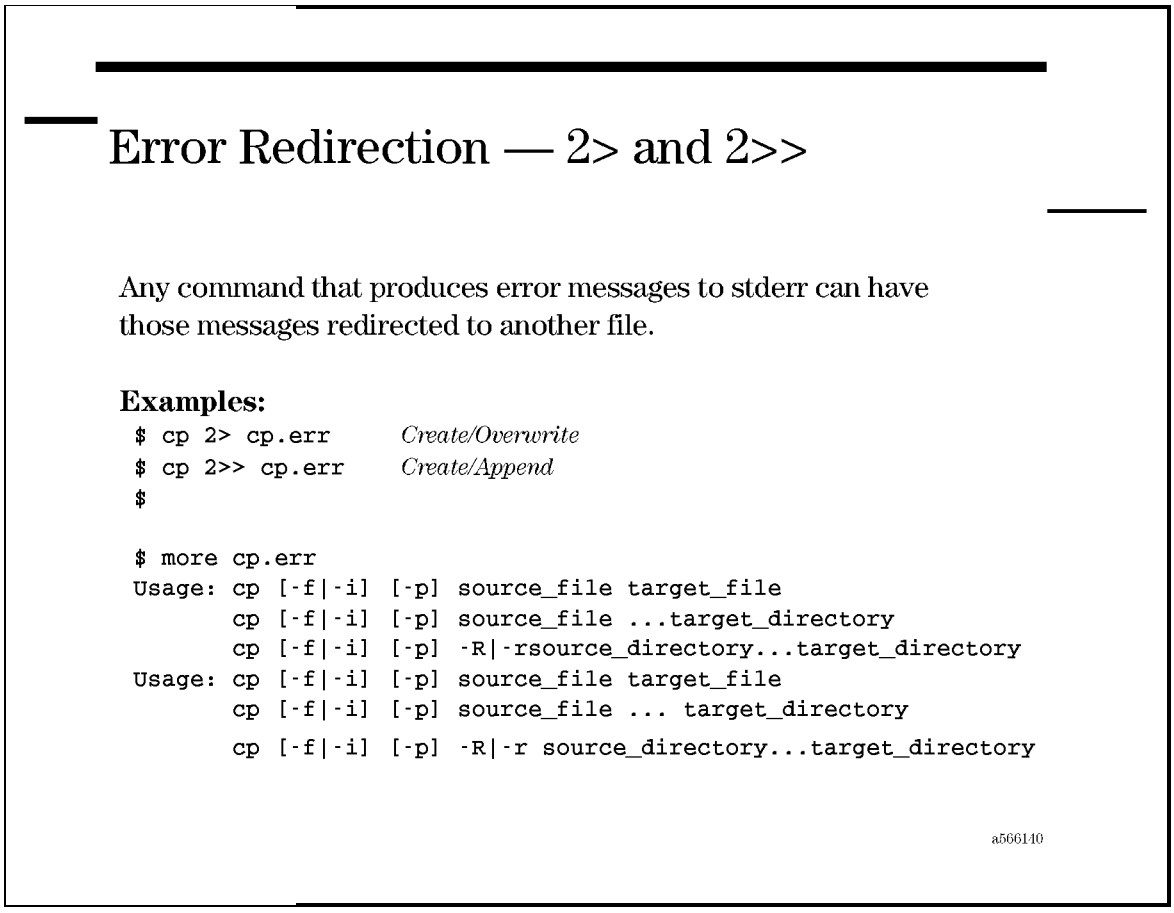
We can use the **cat** command as an example:

```
cat < f1 > f2
```

This effectively copies the contents of **f1** to **f2**.

The only restriction is that the input file and the output file *must* be different. The shell cannot open a file for input redirection and output redirection at the same time. If you do, the contents of the file will be *lost*.

11-5. SLIDE: Error Redirection — 2> and 2>>



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.

11-5. SLIDE: Error Redirection — 2> and 2>> Instructor Notes

Key Points

- **2>** provides for error logging.
- *No space* between the **2** and the **>**.

Teaching Tips

Point out that the error messages are separated from the standard output in most UNIX system commands. You might also point out that **2>&1** will send error messages to the same place as the standard output. Other ways to use standard error redirection follow.

You redirect the standard output messages from the **echo** command to standard error:

```
echo "message" >&2
```

This allows you to create error messages for your shell scripts.

These messages can then be saved to an error file, like any other message transmitted to **stderr**:

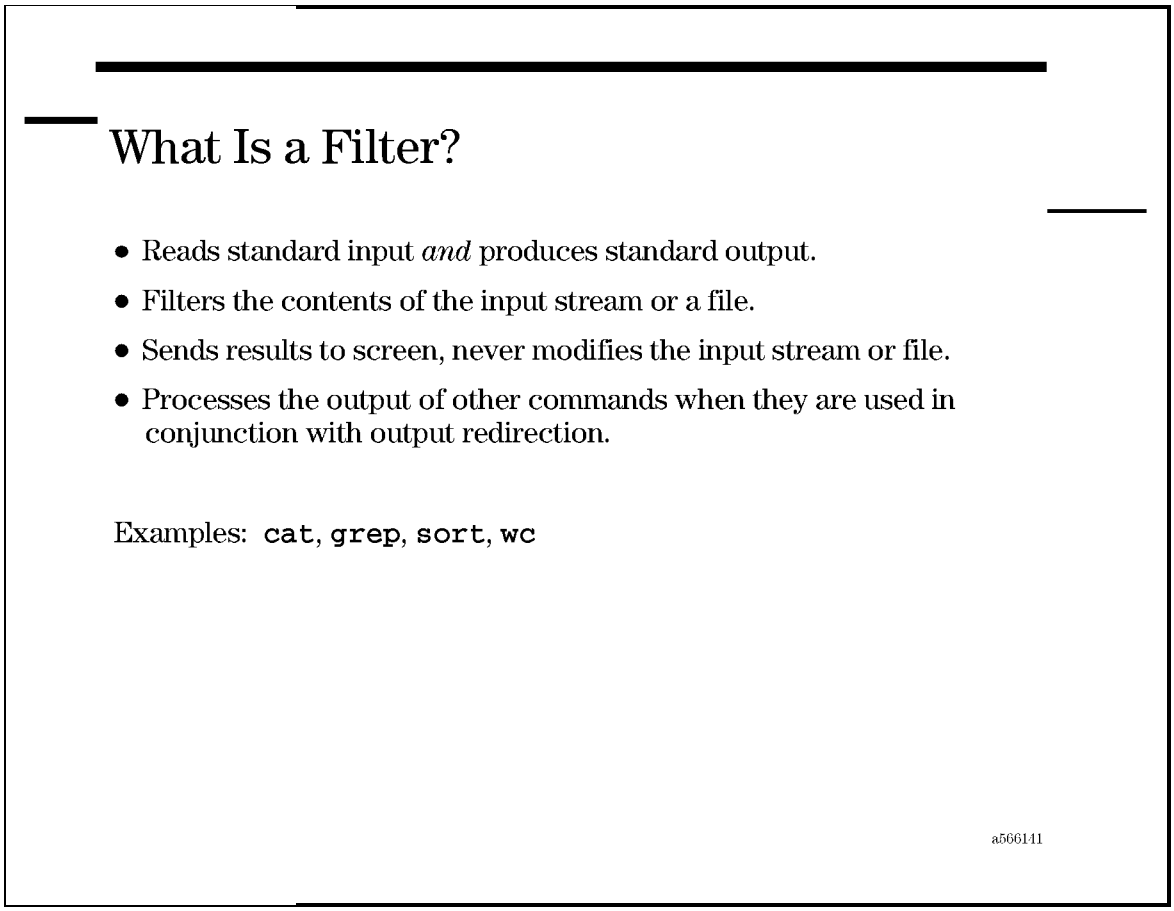
```
my_prog 2> my_prog.err
```

When you want to disregard the error messages, you can send them to the *bit bucket* — **/dev/null**:

```
$ command 2> /dev/null
```

When looking at the above examples, students often are confused between redirecting *to* standard error (**>&2**) and capturing the messages that are coming *out* of standard error (**2>**). Considering the source and destination will often help to reduce the confusion.

11-6. SLIDE: What Is a Filter?



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.

11-6. SLIDE: What Is a Filter? Instructor Notes

Key Points

- Filters accept input *and* generate output.
- Filters generally can operate on files.
- Filtered output generally goes to the screen.
- Filtered output can be saved to a file.
- Filters can be used to process the output of any command when the command's output is saved to a file.

Teaching Tips

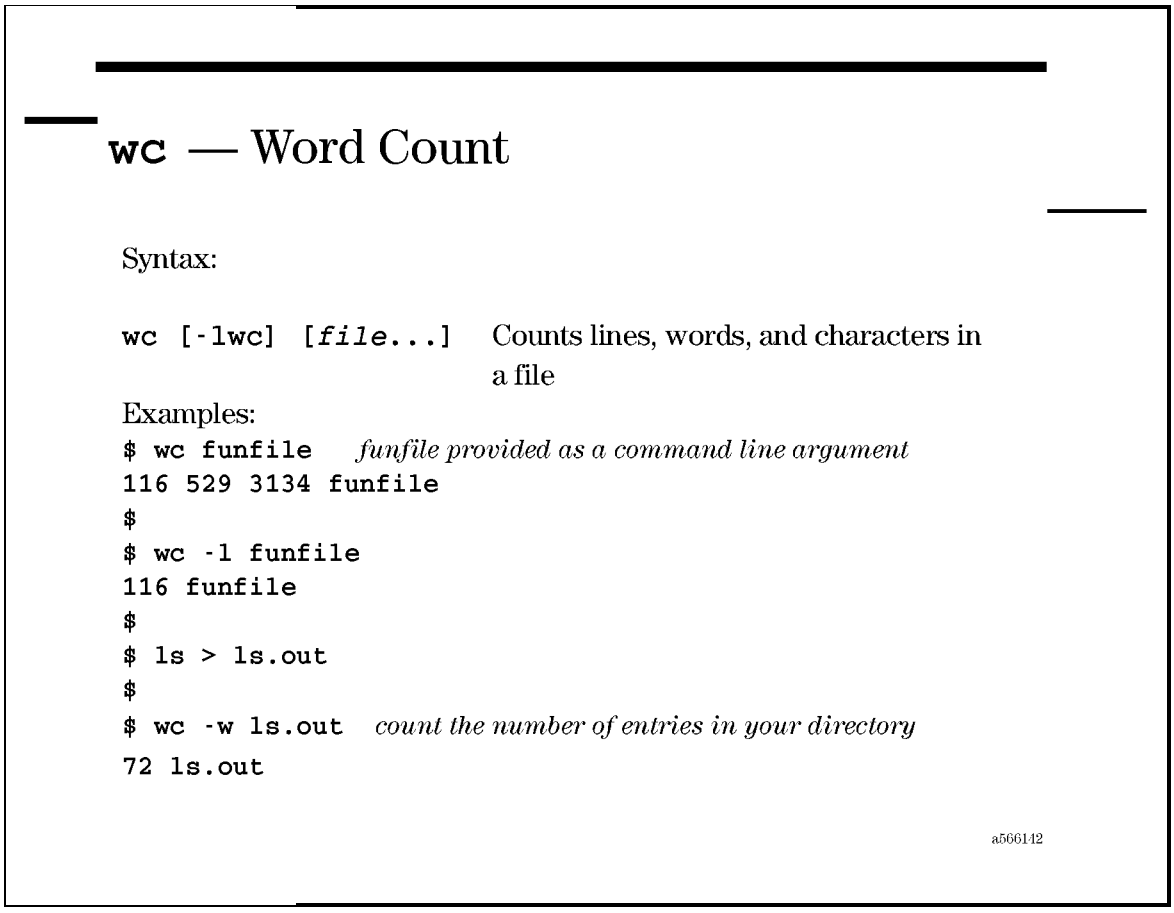
Don't explain these filters here; they are explained over the next several pages.

This is where the *real* power of the UNIX system is first exhibited. Output redirection and filters provide a very powerful and flexible mechanism to perform additional processing on the output of many commands.

Transition

The remainder of this module will introduce you to three useful filters: **`wc`**, **`sort`** and **`grep`** .

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
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.

11-7. SLIDE: wc — Word Count Instructor Notes

Key Points

- **wc** is a filter. It will accept input from the keyboard or operate on the contents of a file provided as a command line argument.
- Therefore, you can also redirect a file into the **wc** command.
- The contents of the file are unaffected.
- The output of commands can be captured, and the **wc** command executed on the resultant file.

Teaching Tips

Have the students type in the examples that are shown on the slide and discuss their results.

The last example on the slide illustrates how the output of a command can be further processed through output redirection.

To reinforce that **wc** actually accepts standard input, you should discuss the first example shown in the student notes. It is especially important that students have a strong understanding of commands that accept standard input, if you are going to be presenting the module *Pipes* .

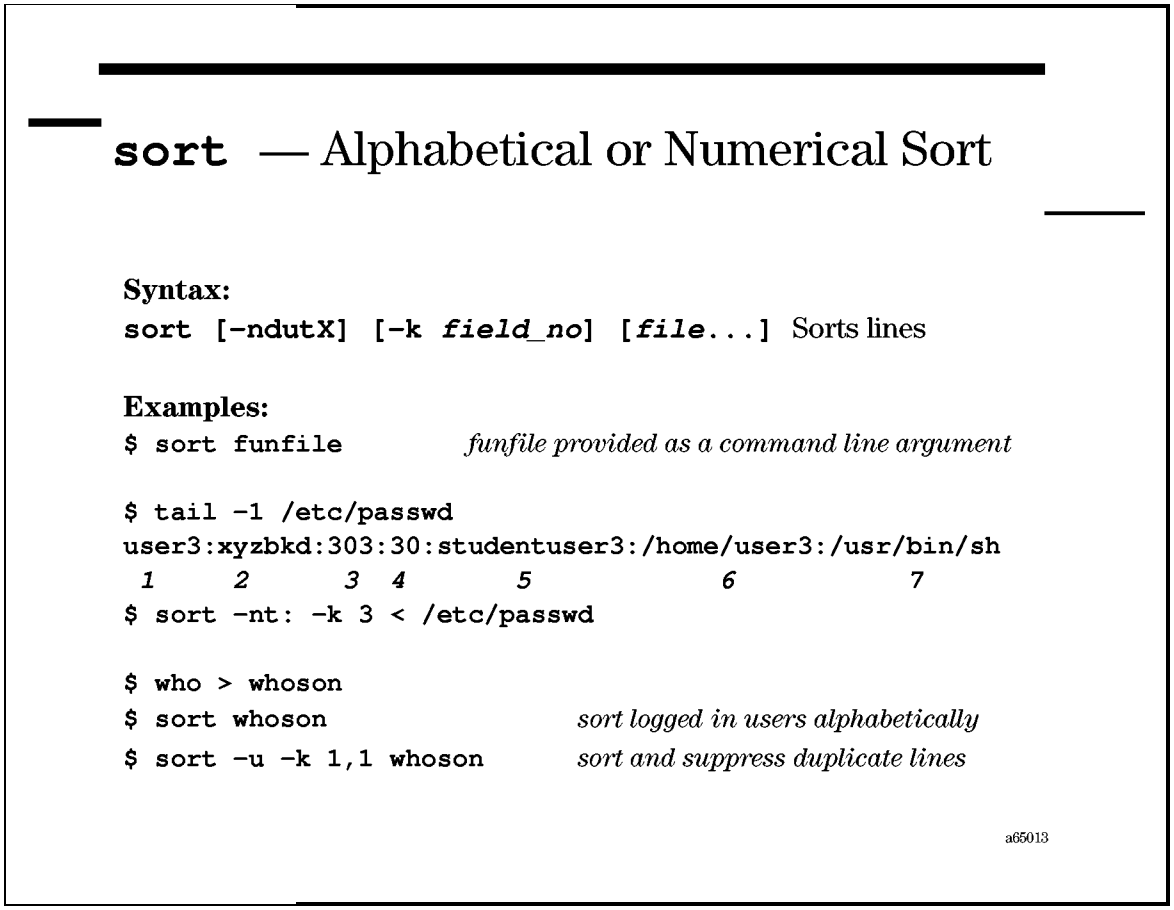
You might also want to discuss the different options that **wc** provides.

Teaching Question

Can you save the output from the **wc** command to a file?

ANSWER:
Yes! **wc funfile > fun.count**

11-8. SLIDE: `sort` — Alphabetical or Numerical Sort



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 which 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. 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**.

11-8. SLIDE: `sort` — Alphabetical or Numerical Sort — Instructor Notes

Key Points

- **`sort`** is a filter.
- **`sort`** can read from standard input *or* use a filename specified as an argument on the command line.
- The output of other commands can be sorted by using output redirection to save their output to a file.

Teaching Tips

Have the students type in the examples that are shown on the slide and discuss the results. Point out that most filters can read from stdin *or* take the file argument from the command line.

The field numbers are provided on the slide to assist you in explaining how the **`sort`** command identifies the fields in each line of **`/etc/passwd`** with a the colon designated as the delimiter.

Your students will probably notice when they sort **`funfile`** that the blank lines are displayed first. You might want to tell the students to enter:

```
sort funfile | more
```

so that they can see the output one screen at a time. Pipelines are presented in a separate module

Advanced Sorting Capabilities

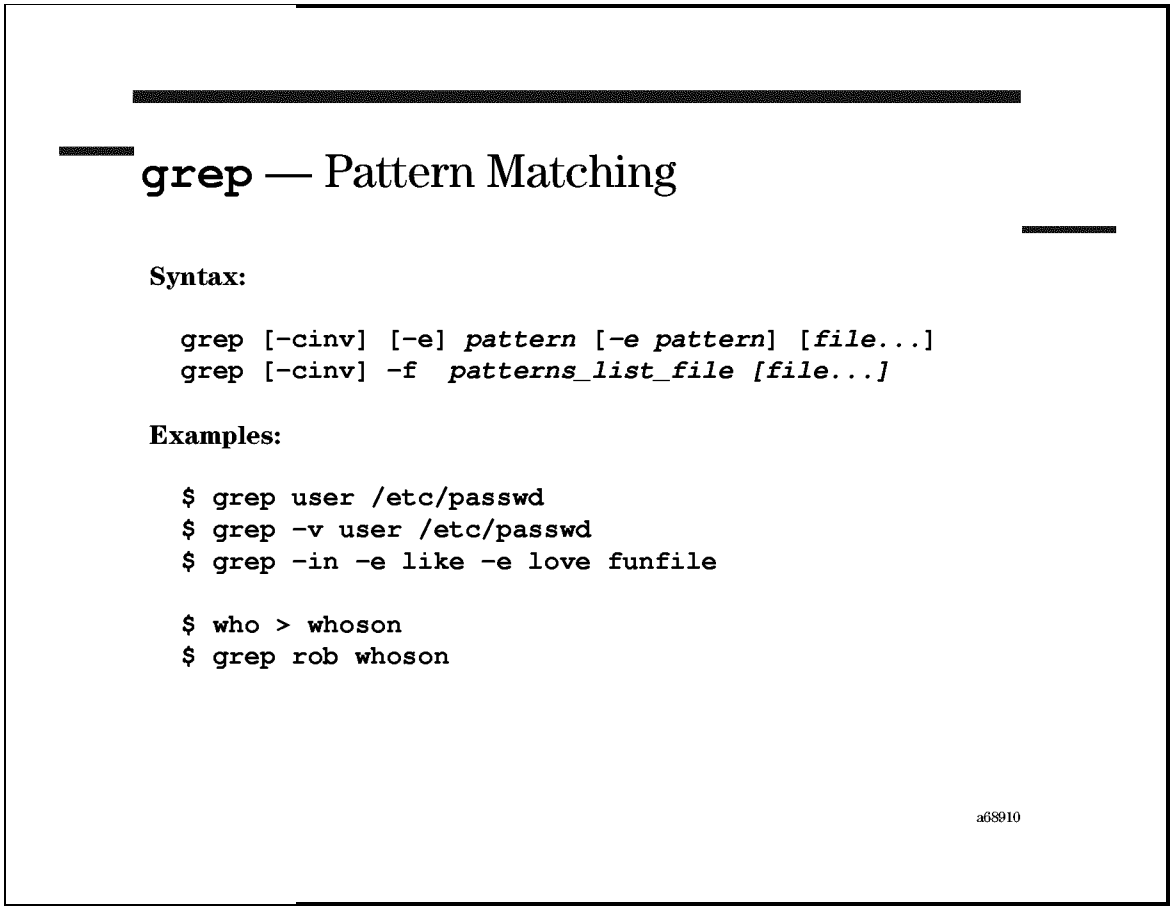
If you have some advanced students you might want to show them how to select a specific field (the examples on the slide assume that the rest of the line will be used), and also test for uniqueness. Duplications within the field will be disregarded.

```
$ sort -t: -k 1
jupiter:bbbbb
jupiter:aaaaa
mars:ccccc
mars:dddd
earth:zzzz
-------------------------
earth:zzzz
jupiter:aaaaa
jupiter:bbbbb
mars:ccccc
```

```
mars:dddd
$ sort -t: -k 1,1 -u                    sort only on the first field, disregard duplications

jupiter:bbbbb
jupiter:aaaaa
mars:ccccc
mars:dddd
earth:zzzz
-------------------------
earth:zzzzz
jupiter:aaaaa
mars:dddd
```

Note that you can also go to the character level within a field, using notation such as **`-k 1.1,1.5`** to sort based on characters 1 through 5 in the first field. See the **`man`** pages for details.

11-9. SLIDE: grep — Pattern Matching



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 which contain the specified pattern. The **grep** command then displays the lines which 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 which *do not* contain the pattern

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.

11-9. SLIDE: `grep` — Pattern Matching Instructor Notes

Key Points

- **`grep`** will search for lines that contain the specified pattern and will echo those lines back out to your terminal.
- The pattern is technically a regular expression.
- The **`-v`** option allows you to search for lines that do *not* contain the pattern.

Teaching Tips

You should have the students enter the commands that are listed on the slide so that they can observe how **`grep`** works. Have them try the last example initially without the **`vi`** edit, and ask why it fails.

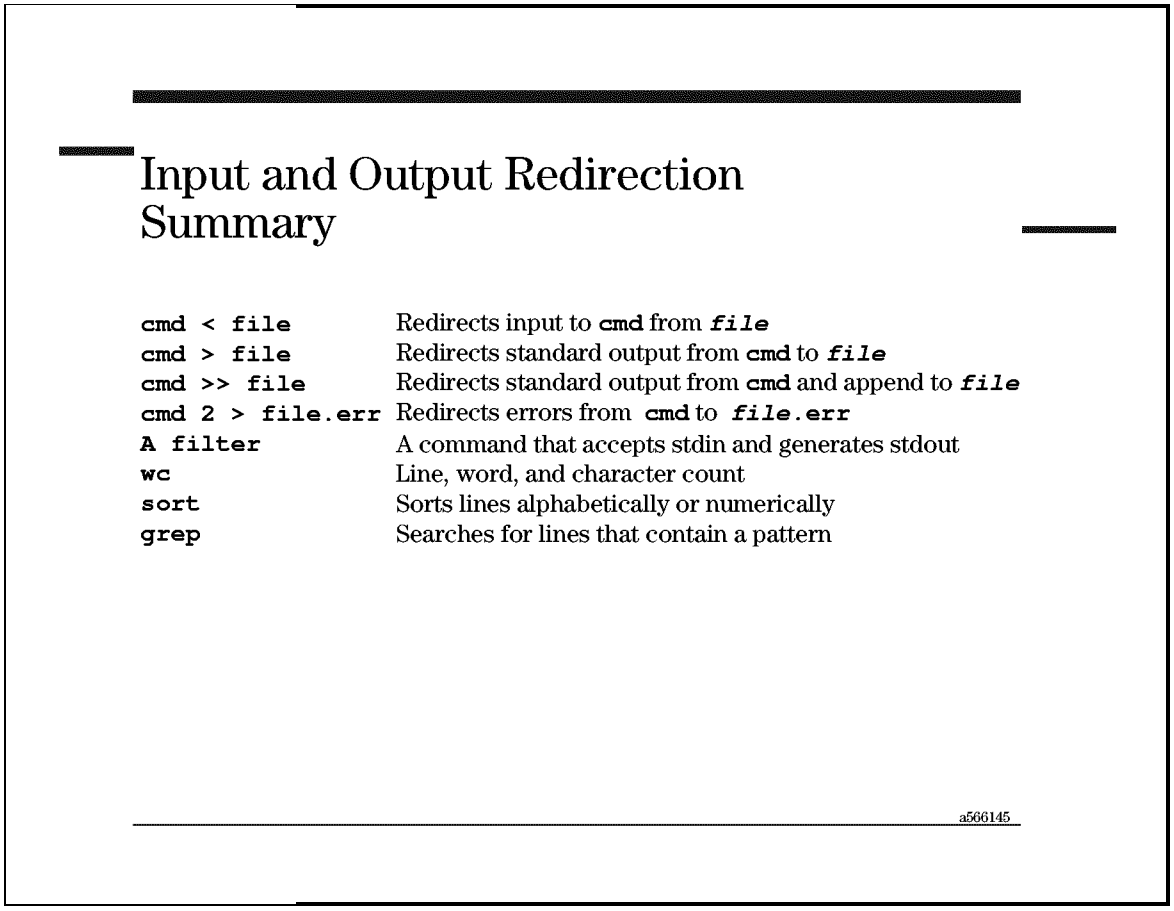
You may wish to tell the class that grep got its name from the **`ed`** command **`g/RE/p`** (global, regular expression, print). This globally (**`g`**) searches for a regular expression (**`/RE/`**), and prints (**`p`**) any line that contains the pattern.

You might want to have the students enter the following interactive example to illustrate how **`grep`** will echo out lines that *contain* a pattern. The following example is looking for the pattern *red* from standard input:

```
$ grep red Return
red white and blue
red white and blue
black and white
red and green
red and green Ctrl + d
```

- The line *red white and blue* will be immediately echoed back, since it contains red.
- The line *black and white* will *not* be echoed back because it does *not* contain red.
- The line *red and green* will be immediately echoed back, since it contains red.

11-10. SLIDE: Input and Output Redirection — Summary



Student Notes

11-10. SLIDE: Input and Output Redirection — Summary

Instructor Notes

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?

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.

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`**.

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?*

11-11. LAB: Input and Output Redirection — Instructor Notes

Time: 30 minutes

Purpose

To practice input and output redirection and filters.

Notes to the Instructor

The labs need the pattern *user* in **/etc/passwd** . This is normally not a problem, unless your user accounts are NOT under the **/home** directory. Also needed is the file **names** in the user's *HOME* directory.

Students who need to practice the basic concepts of input and output redirection should start with Exercise 1. Students who are comfortable with the basic concepts should start with Exercise 10.

Most students have fun with Advanced Exercise 16, sending output to another user's terminal. Remember, that the other user's terminal must have messaging enabled for the exercise to succeed.

Solutions

1. Redirect the output of the **date** command to a file called **date.out** in your *HOME* directory.

Answer:

```
$ cd
$ date > date.out
```

2. Append the output of the **ls** command to the file **date.out**. Look at the contents of **date.out**. What do you notice?

Answer:

```
$ ls >> date.out
$ more date.out
```

The output of the **ls** command is a list of files in the current directory. Each file name is on a separate line. The shell knows to put the output of the **ls** command in columns only when the output goes to the terminal. You can override this with the **-C** option to **ls**.

3. Using input redirection, mail the file **date.out** to your mail partner.

Answer:

```
$ mail mail_partner_login_name < date.out
```

4. Create two very short files called **f1** and **f2** using **cat** and output redirection.

Answer:

```
$ cat > f1
This is the file f1
Ctrl + d
$ cat > f2
This is the file f2
Ctrl + d
```

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?

Answer:

```
$ cat f1 f2
This is the file f1
This is the file f2
$ cat f1 f2 > f.join                output of both files is sent to f.join
```

You will not see any output on the screen. All of the standard output has been sent to the file **f.join**.

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?

Answer:

```
$ cat f1 f2 f.new
This is the file f1
This is the file f2
cat: Cannot open f.new
```

It is not obvious that two output streams are being used, since all of the messages are sent to your display.

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.

Answer:

```
$ cat f1 f2 f.new 2> f.error
This is the file f1
```

```
This is the file f2
$ cat f.error
cat: Cannot open f.new
```

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.

Answer:

```
$ cat f1 f2 f.new > f.good 2> f.bad
$ cat f.good
This is the file f1
This is the file f2
$ cat f.bad
cat: Cannot open f.new
```

The files **f.good** and the file **f.bad** are created. You do not see any output to your screen because all output streams have been redirected to one file or the other.

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?

Answer:

```
$ cp
Usage: cp f1 f2
cp [-r] f1 ... fn d1
$ cp 2> cp.error
```

The **cp** command does not generate any standard output messages. It is normally silent when it succeeds.

10. Display the contents of the file **/etc/passwd** sorted out by user name.

Answer:

```
$ sort -d /etc/passwd
```

11. Sort the file **/etc/passwd** on the third field. What happens? Now do a numeric sort on the third field. Any difference?

Answer:

```
$ sort -t: -k 3 /etc/passwd          lexicographic sort
```

(Note that the numbers in the third field are not quite sorted. This is because an ASCII sort is being done on a numeric field.)

```
$ sort -nt: -k 3 /etc/passwd         numeric sort
```

(The results of this command are much better since the numbers in the third field are now arranged numerically.)

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*.

Answer:

```
$ grep user /etc/passwd > grepped
$ wc -l grepped
16 grepped
```

(Note that on the system you are using, this number may vary.)

13. Using redirection and filters, how many users are logged in on the system?

Answer:

```
$ who > whoson
$ wc -l whoson
```

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**.)

Answer:

```
$ wc -l /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**.

Answer:

```
$ sort names > names.sort
$ sort -r names > names.rev
$ sort names -o names
```

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?*

Answer:

You must first determine the device file associated with your mail partner's terminal:

```
$ who > whoson
$ grep mailpartner whoson
mailpartner tty03 Jul 16 8:02
```

Check out the **tty** designation. This tells you what device file is associated with your mail partner's terminal session.

```
$ banner good morning > /dev/tty03
```

If you get a *Permission denied* message, your mail partner has disabled the write permissions on his or her terminal with the command, **`mesg n`**.

Pipes

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.

Pipes

Overview of Module 12

Audience

general user — General system user

Product Family Type

open sys — Open systems environment

Abstract

This module is designed to show the student how to construct pipelines. Also, four filters are discussed: **`tee, cut, tr`** and **`pr`**.

Time

Lab — 45 minutes

Lecture — 60 minutes

Prerequisites

m1306m — Input and Output Redirection

Instructor Profile

UX — General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000 — Logon access to an HP-UX system

Software Requirements

UX11 — HP-UX release 11.0

Material List

P/N B2355-90033(T)	*HP-UX Reference Manual* , one per terminal

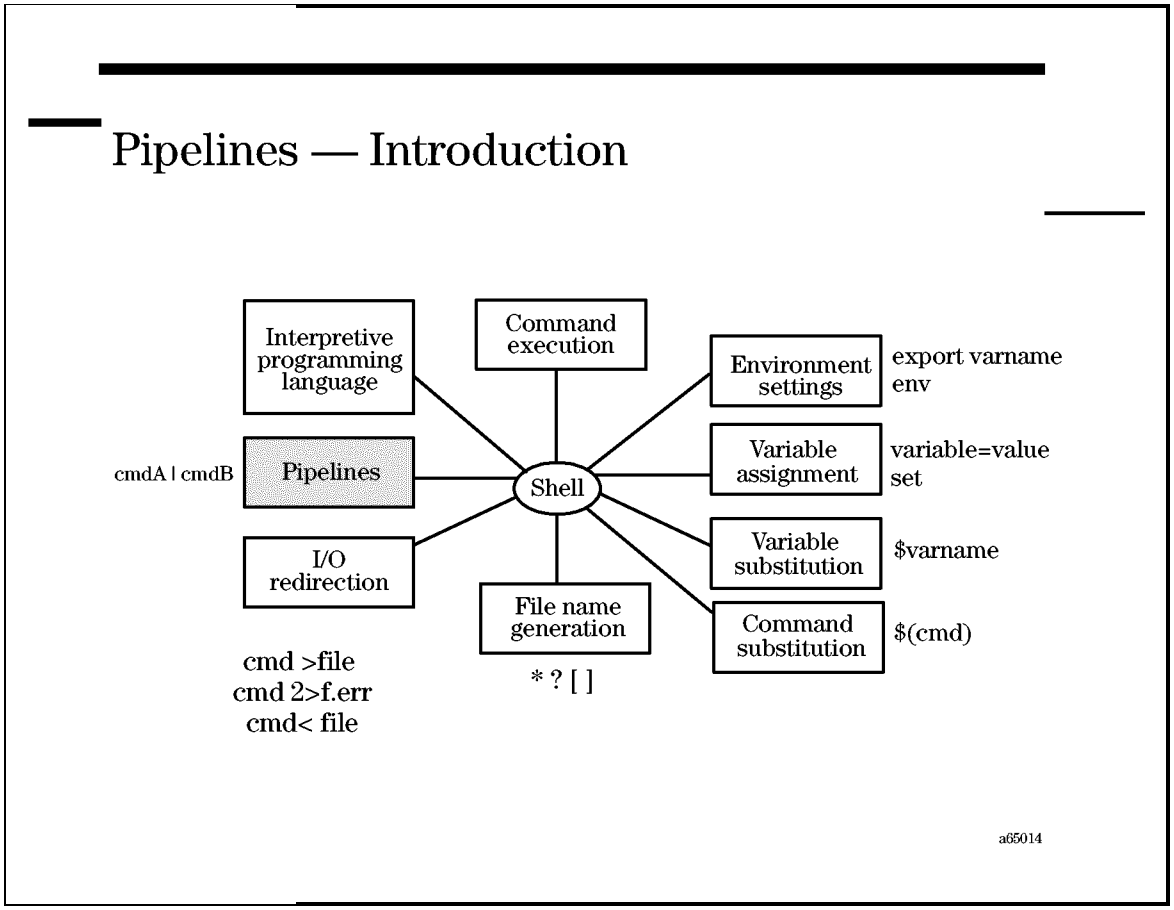
Lab Instructions

setup1	Create user logon of **user1**, **user2**, ... **user***n*, where *n* is the number of students in the class. Set up one user per student.
copyfiles	Copy the lab files to the users' home directories.

Lab Files

```
-rw-r--r-- 1 karenk users 61 May 28 16:12 names
```

12-1. SLIDE: Pipelines — Introduction



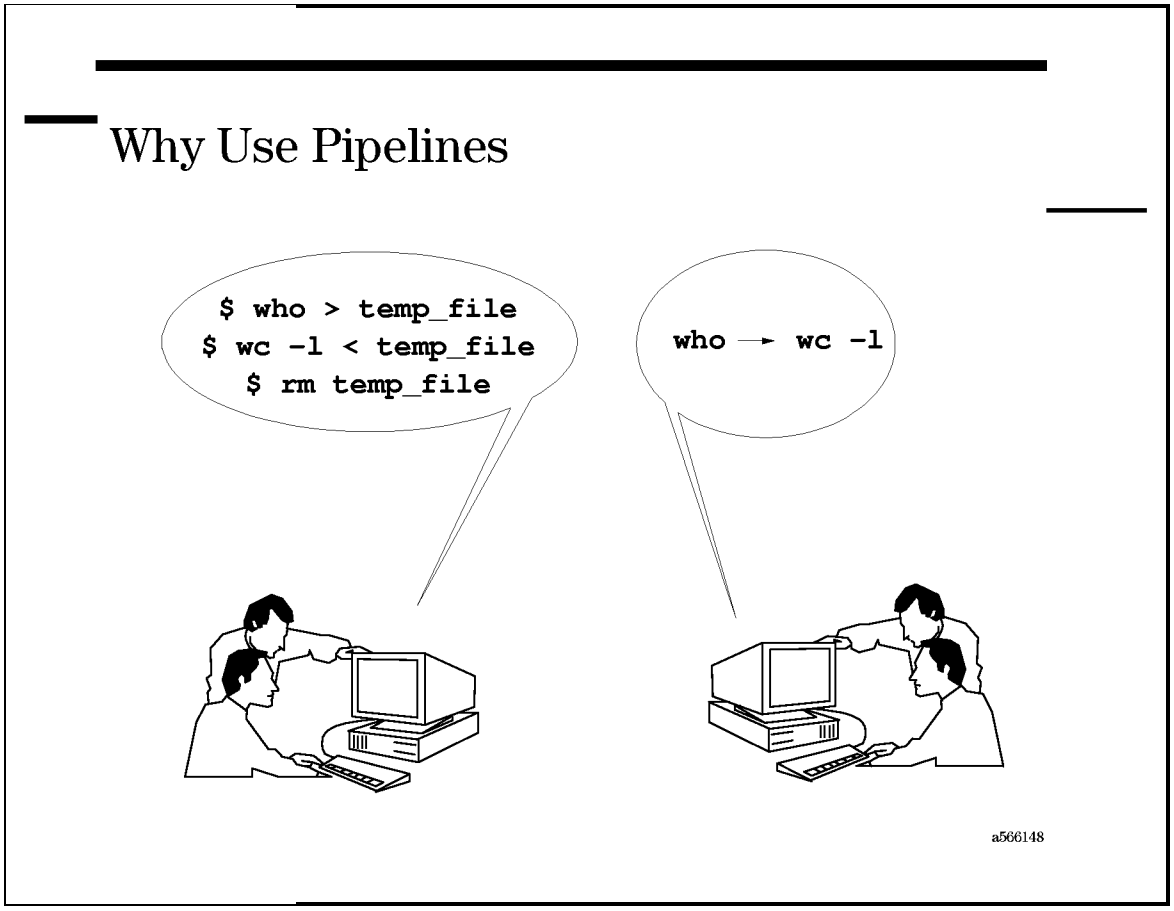
Student Notes

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.

12-1. SLIDE: Pipelines — Introduction Instructor Notes

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.

12-2. SLIDE: Why Use Pipelines? Instructor Notes

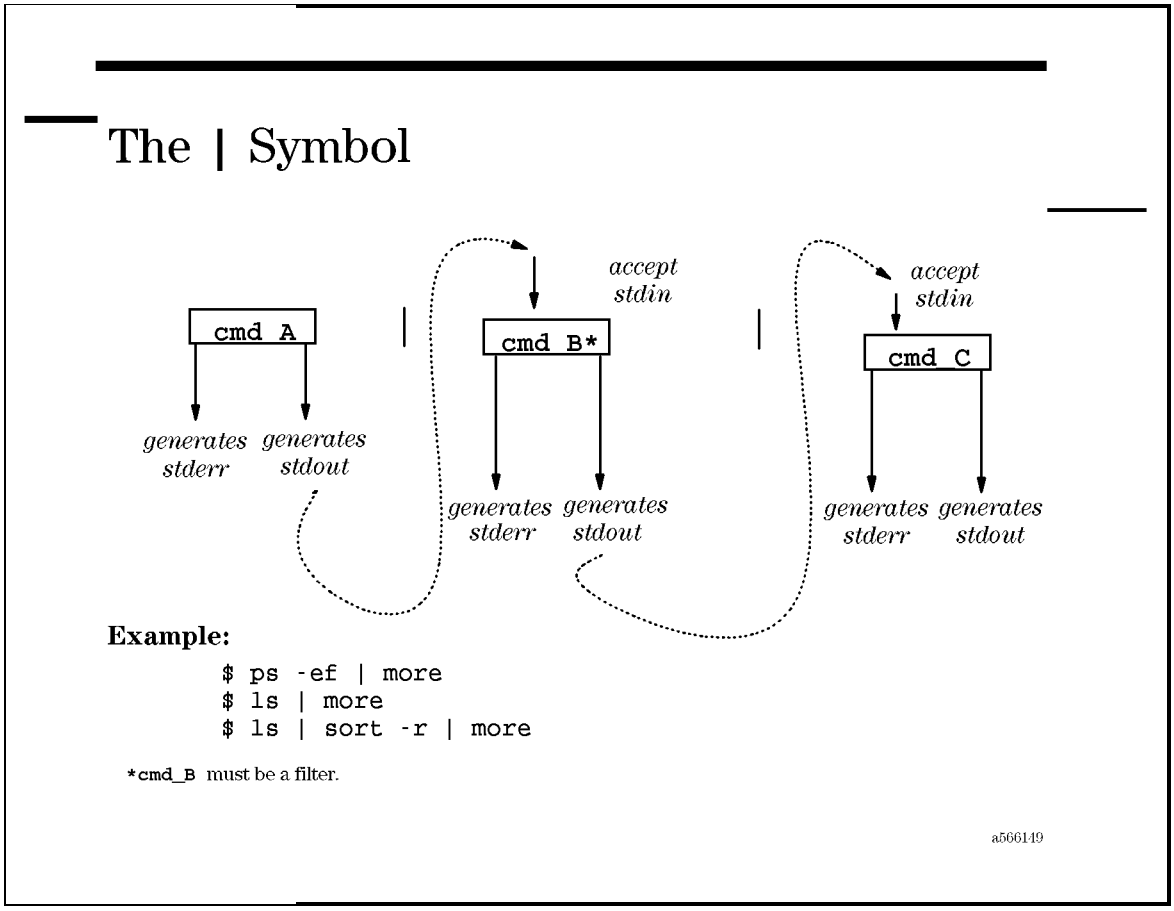
Key Points

- Pipelines allow chaining of commands.
- The output of one command becomes the input for another command.
- There is no temporary file to remove.

Teaching Tip

- Describe why pipelines are needed. Do not explain how to use them, yet.

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.)

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.

Pipes

12-3. SLIDE: The | Symbol — Instructor Notes

Key Points

- Present the requirements for each command position in the pipeline.
- The output of a command in the middle of a pipeline is not directly viewable by the executor — see the third example on the slide.
- We will look at the **`tee`** command, which allows us to capture the output of commands in the middle of a pipeline.
- The **`more`** command accepts standard input, so it can be used to view the output of any command that generates output to standard output one screen at a time

Teaching Questions

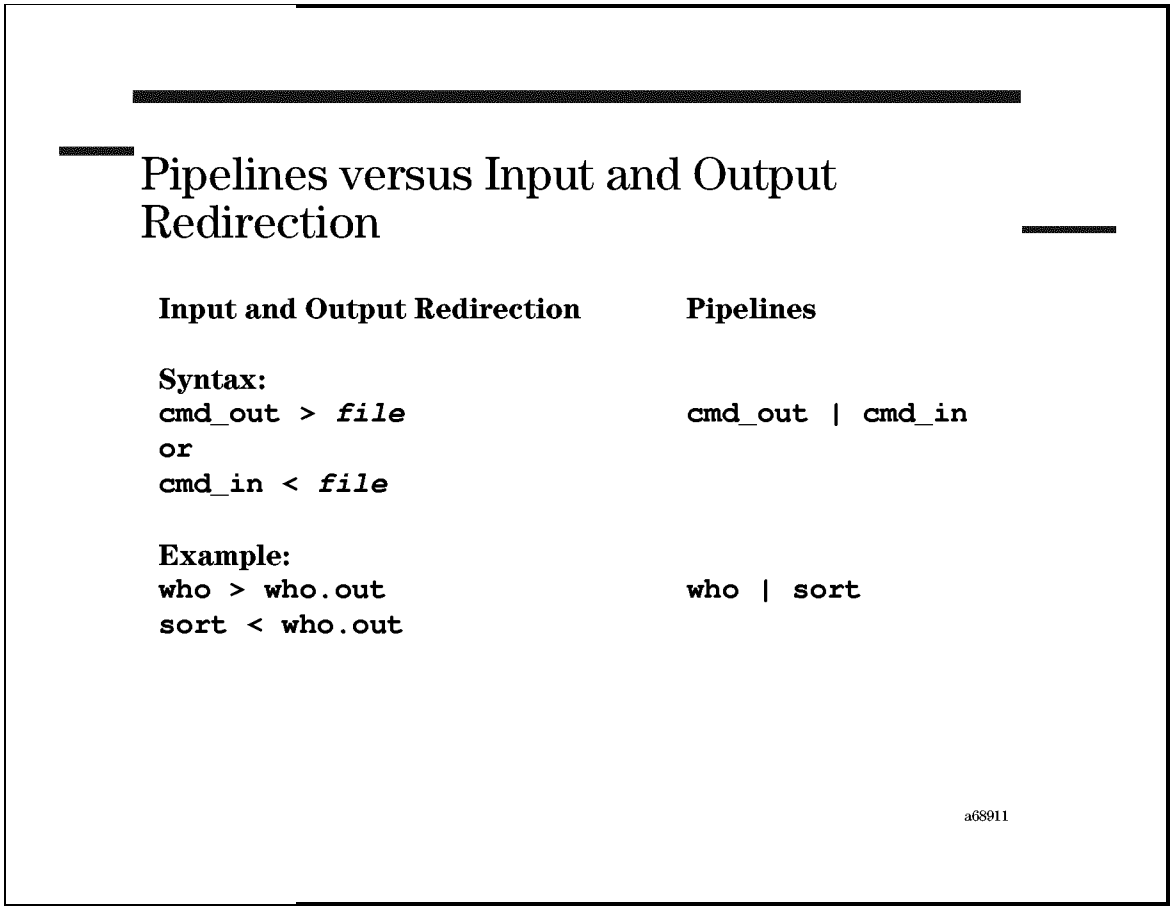
1. Using a pipeline, how could you count the number of entries under your current directory?
   Answer: **`ls | wc -w`**

2. Using a pipeline, how could you count the number of subdirectories under your current directory?
   Answer: **`ls -F | grep / | wc -w`**

3. Can the **`cp`** command appear in the middle of a pipe?
   Answer: No, because it does not accept standard input and it does not generate standard output.

4. Can the **`mail`** command appear in the middle of a pipe?
   Answer: No, it does accept standard input, but it does not generate standard output.

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.

12-4. SLIDE: Pipelines versus Input and Output Redirection — Instructor Notes

Key Points

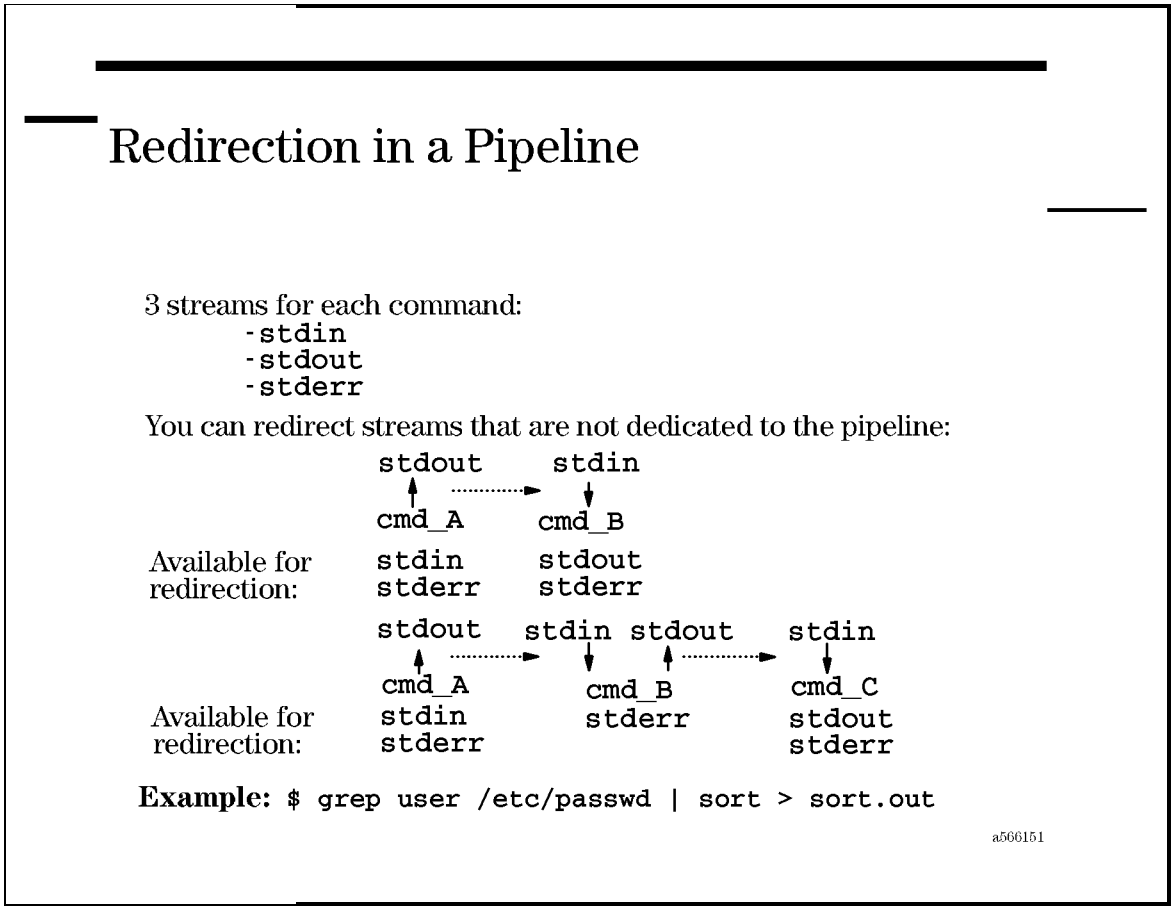
- Redirection is always between a command and a file.
- Pipelines are always between two or more commands.

Teaching Questions

Can piping and redirection be combined on a single command line?
Answer: Yes

Can the output of a pipe be redirected to a file?
Answer: Yes

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.

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.

12-5. SLIDE: Redirection in a Pipeline — Instructor Notes

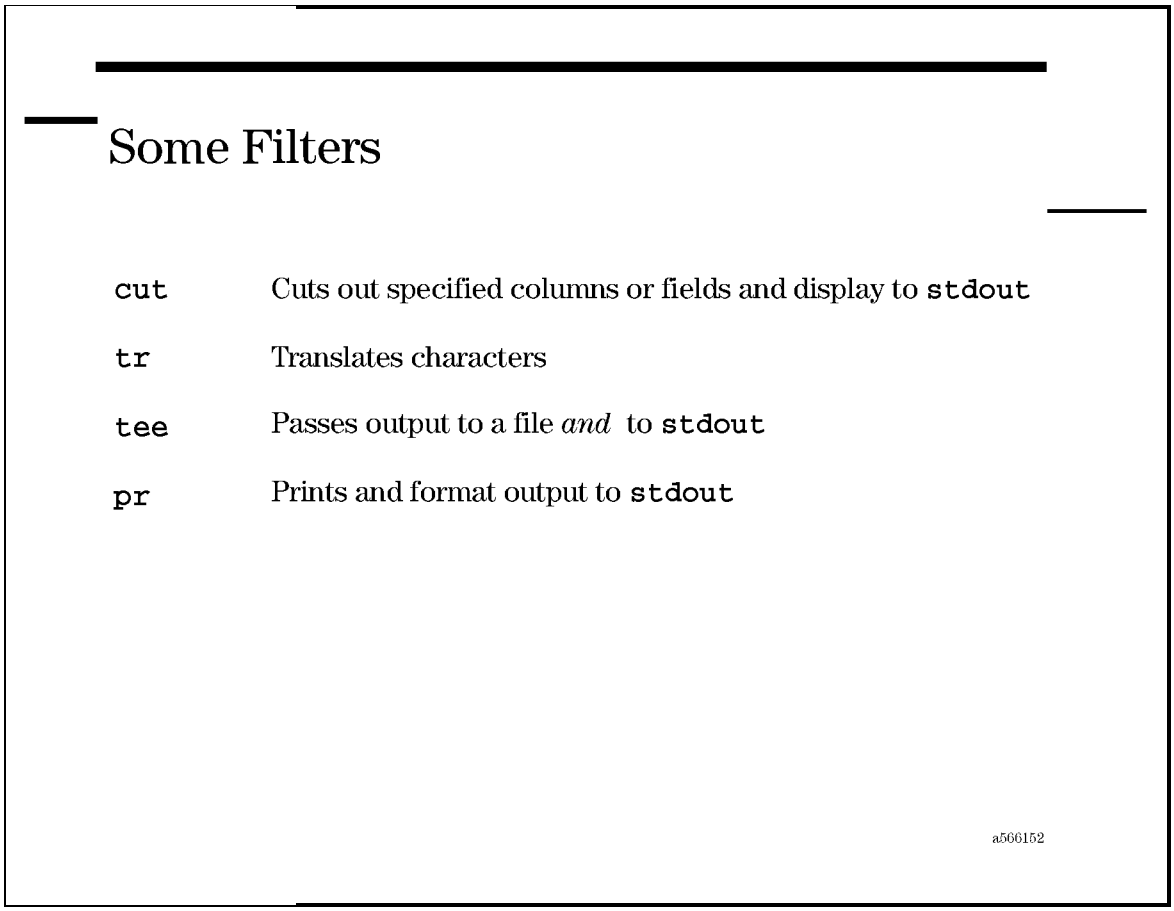
Key Points

- This slide serves as a review of the I/O redirection and pipeline concepts. If students understand what redirection is allowed in a pipeline, they have a solid grasp of redirection *and* pipelines.
- The slide illustrates which streams are dedicated to the pipeline (above each pipeline) and the streams that are not dedicated to the pipeline (below the pipeline). The paths above the pipelines are illustrating how the **`stdout`** on the left of a pipe becomes the **`stdin`** for the command on the right side of the pipe.
- Streams that are not dedicated to the pipeline can be redirected.

Teaching Tip

- The student notes provide some additional examples that use more redirection than the example on the slide.

12-6. SLIDE: Some Filters



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.

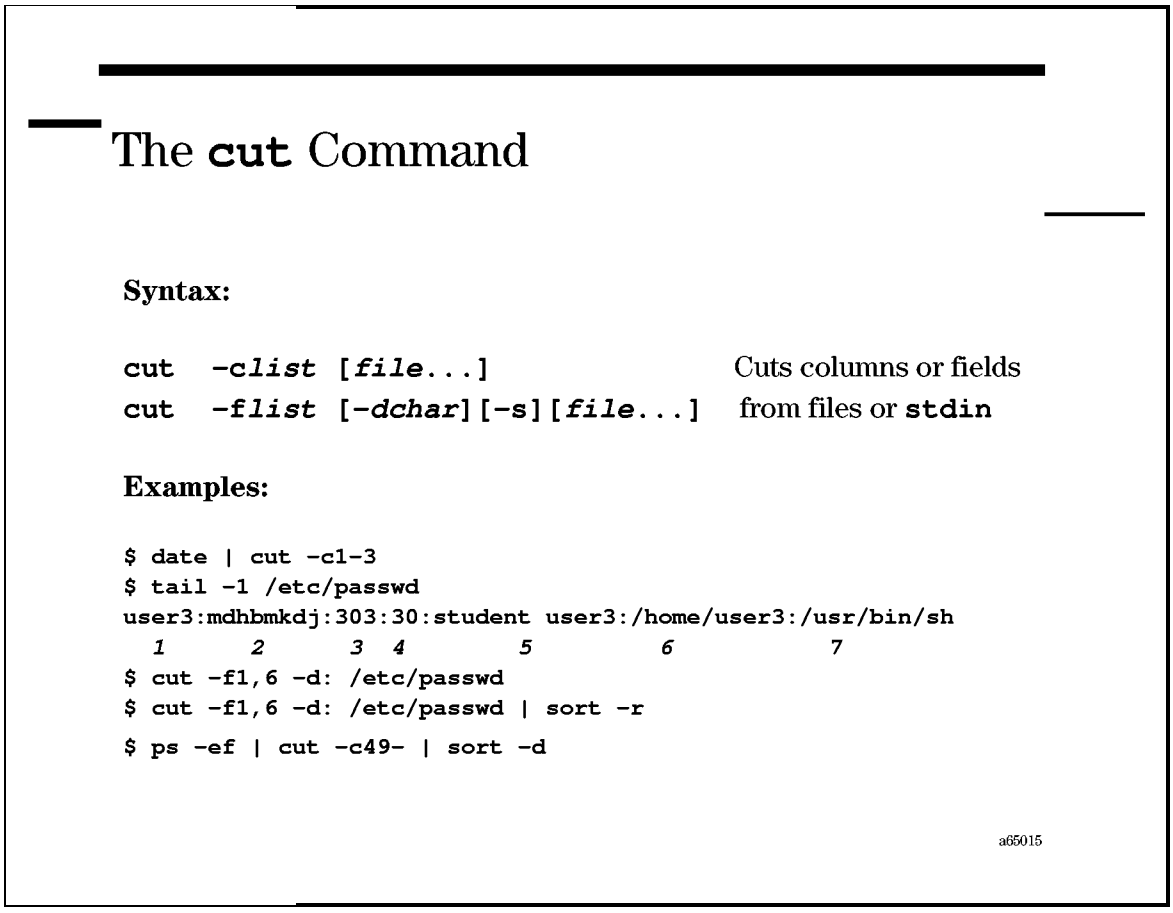
As with all filters, these commands will not modify the original file. The processed results will be sent to standard output.

12-6. SLIDE: Some Filters — Instructor Notes

Key Points

- These are filters.
- They accept standard input; they generate standard output.
- The original file is not affected.

12-7. SLIDE: The `cut` Command



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*

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.

Examples

```
$ cut -c1-3 Return
12345
123
abcdefgh
abc
Ctrl + d

$ date | cut -c1-3
```

Pipes

12-7. SLIDE: The `cut` Command — Instructor Notes

Key Points

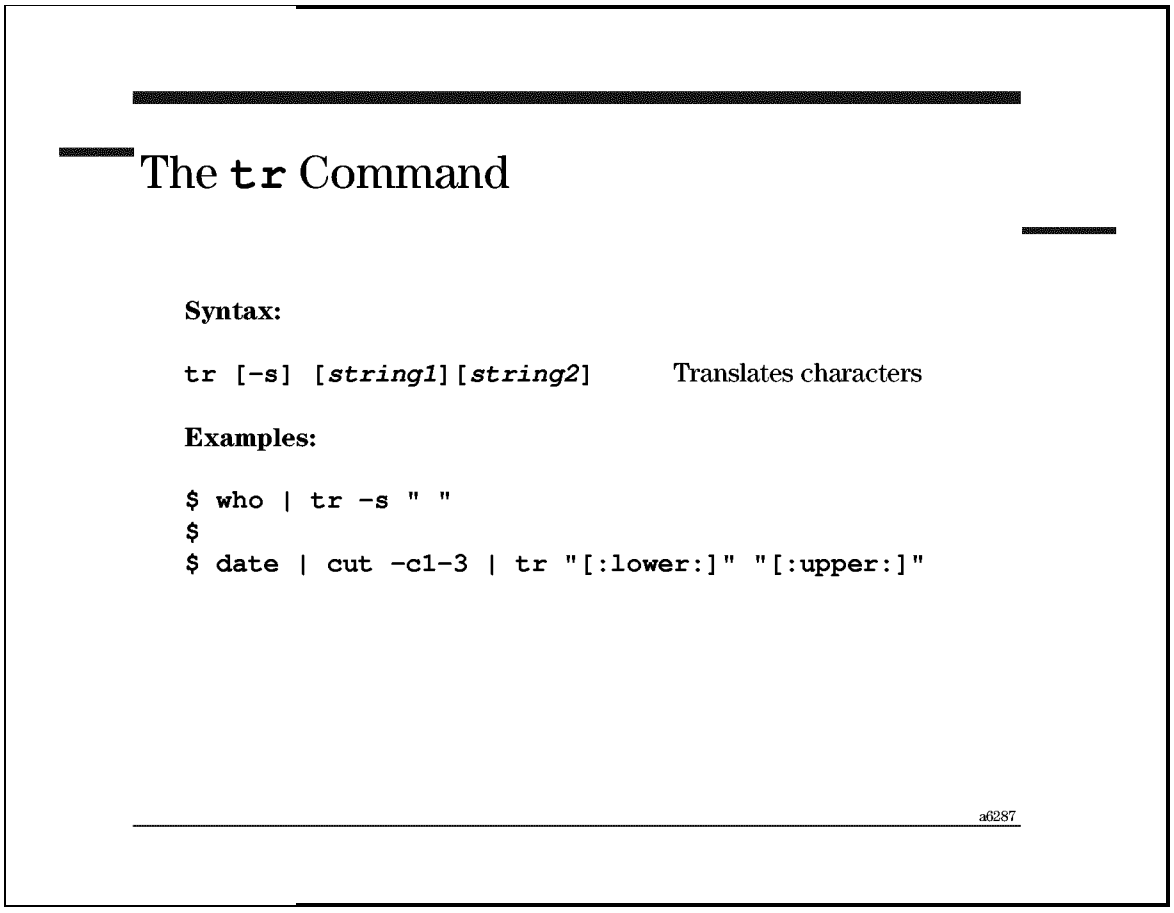
- Cut the specified columns or fields from **`stdin`** or a file and send to **`stdout`**.
- Fields start numbering at 1.
- Useful for extracting part(s) of a line of text.

Teaching Tips

Have your students key in the examples on the slide so they can see how the **`cut`** command works. You might also want to have them try an interactive example, as listed in the student notes, to reinforce that **`cut`** accepts standard input.

The second example demonstrates how a filter can be used with command substitution. Since filters operate only on standard input or a file, the value of the variable must be echoed and piped to the **`cut`** command.

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.

12-8. SLIDE: The `tr` Command — Instructor Notes

Key Points

- **`tr`** is useful for converting multiple blank spaces to a single space. Examples of commands are **`who`**, **`ps`**, and **`date`**, which embed a variable number of spaces between fields.
- **`tr`** allows the use of the field option for **`cut`** or **`sort`** instead of referencing literal column numbers.
- **`tr`** can be used to convert strings from lowercase to uppercase or from uppercase to lowercase.
- The **`s`** option squeezes all strings of repeated output characters that appear in *string2* to single characters.

12-9. SLIDE: The `tee` Command

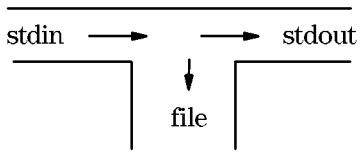
The `tee` Command

Syntax:

`tee [-a] file [file. . .]` Tap the pipeline

Example:

```
$ who | sort
$ who | tee unsorted | sort
$ who | tee unsorted | sort | tee sorted
$ who | wc -l
$ who | tee whoson | wc -l
```



a566155

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.

Pipes

12-9. SLIDE: The `tee` Command — Instructor Notes

Key Points

- **`tee`** is useful for capturing intermediate output in a pipeline that cannot normally be viewed (examples 2 and 5 on the slide).
- **`tee`** is useful for logging the output of commands to a file and seeing the output on the screen (example 3 on the slide).
- Differentiate using the **`tee`** command at the end of a command versus redirecting. When output is redirected, you will not see any output to the screen. When the output is **`tee`**'d, the output goes to the screen and to the file.

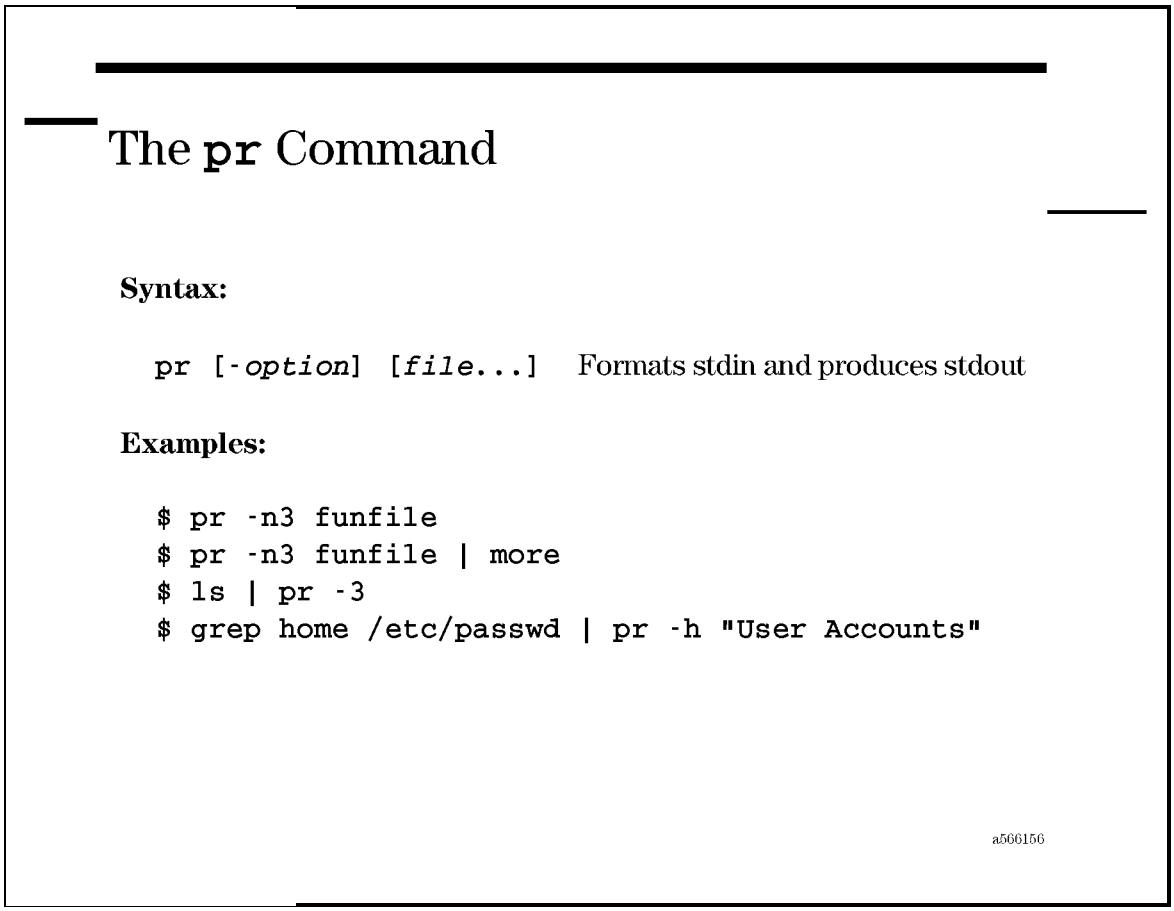
```
$ who | tee unsorted | sort > sorted
```
Output just to file

```
$ who | tee unsorted | sort | tee sorted
```
Output to screen and file

Teaching Tips

Have the students type in the commands that are listed on the slide.

12-10. SLIDE: The pr Command



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

-d	Doublespaces the output
-w*N*	Sets the width of a line to *N* characters
-l*N*	Sets the length of a page to *N* lines
-n*CK*	Produces *K*-digit line numbering, separated from the line by the character *C*; *C* defaults to Tab
-o*N*	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

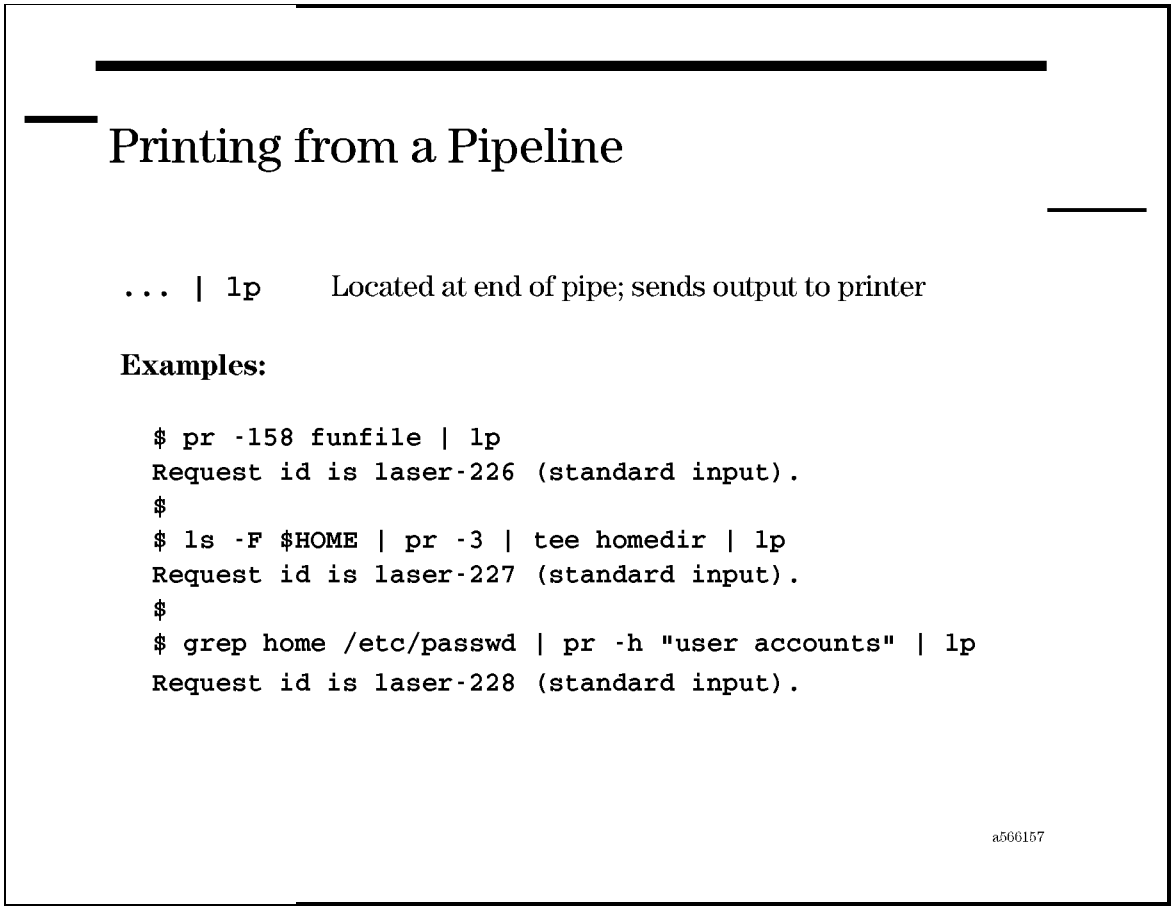
Pipes

12-10. SLIDE: The `pr` Command — Instructor Notes

Key Points

- **`pr`** does not send output to the printer; it sends output to **`stdout`**.
- **`pr`** stands for *print to stdout* or *pretty*.
- **`pr`** has many handy options.

12-11. SLIDE: Printing from a Pipeline



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.

12-11. SLIDE: Printing from a Pipeline — Instructor Notes

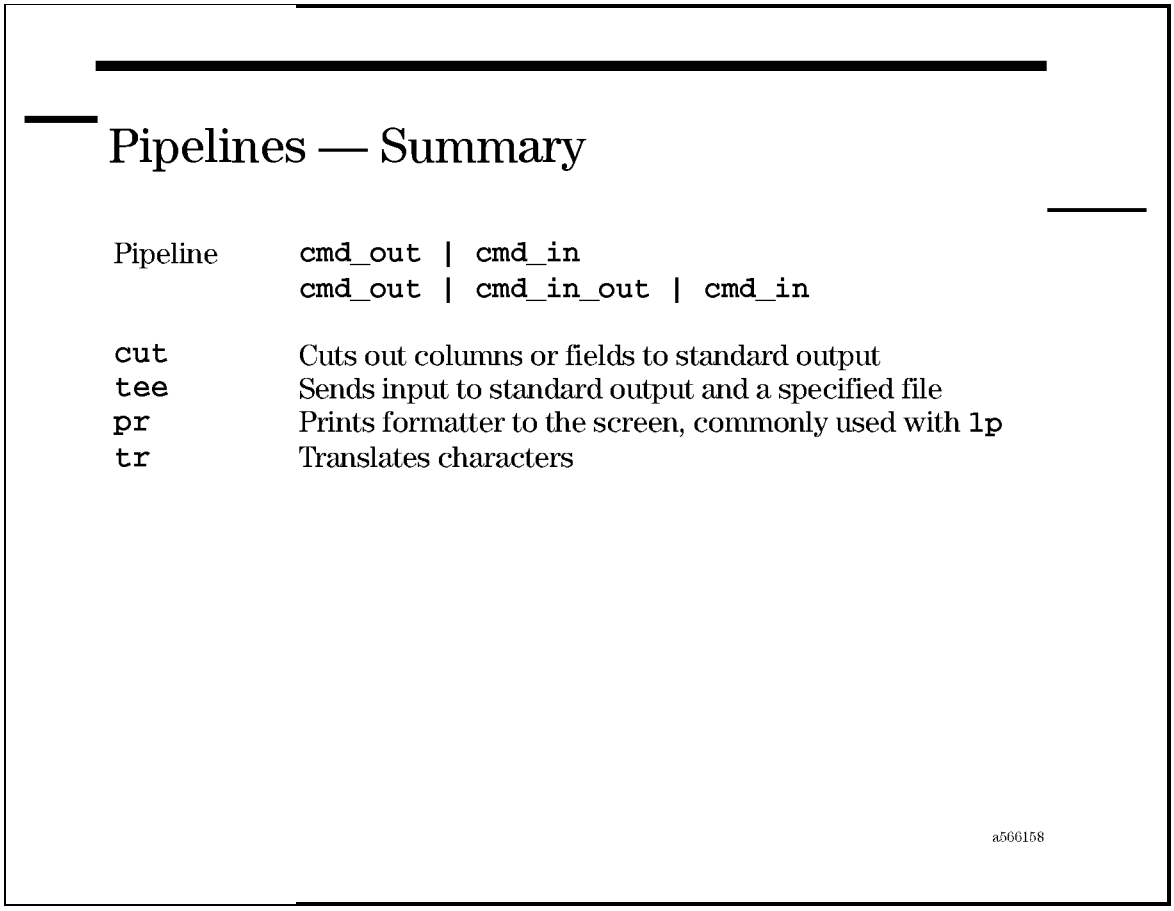
Key Points

- **`lp`** accepts **`stdin`** as well as file names as arguments.
- Any command that generates **`stdout`** can have its output printed.
- Put a **`tee`** in your pipeline if you want to save the output to a file and print the file.
- Printing of standard output is done on a command line by command line basis. There is no switch that you set that directs the output of all subsequent commands to go to the printer.

Teaching Question

How could you send the output to the printer and to your screen?
Answer: Include a **`tee /dev/ttyXX`** in the pipeline. You could include more than one **`tee`** command in one command line.

12-12. SLIDE: Pipelines — Summary



Student Notes

12-12. SLIDE: Pipelines — Summary

Instructor Notes

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** that will display an alphabetical listing of the users currently logged into your system.

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`**.

12-13. LAB: Pipelines

Instructor Notes

Time: 45 minutes

Purpose

To practice constructing pipelines and using some additional filters.

Notes to the Instructor

Lab exercises 1–5 are introductory. Exercise 2 requests the students to search for all lines containing the pattern *user* in **/etc/passwd**. If your user accounts are under **/home**, your students should be able to complete this exercise successfully.

Lab exercises 6–11 are advanced.

Solutions

1. Construct a pipeline that will count the number of users presently logged on.

Answer:

```
$ who | wc -l
```

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.

Answer:

```
$ grep home /etc/passwd | wc -l      Number of lines containing
                                     home
$ grep -v home /etc/passwd | wc -l   Number of lines not containing
                                     home
```

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.

Answer:

```
$ grep home /etc/passwd | tee all.users | wc -l
```

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.

Answer:

```
$ sort names | pr -3 -t
```

5. Create an alias called **whoson** that will display an alphabetical listing of the users currently logged into your system.

Answer:

```
$ alias whoson="who | sort"
```

6. Construct a pipeline to obtain a listing of just the user names of those users presently logged into the system.

Answer:

```
$ who | cut -c1-8

or

$ who | cut -f1 -d" "
```

7. Construct a pipeline to obtain a long listing of just file permissions and file names currently in your working directory.

Answer:

```
$ ll | cut -c2-10,58-
```

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.

Answer:

```
$ ll | cut -c16-24,34-44,58- | tee listing.out | wc -l
```

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**.

Answer:

```
$ cut -f1,3,6 -d: /etc/passwd | sort
                                                    Alphabetical sort
$ cut -f1,3,6 -d: /etc/passwd | sort -n -t: -k 2    Numerical sort
```

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.

Overview of Module 13

Audience

general user — General system users

Product Family Type

open sys — Open systems environment

Abstract

It is very common for a user to purchase a computer to use in a networked environment. This module gives the basic forms of some user LAN services. The students must know the basic HP-UX commands.

This module does not replace the ARPA/Berkeley Services class! It is just an overview of the basic LAN services.

Time

Lab — 30 minutes

Lecture — 30 minutes

Prerequisites

m46m — Navigating the File System

Instructor Profile

UX — General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000 — Logon access to an HP-UX system

Software Requirements

UX11 — HP-UX release 11.0

LAN	LAN software (ARPA, Berkeley Services

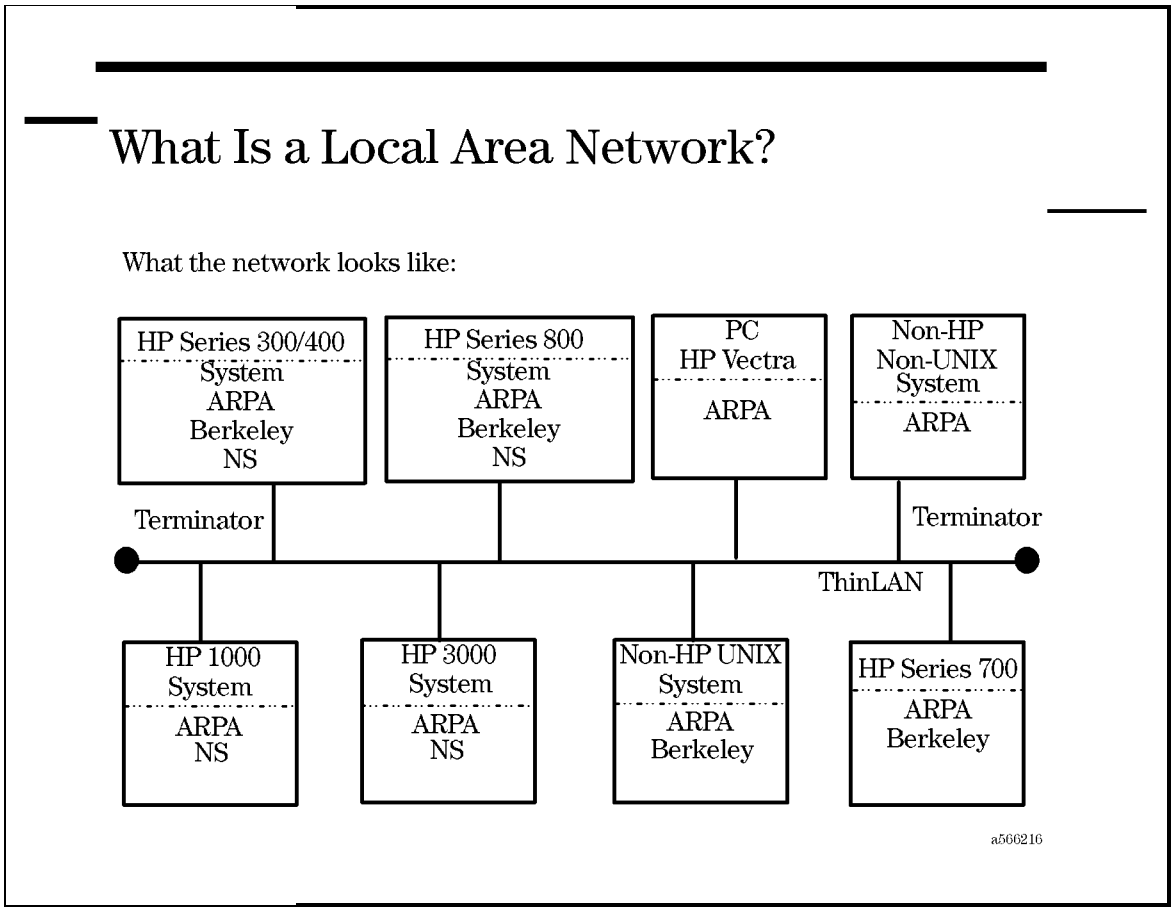
Material List

P/N B2355-90033(T)	*HP-UX Reference Manual*, one per terminal

Lab Instructions

setup1	Create user logon of **`user1`**, **`user2`**, ... **`user`***n*, where *n* is the number of students in the class. Set up one user per student.
lan	If you are in a networked classroom, make sure the students can log in to another system. Make sure the **`/etc/hosts.equiv`** file contains the names of all of your computers on all systems.

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.

13-1. SLIDE: What Is a Local Area Network? Instructor Notes

Purpose

To introduce the kinds of things we can do with a LAN.

Key Points

These are the four basic functions that users will perform on a LAN. Other functions are normally built out of these four.

Presentation Suggestions

If you have a LAN in the training facility, it helps to discuss the setup of the machines.

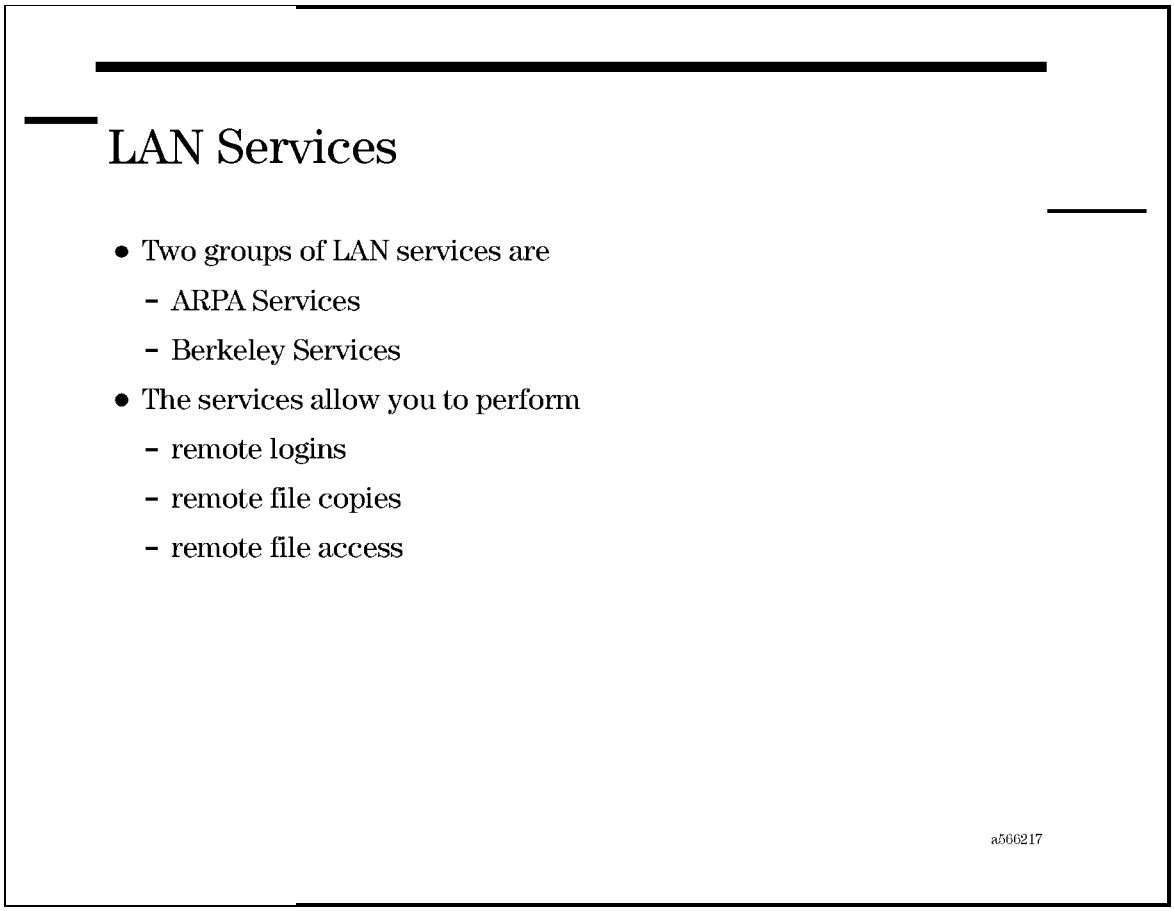
Ask how many students have LAN installations and pick a few to describe why they wanted a LAN in their company. This will help establish the need for a LAN.

Transition

There are two sets of services that you can use to perform these functions:

- ARPA Services
- Berkeley Services

13-2. SLIDE: LAN Services



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.

13-2. SLIDE: LAN Services — Instructor Notes

Purpose

To show the many commands that you can use to perform remote tasks. Which commands the students can use will depend on which of these services they have installed on their computers.

Key Points

ARPA Services are used to communicate between many different types of computers with different operating systems. The Berkeley Services are primarily used between UNIX systems (although they do work on other systems).

Where Problems Arise

Many students have heard about TCP/IP (Transmission Control Protocol/Internet Protocol). This is not a service you can run interactively. Instead, it is the underlying protocol that the network services use to facilitate the communications across the LAN.

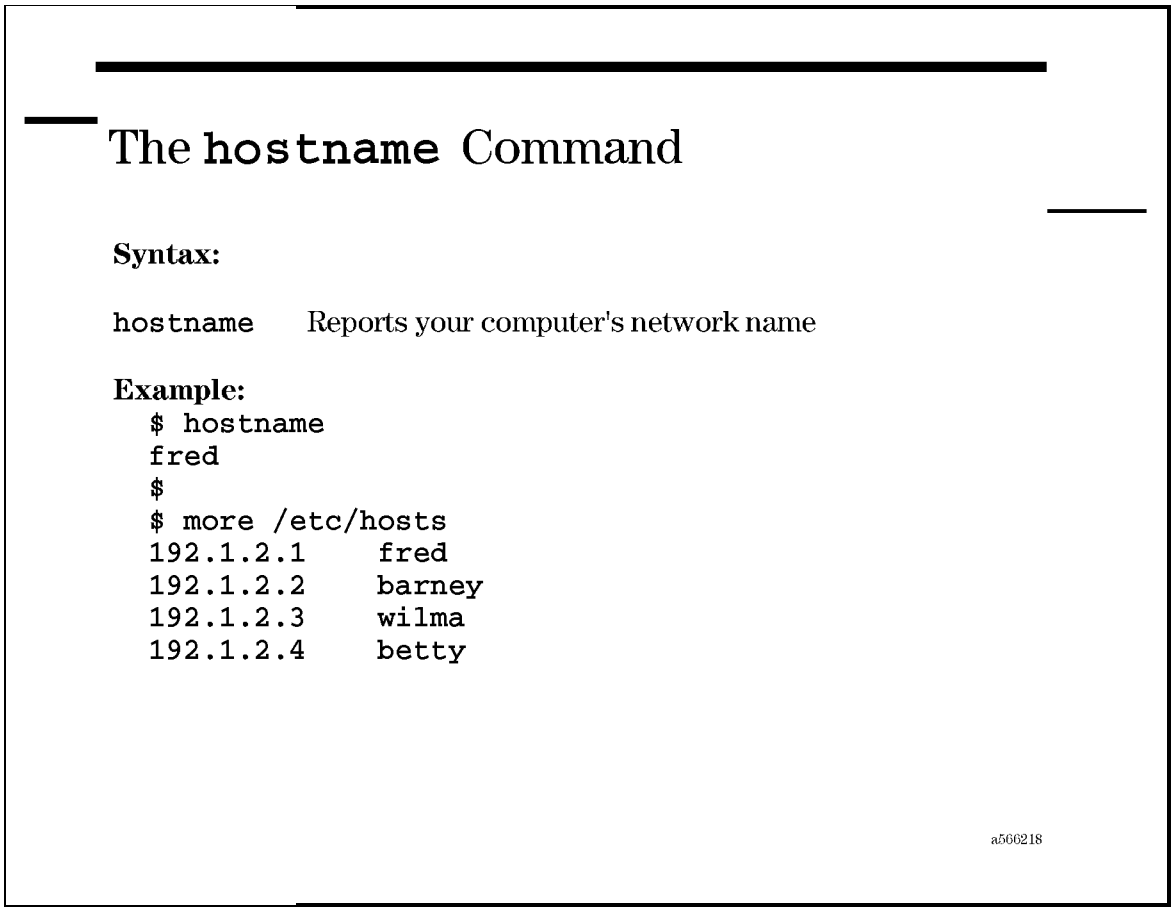
Evaluation Questions

When would you use the ARPA Services or the Berkeley Services? Answer: Use ARPA Services for communicating to computers with different operating systems across a single LAN. Use Berkeley Services for communicating between computers running the UNIX operating system.

Transition

A computer on a LAN is known by its host name.

13-3. SLIDE: The `hostname` Command



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.

13-3. SLIDE: The `hostname` Command — Instructor Notes

Purpose

We communicate to other machines using host names.

Key Points

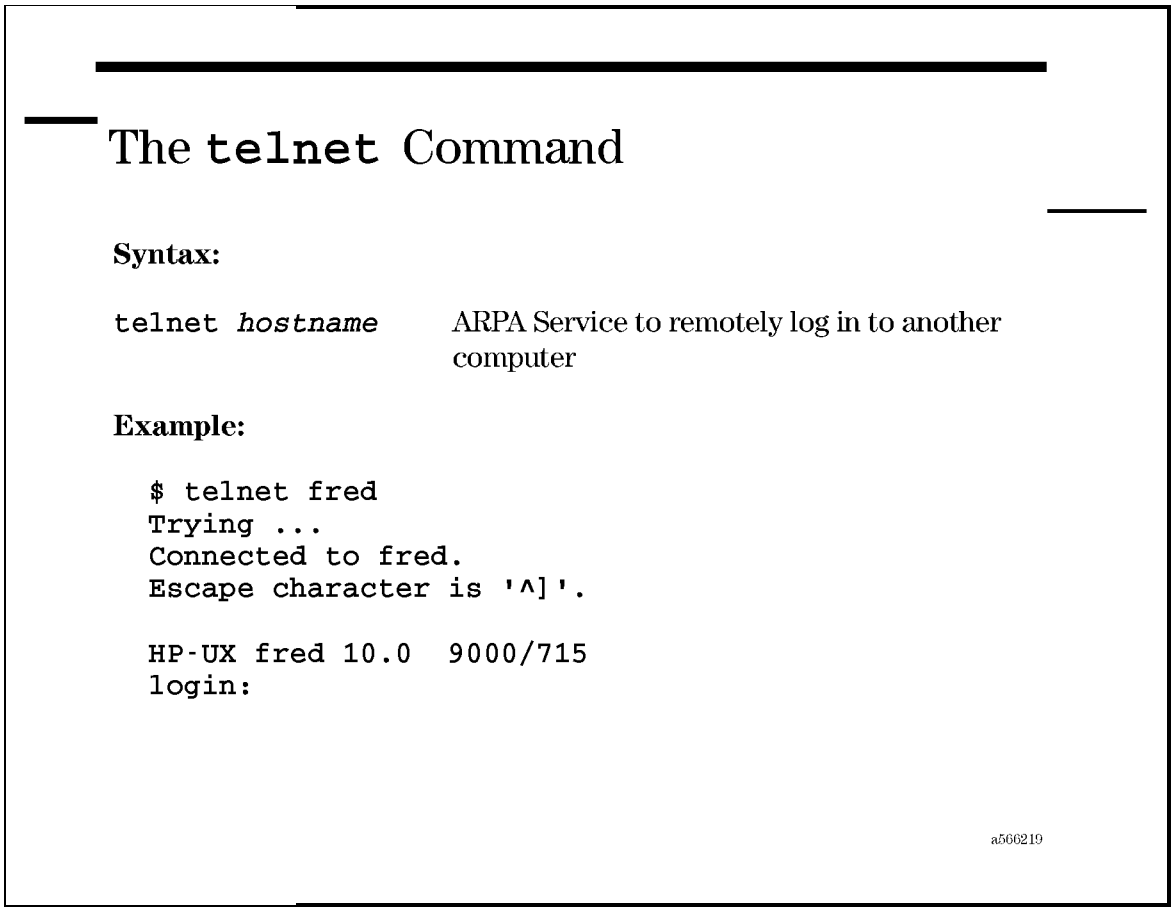
You must be a valid user on the remote computer.

The administrator of the other computer must also allow you access permissions to his or her computer through the network configuration files.

Transition

Let's take a look at the two ARPA Services first.

13-4. SLIDE: The `telnet` Command



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`**.

13-4. SLIDE: The `telnet` Command — Instructor Notes

Purpose

To log in remotely to another computer running the ARPA Services, you would use **`telnet`**.

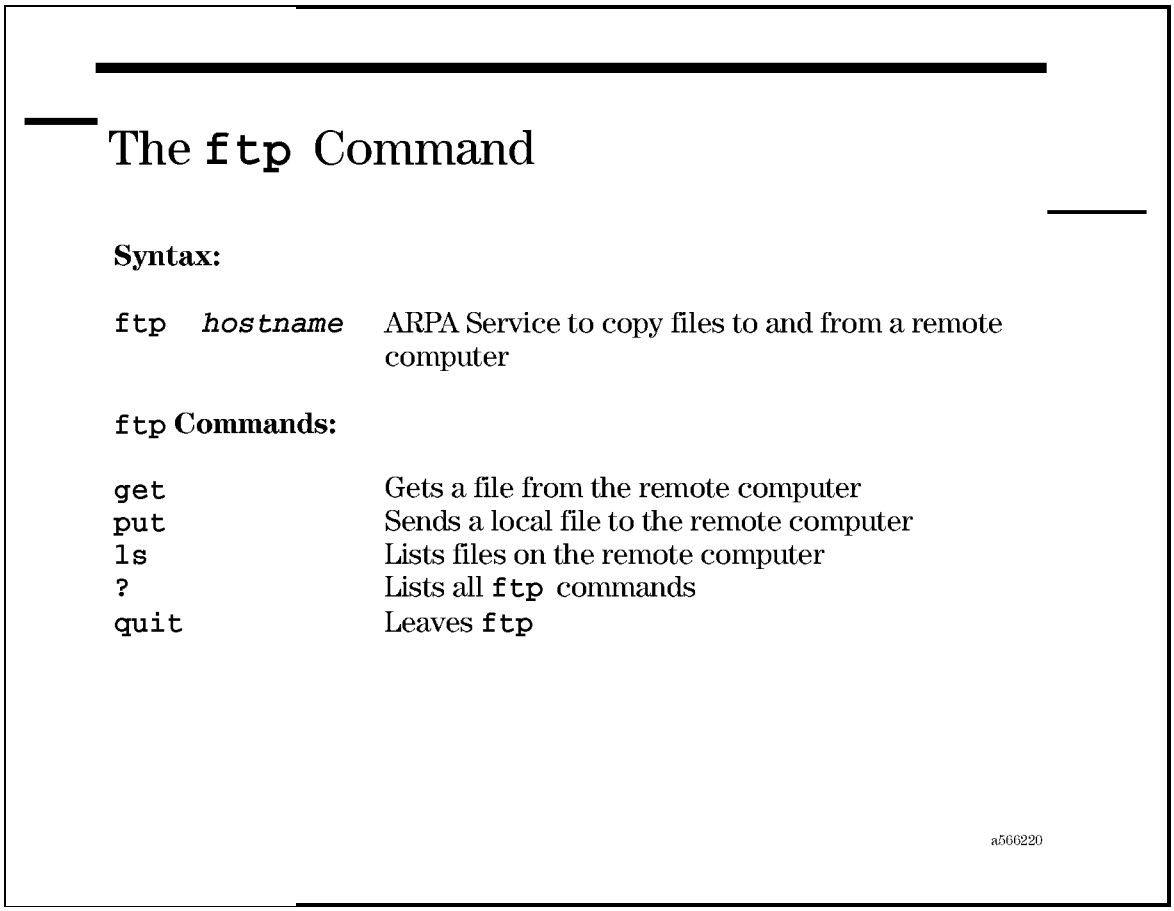
Key Points

`telnet` has many more facilities than we will discuss here. **`telnet`** *hostname* is the most basic form of the command.

Transition

The ARPA Service to perform remote file transfers is called **`ftp`**.

13-5. SLIDE: The `ftp` Command



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.
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!)

13-5. SLIDE: The `ftp` Command — Instructor Notes

Purpose

If you use ARPA Services, you will use **`ftp`** to transfer files.

Key Points

`ftp` requires that the user have a password set on the remote computer.

There are many **`ftp`** commands to facilitate remote file transfers. **`?`** will list the commands.

Where Problems Arise

`ftp`'s output can be very confusing to the first-time user. The example shows a typical **`ftp`** session.

Presentation Suggestions

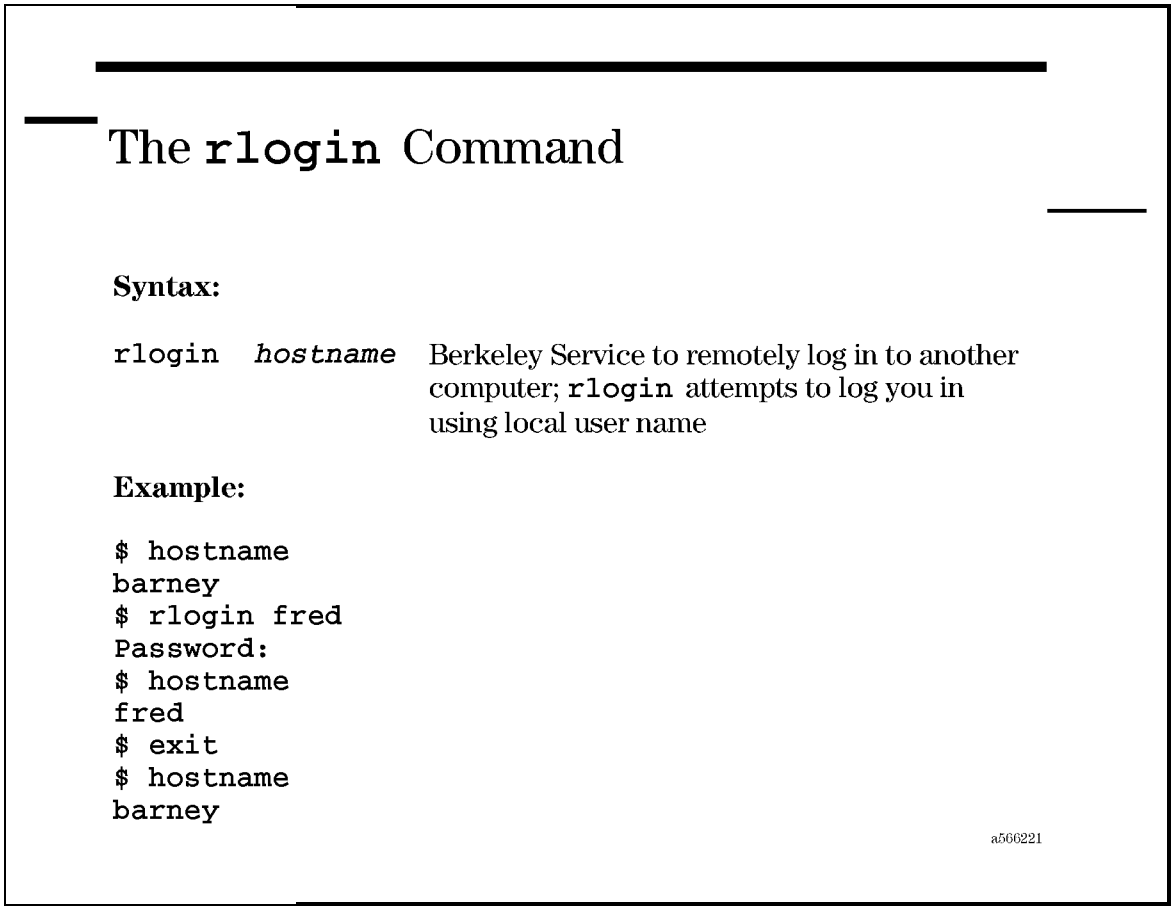
Go through the example, explaining that the messages are not error messages. They are **`ftp`**'s way of telling you what it is doing.

You may want to explain a **`get`** example also.

Transition

The Berkeley Services have commands to perform similar tasks.

13-6. SLIDE: The `rlogin` Command



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.

13-6. SLIDE: The `rlogin` Command — Instructor Notes

Purpose

`rlogin` is usually the preferred way to log in remotely to another UNIX system computer because it is easier to manipulate than **`telnet`**.

Key Points

The file **`/etc/hosts.equiv`** is a list of host names that have the same users. If your login name is **`gerry`** on one system, it is assumed that you are the same **`gerry`** on all hosts listed in **`hosts.equiv`**. This is helpful when **`rlogin`** is used, and it *must* be configured correctly when using **`remsh`** and **`rcp`**. You may want to mention **`$HOME/.rhosts`** as an alternative to **`hosts.equiv`**.

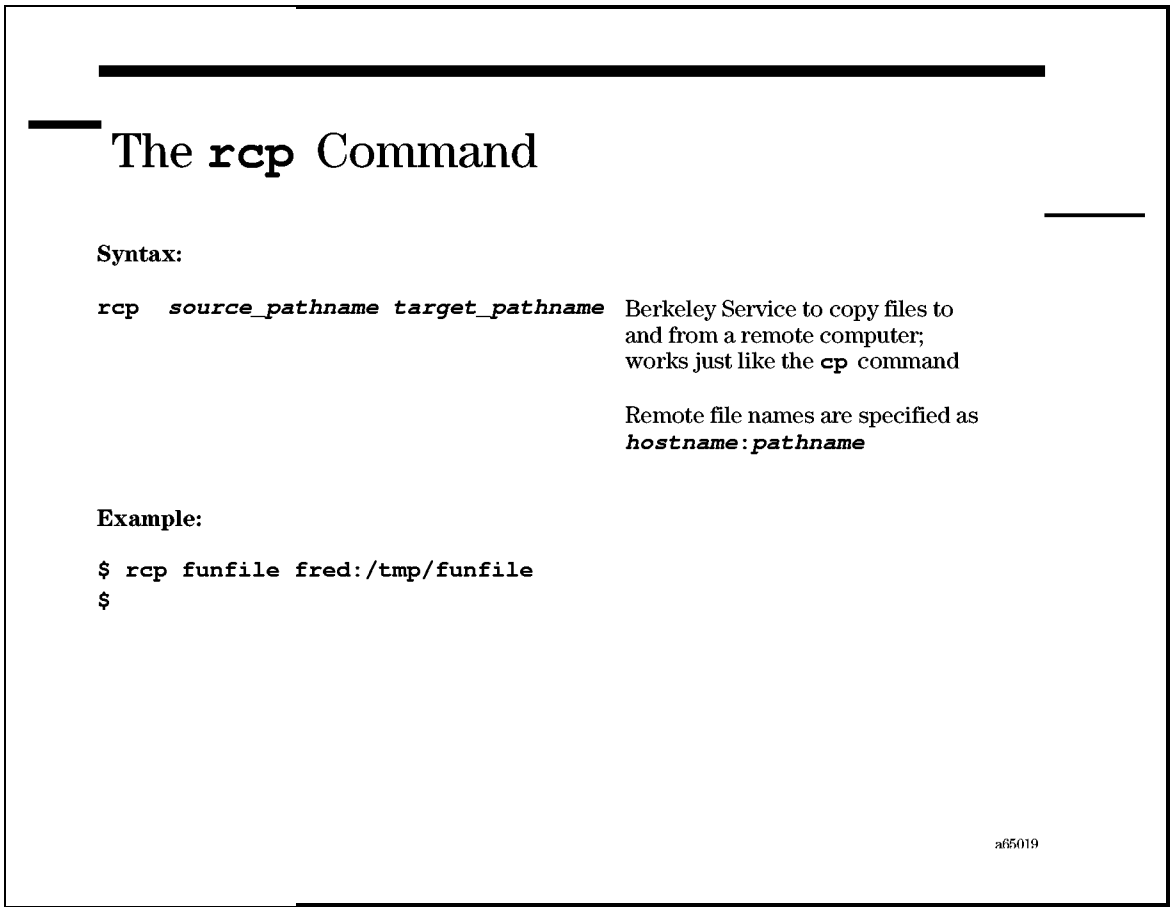
Evaluation Question

Why would you use **`rlogin`** instead of **`telnet`** and vice versa? It depends a great deal on what services you and the remote computer have in common.

Transition

Let's take a look at how to do remote file transfers using the Berkeley Services.

13-7. SLIDE: The `rcp` Command



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
```

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.

13-7. SLIDE: The `rcp` Command — Instructor Notes

Purpose

`rcp` is a simple way to copy files between UNIX systems.

Key Points

Each system (**`bambam`**, **`fred`**, and **`barney`**) should have a **`hosts.equiv`** file that contains all three names.

You can leave off **`hostname:`** if the file is on the local machine.

`rcp` is easier to use than **`ftp`**; however, **`rcp`** is much slower if you are transferring a large number of files between two machines.

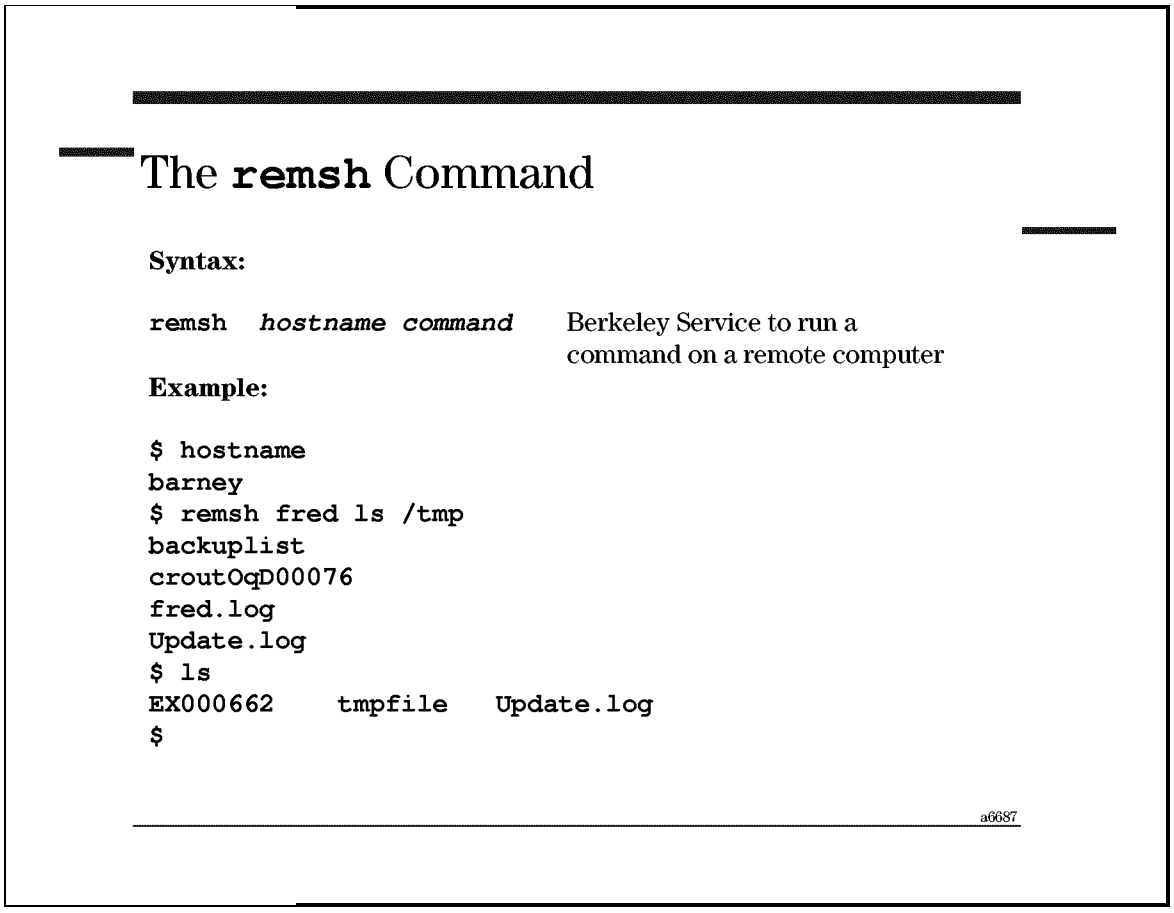
Presentation Suggestions

You could show the students a few more examples, if necessary.

Transition

The next Berkeley Service that we will look at is the **`remsh`** command.

13-8. SLIDE: The `remsh` Command



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 wanted to see what is running on the system **fred**, you could execute

```
$ remsh fred ps -ef
```

List the files in **fred**'s **/tmp** directory:

```
$ remsh fred ls /tmp
fredfile
funfile
reconfig.log
update.log
```

Or, if you wanted 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.

13-8. SLIDE: The `remsh` Command — Instructor Notes

Purpose

`remsh` can be used in many ways to facilitate two machines working together across the LAN. It is an extremely powerful tool when used in shell programs.

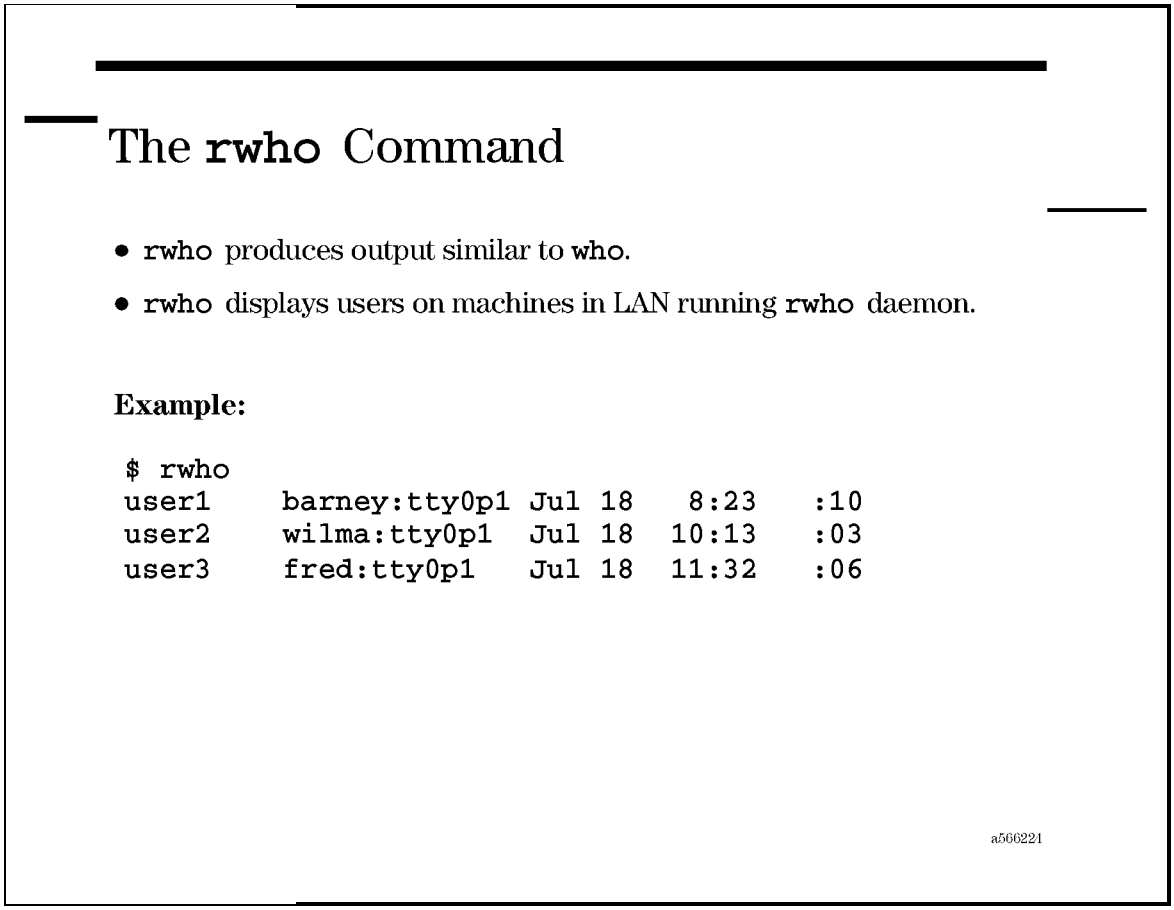
Key Points

You may want to quote the command to be executed remotely to prevent the local shell from interpreting any special characters.

Transition

The Berkeley Services have commands to perform similar tasks.

13-9. SLIDE: Berkeley — The rwho Command



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.

13-9. SLIDE: Berkeley — The rwho Command Instructor Notes

Key Points

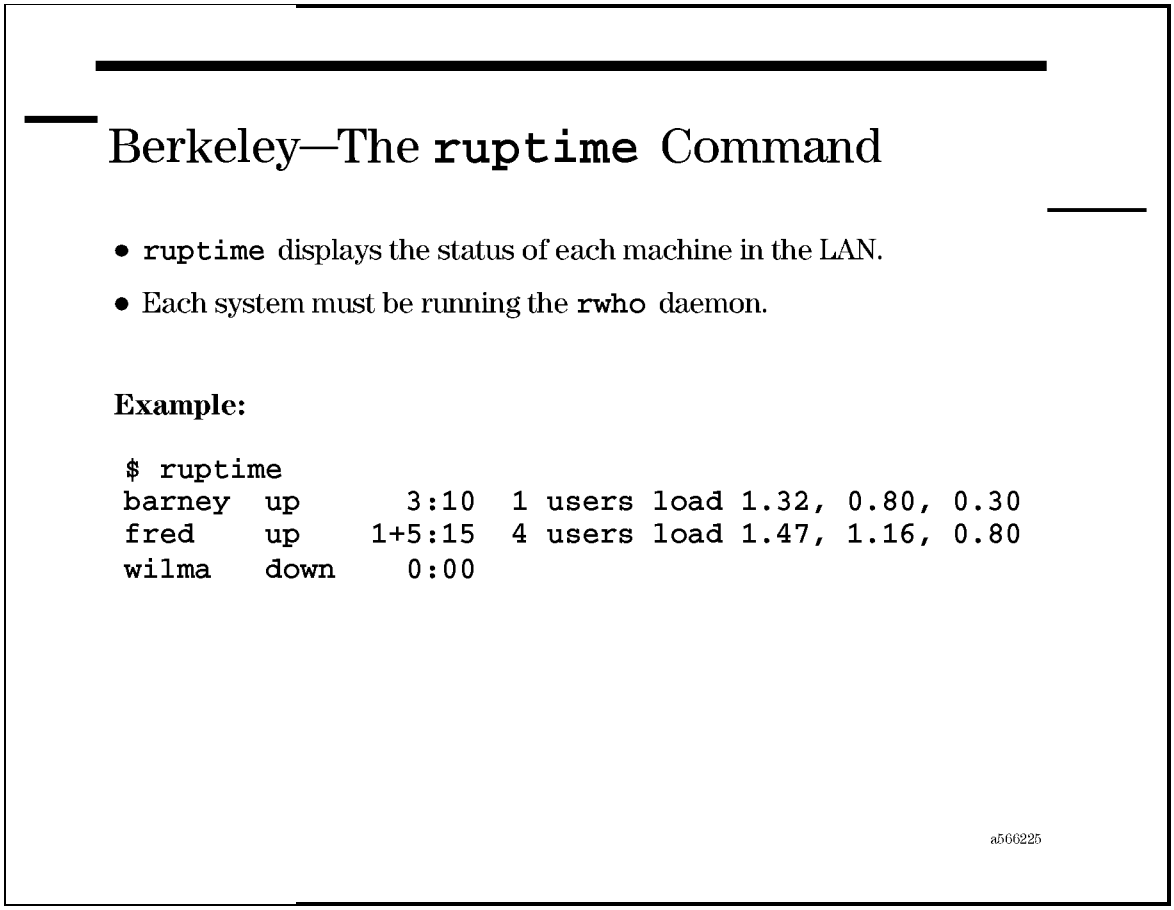
- This is a useful command to see if someone else is currently logged in on your network.

Teaching Tips

The **rwho** daemon, **rwhod**, is started at boot time through **/etc/rc.config.d/netdaemons**.

You can run **ps -ef | more** to see if the **rwhod** daemon is running.

13-10. SLIDE: Berkeley — The `ruptime` Command



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.

13-10. SLIDE: Berkeley — The `ruptime` Command

Instructor Notes

Transition

Let's try a few exercises.

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.

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.

13-11. LAB: Exercises — Instructor Notes

Purpose

To practice using the basic LAN services.

Notes to the Instructor

If you are not in a network, the students can still do the exercises on the local system. Just be careful of which file names get used in the lab.

Solutions

1. Use the **hostname** command to determine the name of your local system. What systems can you communicate with?

Answer:

The **hostname** command reports the local host name. By looking at the **/etc/hosts** file, you can see all of the computers your local computer can talk to.

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.

Answer:

```
$ telnet fred
Trying...
Connected to fred
Escape character is '^]'.

HP-UX fred 10.00 B 9000/715

login: enter your name
Password: enter your password
.
.
.
$ hostname
fred
$ exit
```

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.

Answer:

In **ftp**, you would use the **put** command, similar to the example given in the student notes.

4. Use **remsh** to list the contents of the remote directory to verify that the copy worked.

   Answer:

   ```
   $ remsh system ls
   ```

   The **ls** command will list your *HOME* directory on *system*.

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.

Overview of Module 14

Audience

general user — General system users

Product Family Type

open sys — Open systems environment

Abstract

This module is designed as an *introduction* to the **`vi`** editor. We present **`vi`** because it is a powerful display-oriented editor that is readily available on almost all UNIX systems. By the end of the module students will be familiar with the basic capabilities and commands offered by **`vi`**.

The commands are presented on a category by category basis. The slides will generally illustrate the use of a specific command, and a table is provided in the text that summarizes the common commands defined for that category.

A final summary slide is presented so that the students can pull it out and use it as a quick reference guide.

Time

Lab — 60 minutes

Lecture — 60 minutes

Prerequisites

m47m — Managing Files

In order to successfully complete this module, the student must be able to log in and navigate the file system.

Instructor Profile

UX — General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000	Logon access to an HP-UX system

Software Requirements

UX11	HP-UX release 11.0

Material List

P/N B2355-90033(T)	*HP-UX Reference Manual*, one per terminal

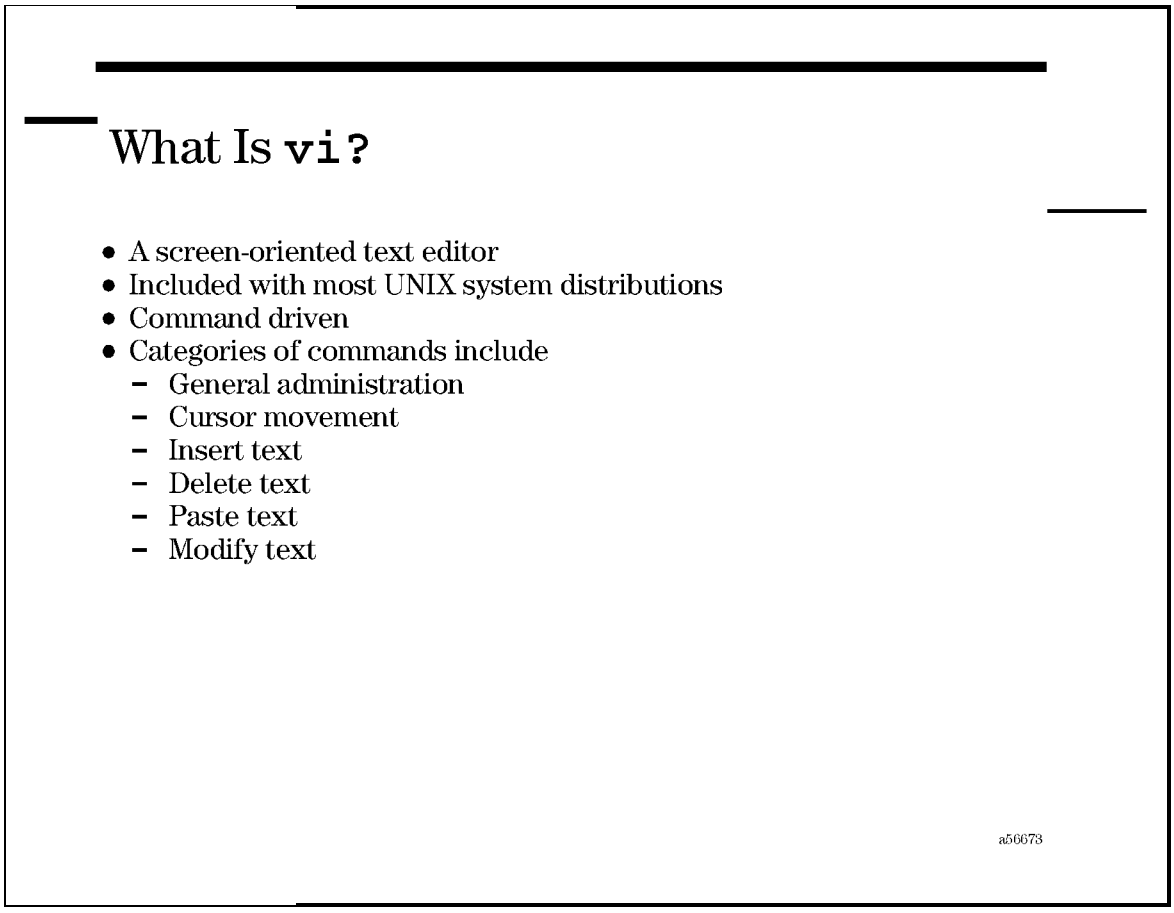
Lab Instructions

setup1	Create user logon of **`user1, user2, ... user`** *n*, where *n* is the number of students in the class. Set up one user per student.
copyfiles	Copy the lab files to the users' home directories.

Lab Files

```
-rw-r-----   1 karenk  users   3081 May 28 16:12 funfile
-rw-r--r--   1 karenk  users     85 May 28 16:12 tst
-rw-r--r--   1 karenk  users    959 Aug 24 12:04 vi.tst
```

14-1. SLIDE: What Is `vi`?



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.

`vi` was developed at the University of California at Berkeley by William Joy. 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 these design philosophies is that **`vi`** can be used on almost any type of terminal, and since 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.

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 to enhance your skills at your own pace.

14-1. SLIDE: What Is vi? Instructor Notes

Key Points

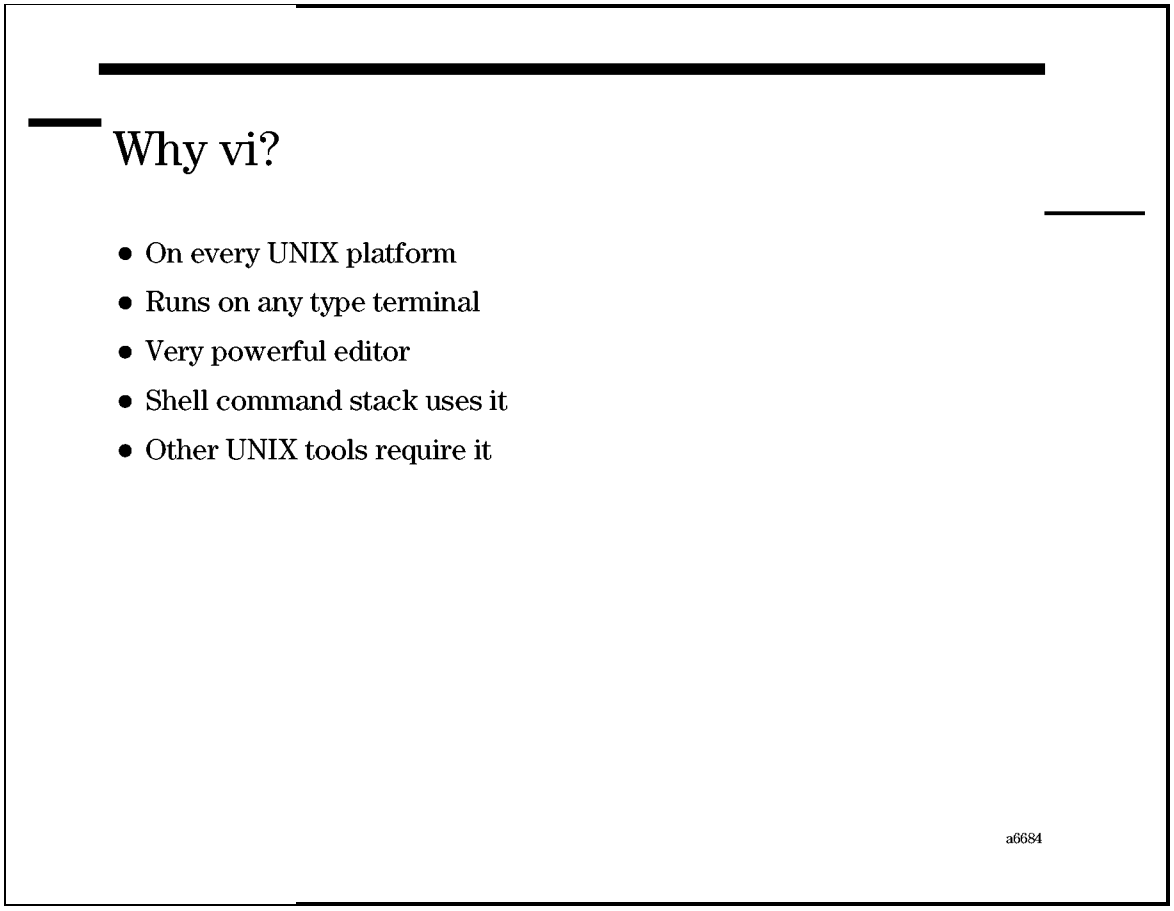
- **vi** is a command driven editor. It often takes time for students to become familiar with the **vi** commands. Most commands are a mnemonic with an associated meaning, so reinforce the commands by presenting their associated meanings.
- With some practice most students will be able to master the basic editing skills.
- Since **vi** is available on most UNIX systems, you only have to learn *one* editor for *all* UNIX systems. It is usually provided at no extra cost.

Teaching Tips

Students often take quite a while to warm up to **vi**. It is important to keep a positive attitude. Point out some of **vi** 's positive attributes, such as

- It is *included* with most UNIX system software distributions.
- It is terminal independent.
- The keyboard is easily configurable.
- Keyboard macros can be defined for common operations.

14-2. SLIDE: Why `vi`?



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.

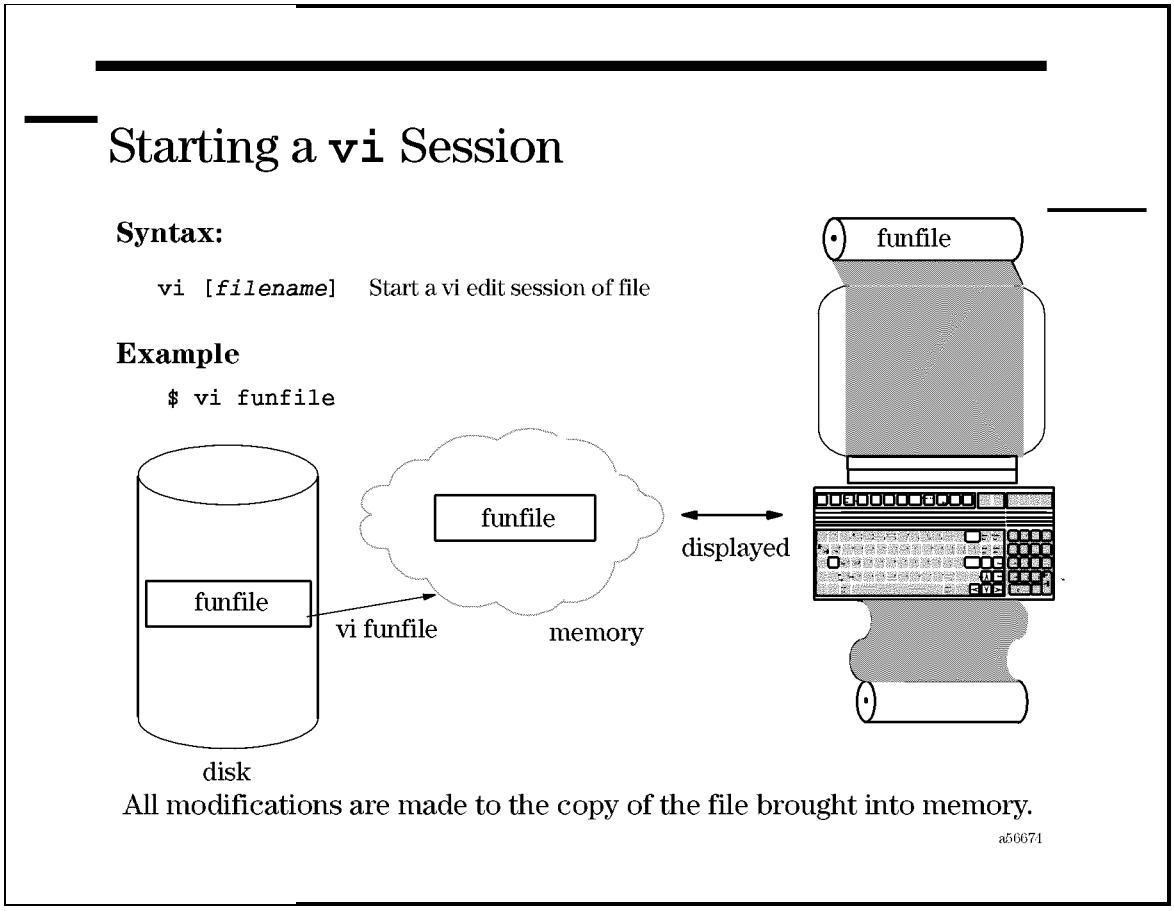
If a large shell program is to be edited, experienced users of **`vi`** find they are more productive with **`vi`** than a windows editor. Although very cryptic, **`vi`** is a very efficient and powerful editor, able to edit multiple files at one time and quickly cut and paste text from one file to another.

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.

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.

14-2. SLIDE: Why vi? Instructor Notes

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.

14-3. SLIDE: Starting a vi Session — Instructor Notes

Key Points

- Invoking **vi** brings a *copy* of your file into memory.
- All modifications are performed on the image in memory.
- Changes can be saved or discarded.
- You will see a *window* of the file.
- Edits normally take place at the cursor. *Focus on the cursor.*
- Changes will be seen immediately on the screen.

Teaching Tips

You might want to mention that a temporary file is also maintained, in case of a power outage during an editing session. The temporary file is *automatically* updated as you proceed through your editing session. If you experience a power outage while editing, **vi** will mail you a message informing you that it has saved your temporary file. To recover the *crash* file you invoke: **vi -r** *filename*.

Activity

If you wish to talk students through the next few slides, you may wish to have them begin a **vi** session on the file **funfile** in their home directory.

```
$ cd
$ vi funfile
```

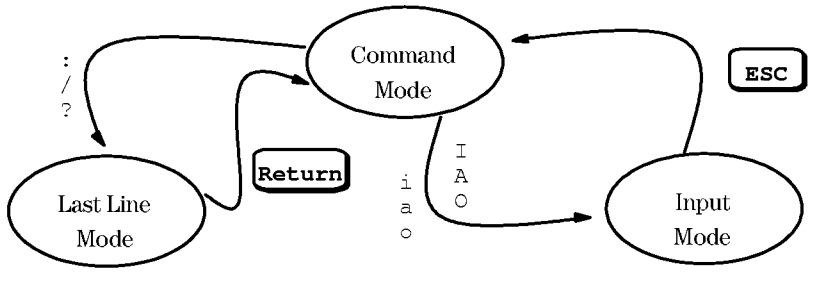
Common Problems

Students will often enter **vi** Return and not provide a file name. If this occurs, you may have to help them exit **vi** and reissue the command specifying the file **funfile**.

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.

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.

14-4. SLIDE: vi Modes — Instructor Notes

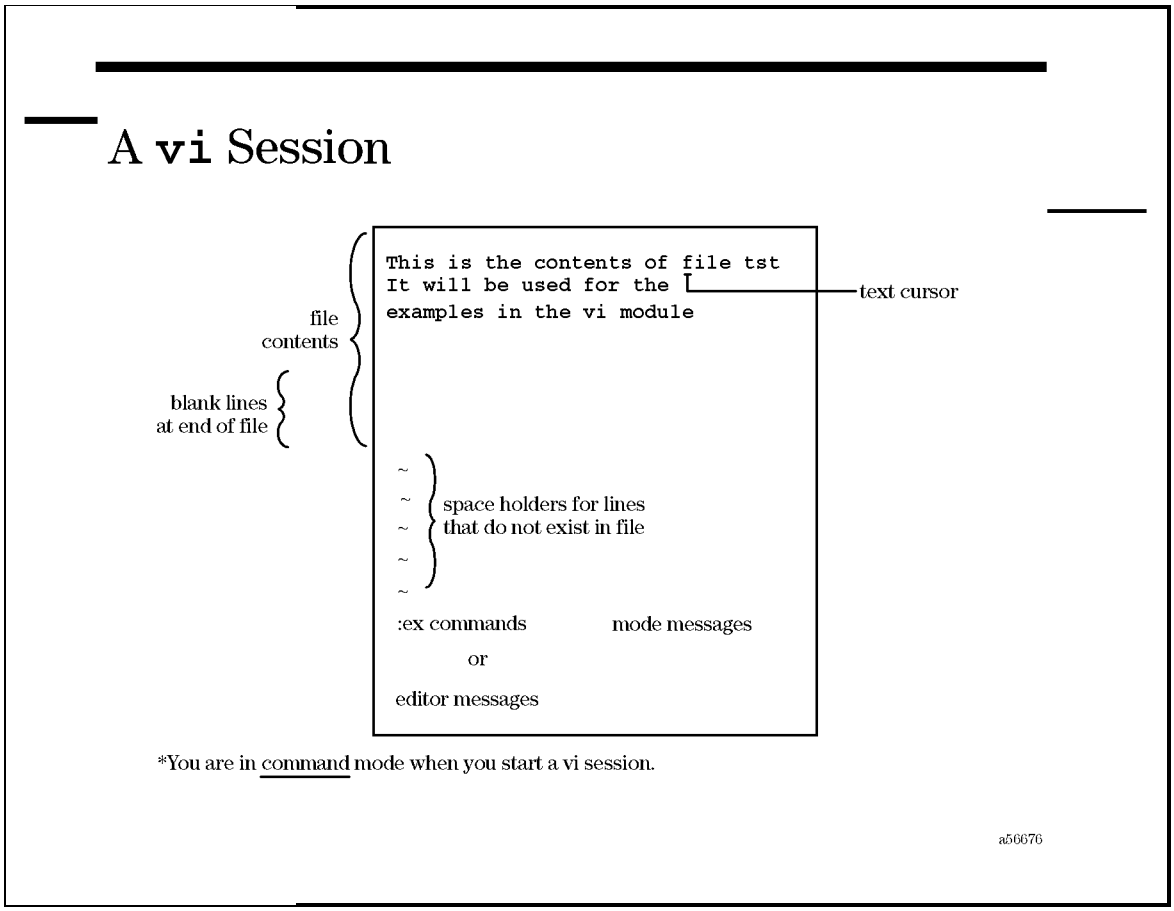
Key Points

- **vi** is a command-driven editor.
- When in *command mode*, everything is interpreted as a command.
- You are in command mode when you start a **vi** session.
- When in *input mode*, everything you type is entered into the file.
- You press the ESC key to return to *command mode* or terminate a command.
- **ex** commands are entered at a colon prompt.
- **ex** commands are echoed to the screen, unlike **vi** commands.
- **ex** commands are available for batch line edit operations, and session customization.

Activity

If you are guiding students through the basic commands, have the students press a couple of keys, like **j** or several **w**'s to see that these characters are not entered into their file because they are in command mode. **vi** is going to try to execute the **j** and the **w** command.

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.

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.

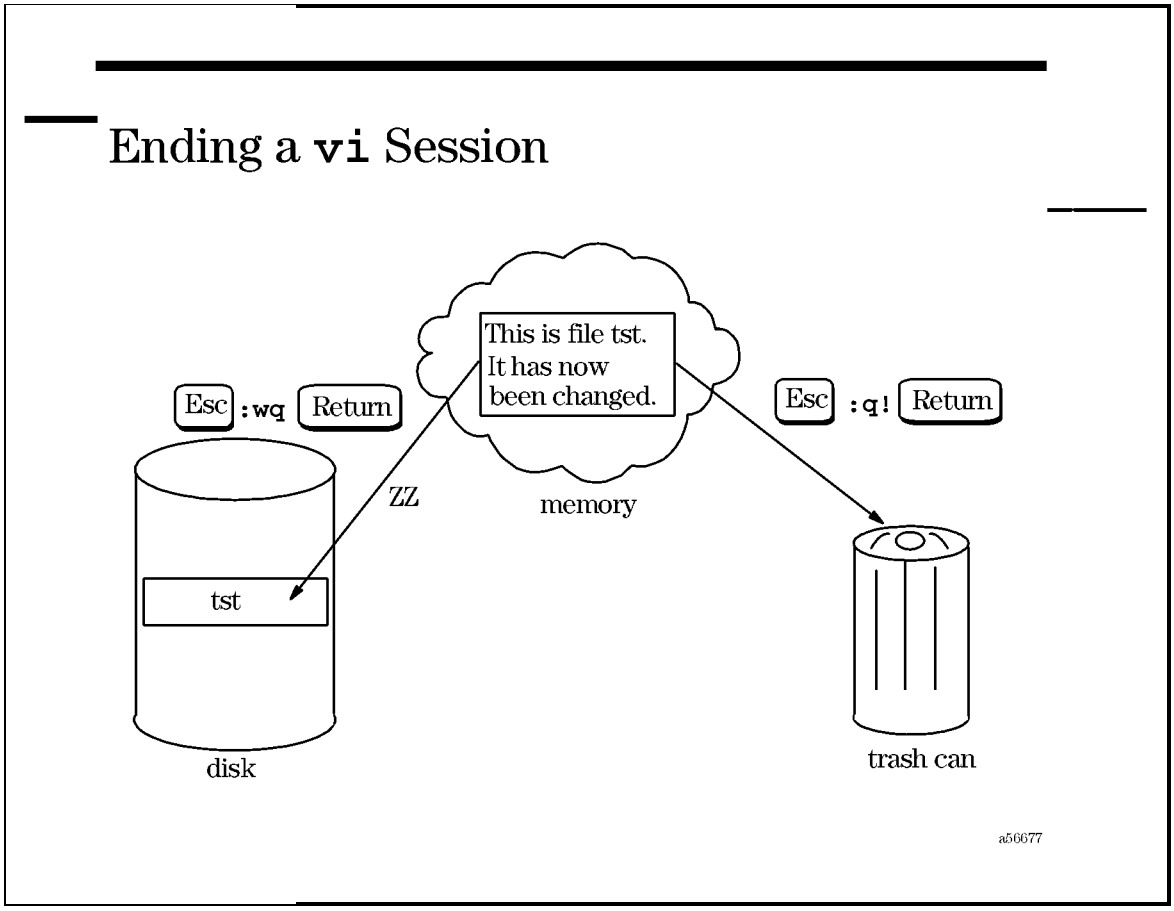
NOTE: If you go into *input* mode and do not see a mode message signifying *input mode*, enter:
ESC **`:set showmode`** Return

14-5. SLIDE: A vi Session — Instructor Notes

Key Points

- Present the components of the **vi** display.
- Remind students that they are in *command mode* when they start a session.
- Point out the tildes, and that they are *not* part of the file.
- Remind students to observe messages in the *mode messages* area, so that they can tell when they are in *input* or *command* mode.
- Changes usually take place at the cursor, so this is where students should focus their attention.

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.)*

14-6. SLIDE: Ending a `vi` Session — Instructor Notes

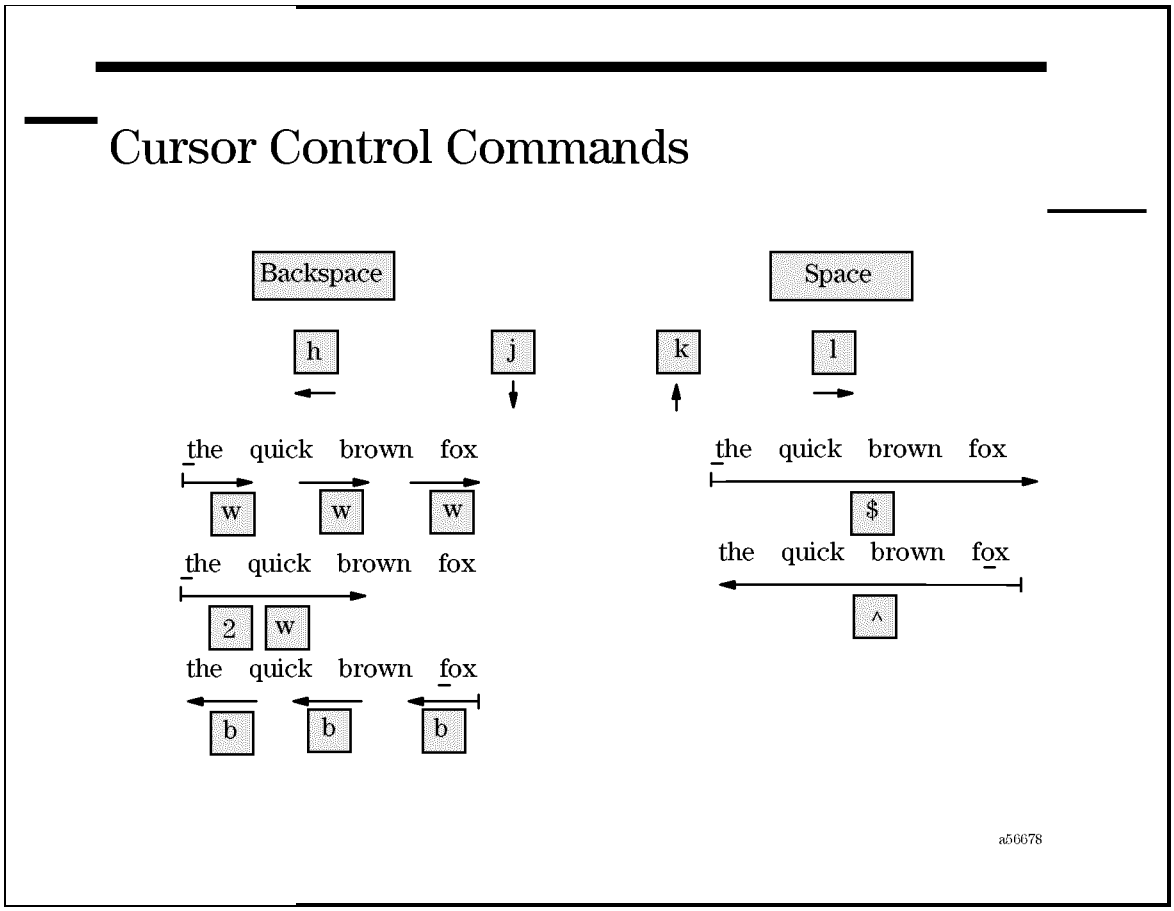
Key Points

- Once you know how to get into **`vi`** , you should know how to get out.
- Remember you must be in *command mode* to issue these commands.
- It is important to present the **`:q!`** command, especially early on, so students can discard their changes, and keep their file intact if they wish.

Activity

Have the students quit the **`vi`** session without saving any changes. (Hint: ESC **`:q!`**) Notice that the **`:q!`** is echoed to the lower left corner of the screen. Any command prefixed with a colon will appear here.

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.

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.

14-7. SLIDE: Cursor Control Commands Instructor Notes

Teaching Tips

- Remind students to *focus on the cursor position* as they edit their files.
- Most students will be inclined to use the cursor keys, and will probably ask if they can be used. If a **vi** session has been configured to recognize the cursor keys (often through **.exrc**), they can be used. But students should not depend on the cursor keys until they know if they are available.
- The cursor keys are not recognized when using the **vi** commands to edit the Korn Shell command stack. Therefore, you might want to discourage the use of the cursor keys until the students become more familiar with the cursor control commands.
- A trick to remember the **h** and **l** commands is their relative position on the keyboard. The **h** key is the leftmost cursor control key, and moves the cursor to the left, the **l** key is the rightmost cursor control key, and moves the cursor to the right. Students usually become more familiar with the **k** command later, after they use it to move *up* through the command stack.

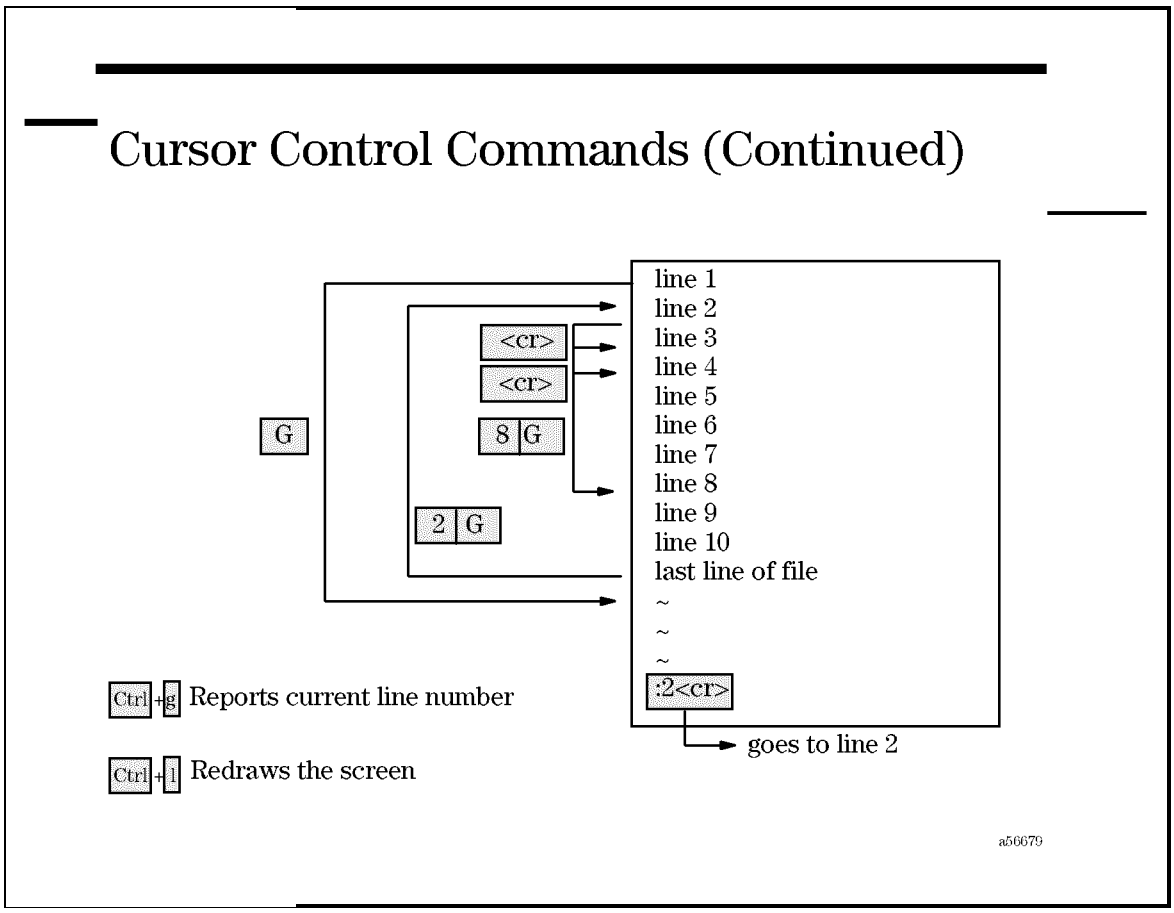
Activity

You may wish to have the students practice using the cursor control commands listed on the slide in the file **funfile**.

Possible Problems

- Users getting to the last character on a line, and pressing the *right cursor* control command. If there is no text (embedded blank spaces), the cursor will not move to the right.
- Users getting to a blank line, and similar to the above, trying to advance the cursor to the right. If there is no text (embedded blank spaces), the cursor will not move to the right.

14-7. 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.
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

14-7. SLIDE: Cursor Control Commands (Continued) — Instructor Notes

Key Points

The commands on these two pages allow the user to *efficiently* move through the text in his or her file.

Activity

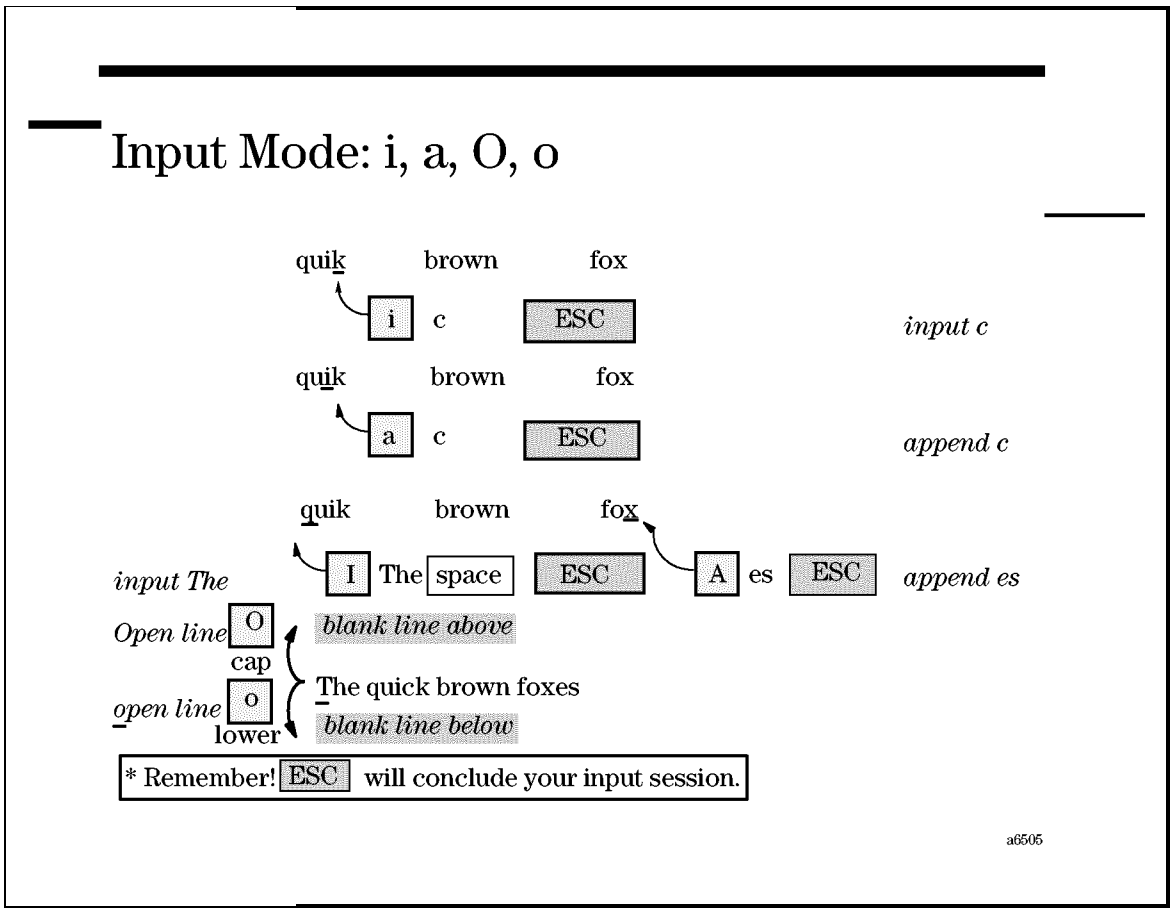
Have the students use the cursor control commands to move the cursor to

the next line	Return
line 10	`10G`
the last line	`G`
the first line	`1G`

Possible Problems

- Users moving to the last line of a file, seeing the tildes and pressing the Return trying to advance to the next line. The tildes do *not* represent empty lines in the file.

14-8. 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.

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.

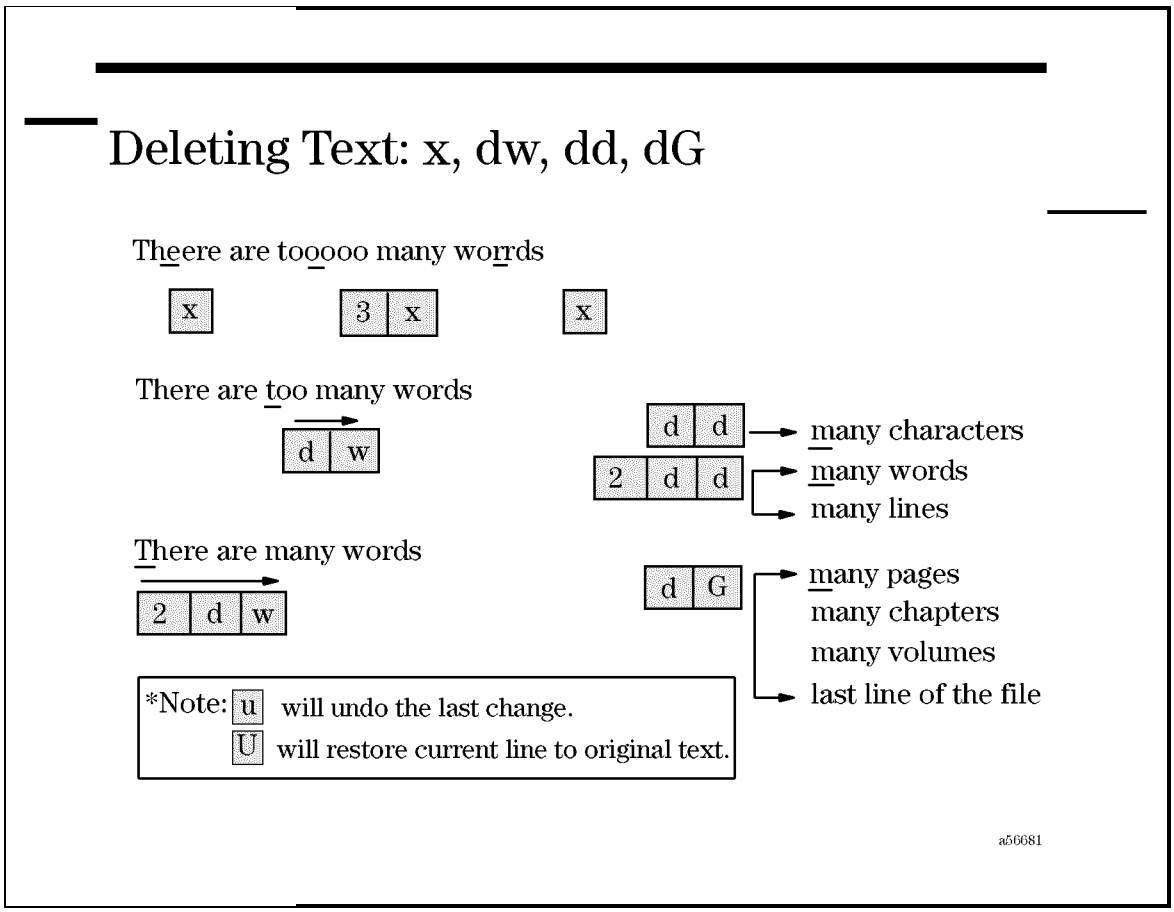
You can use Backspace to correct typing mistakes for the *current* input session. Pressing the ESC key concludes an input session. To modify text that was entered in a previous input session you must use **`vi`** commands.

14-8. SLIDE: Input Mode: i, a, O, o Instructor Notes

Key Points

- New text can only be input from *input mode.*
- Students must be careful when using the Backspace key to correct typing mistakes.
- ESC will return you to *command mode.* You are constantly toggling between the two modes.

14-9. 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.

`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*.

14-9. SLIDE: Deleting Text: x, dw, dd, dG Instructor Notes

Key Points

- **x** is used to delete single characters.
- **d** is used to delete objects, which are defined through the motion control commands.
- Focus on the cursor position because objects are commonly defined relative to the current cursor position.
- **u** allows users to undo their last change.
- **U** allows users to recover the current line.

Teaching Tips

The slide shows several examples of the **x** and **d** commands, showing two progressive applications of the commands.

Use a shield to cover the 2nd column and present the first column examples. The first column is character- and word-oriented.

Use a shield to cover the 1st column and present the second column examples. The second column is line-oriented.

Activity

You may wish to stop at this point and have students to the first lab, "Adding and Deleting Text and Moving the Cursor."

14-10. 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`).

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?

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`**.

14-10. LAB: Adding and Deleting Text and Moving the Cursor — Instructor Notes

Time: 30 minutes

Purpose

To become familiar with **`vi`** commands that move the cursor and toggle between *command mode* and *input mode*.

Solutions

1. **`vi`** the file **`tst`**.

   Answer:

   $ **`vi tst`**

2. Insert the word *only* between the words *will be*.

   Answer:

   Move cursor to the second line, type **`j`** or Return or **`2G`**.
   Move cursor to the last *l* in *will*, type **`ee`** or **`2e`** or several **`l`**s.
   Append text after the **`l`**, type **`a`**; type **`only`** or
   Move cursor to the first *b* in *be*, type **`ww`** or **`2w`** or several **`l`**s.
   Insert text before the *b*, type **`i`**; type **`only`**.

3. Add the words *many, many* on the end of the line *It will be used for*.

   Answer:

   Go back to command mode, type ESC.
   Move the cursor to the end of the line, type **`$`**.
   Add (append) the text, **`a`**; enter text **`many, many`**.

4. Add a new blank line at the end of the file, and enter your name. DON'T PRESS THE ESC !

   Answer:

   Go back to command mode, type ESC.
   Move cursor to the last line: **`G`**.
   Open a new line below, **`o`**.
   Type your name.

5. Using the Backspace, remove your name and enter your partner's name.

   Answer:

   Backspace over the first name.

Type in your partner's name.

6. Open a new line at the top of your file (Hint: `O`).

Answer:

Go back to command mode, type ESC.
`1GO`

7. Enter `12345`.

Answer:

`12345`

8. Backspace 2 times. Do any of the numbers disappear from your display?

Answer:

Backspace Backspace
The cursor will be under the 4. No characters disappear.

9. Enter `1234`. What happens to the numbers that you backspaced over?

Answer:

The 4 and 5 will be typed over.

10. Backspace 3 times.

Answer:

Backspace Backspace Backspace
The cursor is now under the second 2.

11. Press ESC . What happens to the characters you backspaced over? Where does the cursor end up?

Answer:

The second 234 will disappear, and the cursor will back up so it is under the 1.

12. Type in 4 `a`'s. How many a's appear? Why?

Answer:

Three a's appear because the first a is taken to be the **`vi append`** command.

13. Backspace 5 times. What happens? Why?

Answer:

You can only Backspace three times, because you have only entered 3 letters in this input session.

14. Press ESC. What happens?

    Answer:

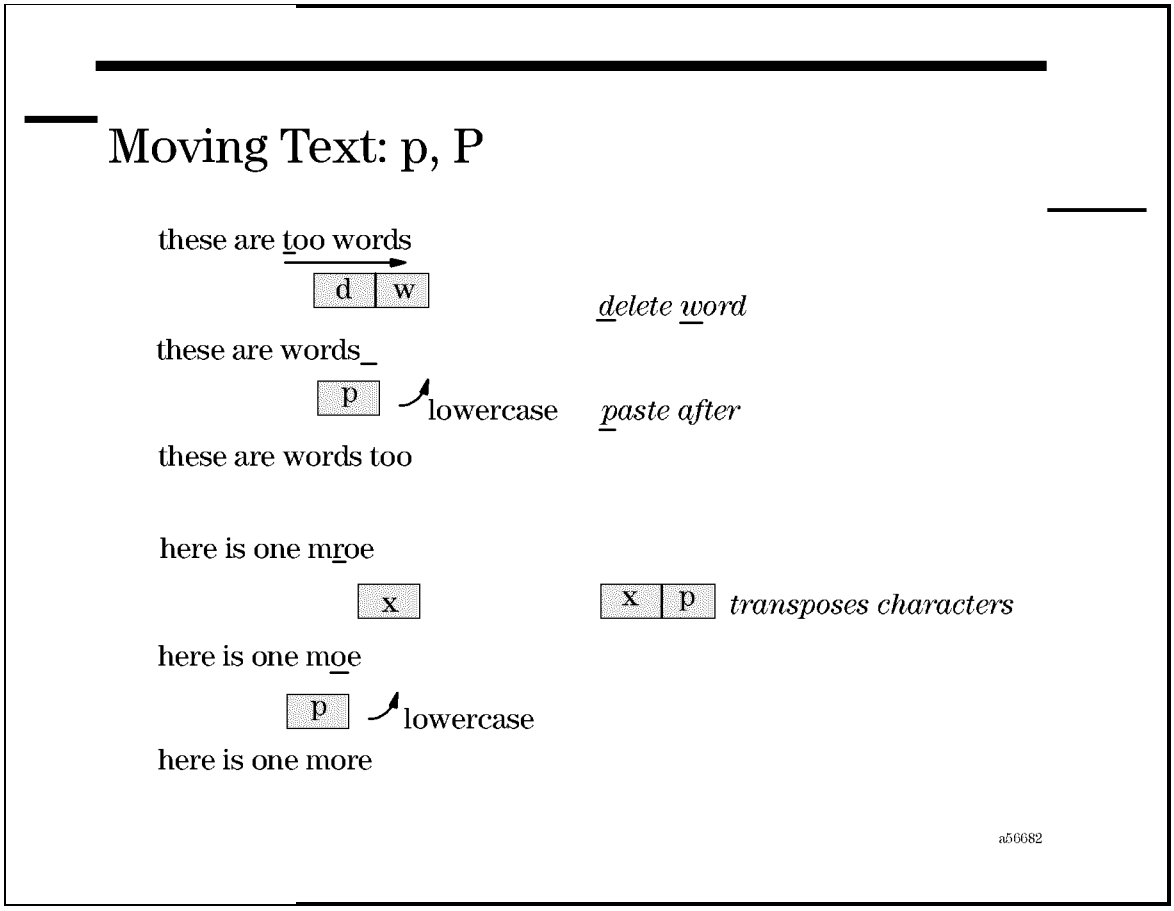
    All of the a's you just entered disappear. You are back in command mode.

15. Quit your **`vi`** session saving the changes you made to the file **`tst`**.

    Answer:

    Enter **`:wq`** or **`ZZ`**

14-11. 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.

14-11. SLIDE: Moving Text: p, P — Instructor Notes

Key Points

- The unnamed cut buffer is immediately overwritten by the next delete or yank (discussed on next slide) operation.
- Any object that is deleted can be pasted back. Text can be easily moved by deleting, moving the cursor, and pasting.

Teaching Tips

The cut and paste examples are presented on two slides. The first slide is character and word oriented, and the second slide is line oriented.

There is actually one unnamed buffer and nine numbered buffers (1–9) used by the delete and yank functions by default.

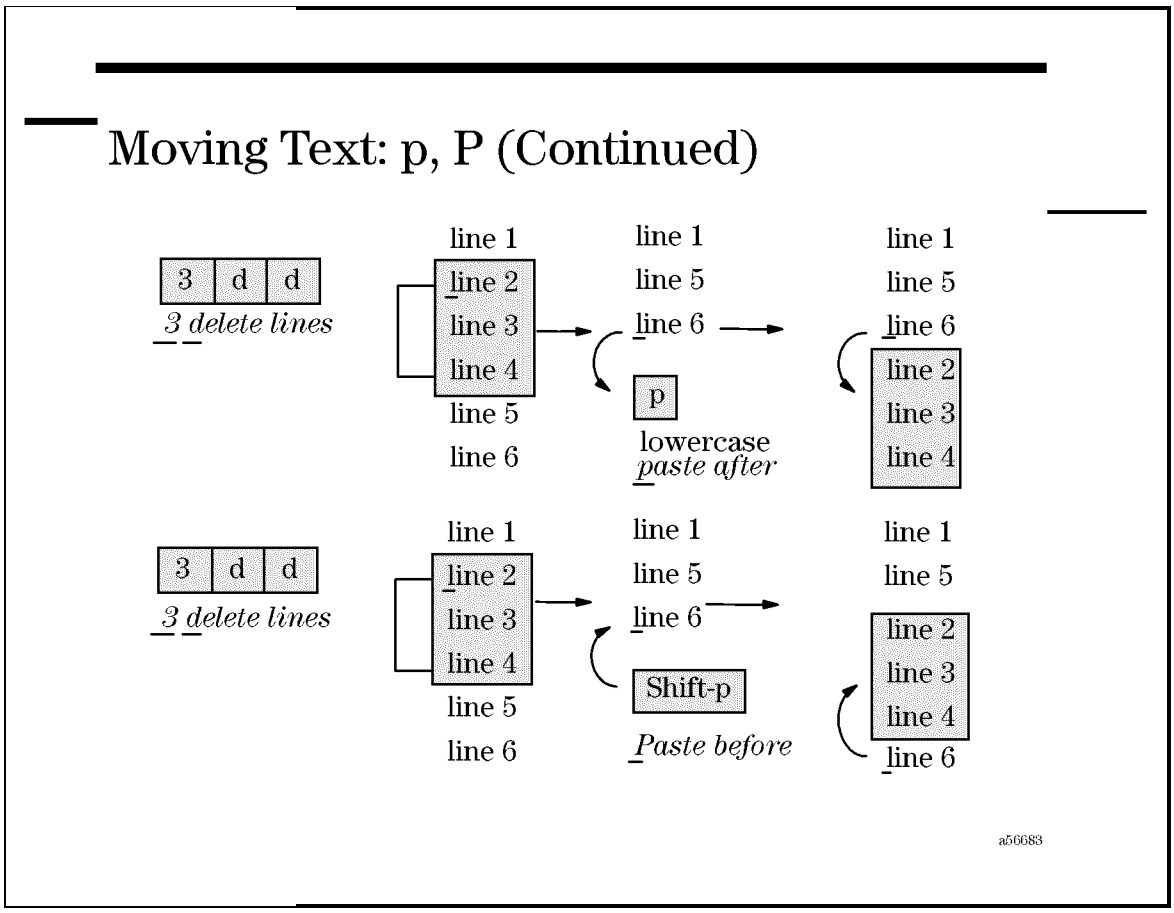
In addition, you can cut or copy text into specified named buffers, identified by a single character. These are not covered in the course materials, but you may have some advanced students who are interested.

Both are presented on the *vi Quick Reference Card* included with the students materials.

Examples

To delete 3 lines to a *named* buffer **b**: **b3dd**
To paste from the *named* buffer **b** : **bp**

14-11. SLIDE: Moving Text: p, P (Continued)



Student Notes

Delete and Paste Summary

d*object* Delete *object* into the cut buffer.

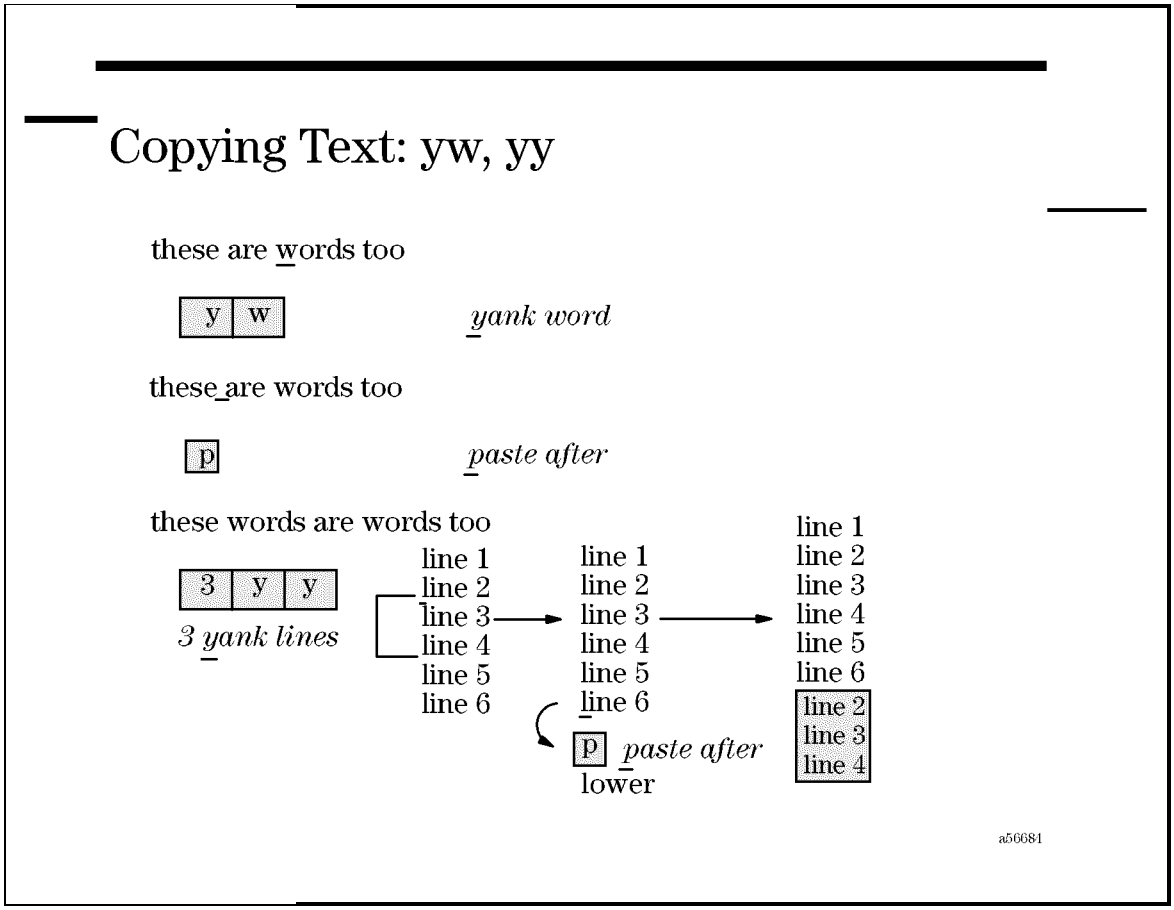
p (lowercase) **paste** contents of the cut buffer back into the text *after* the cursor.

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.

14-11. SLIDE: Moving Text: p, P (Continued) Instructor Notes

14-12. 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.

Copy and Paste Summary

`y`*object*	**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.

14-12. SLIDE: Copying Text: yw, yy Instructor Notes

Key Points

- The unnamed cut buffer is immediately overwritten by the next delete or yank operation.
- Any object that is yanked can be pasted back. Text can be easily copied by yanking, moving the cursor, and pasting.

Teaching Tips

You might want to mention that the command **Y** is equivalent to **yy**. But **yy** follows the convention of repeating the command and affecting the entire line.

There is actually one unnamed buffer and nine numbered buffers (1–9) used by the delete and yank functions by default.

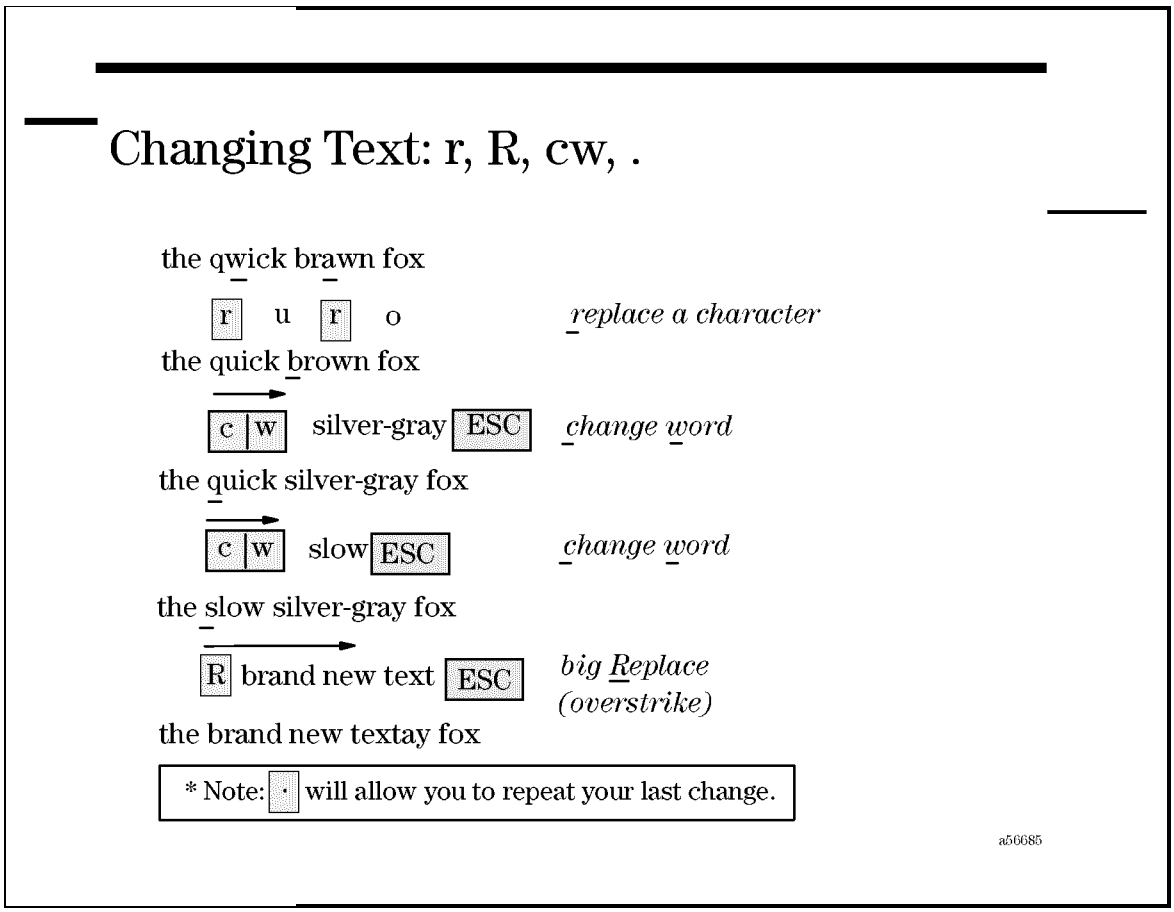
In addition, you can cut or copy text into specified named buffers, identified by a single character. These are not covered in the course materials, but you may have some advanced students who are interested.

Both are presented on the *vi Quick Reference Card* included with the students materials.

Examples

To yank 3 lines to a *named* buffer **b**: **3yy**
To paste from the *named* buffer **b** : **bp**

14-13. 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.

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 the entire thing in again, but look what the dot command will 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.

14-13. SLIDE: Changing Text: r, R, cw, . Instructor Notes

Key Points

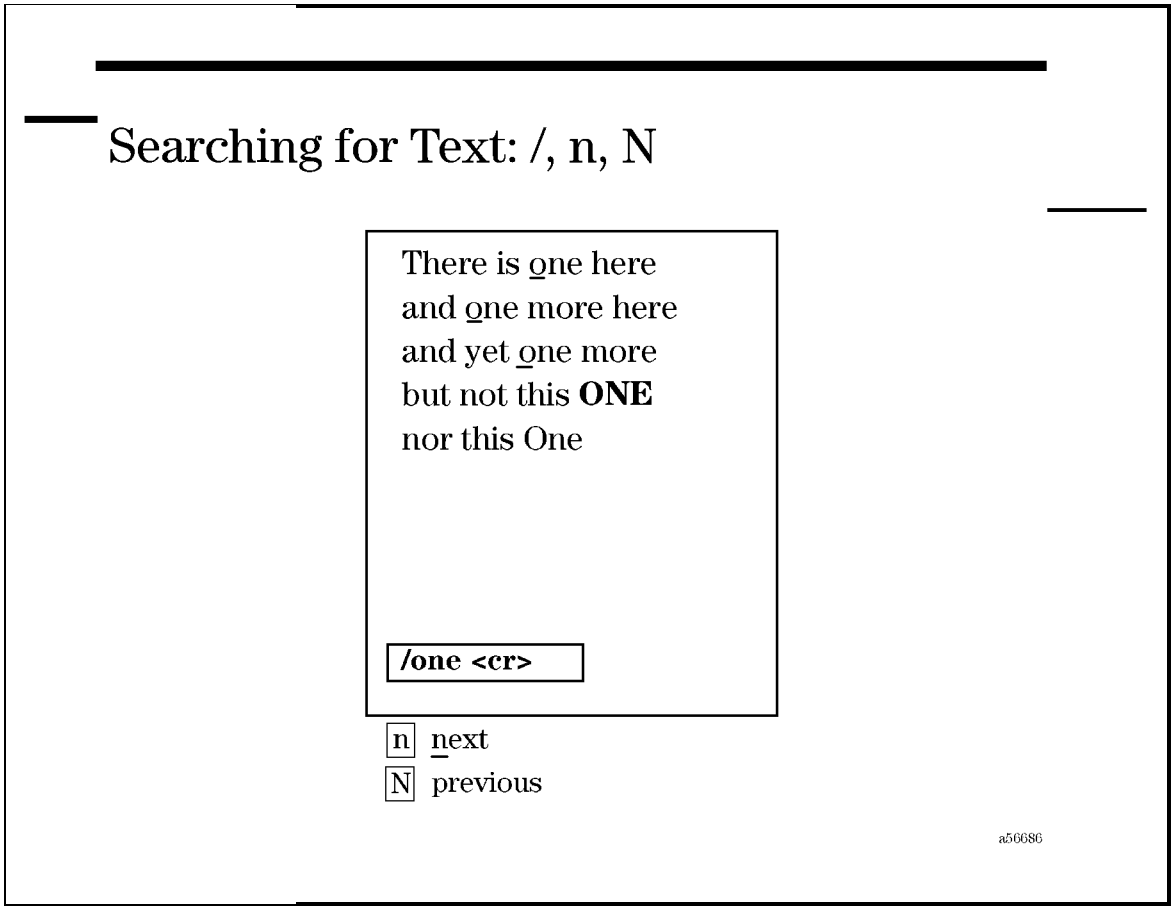
- One of the most common typing mistakes is incorrectly entering a *single character*. The **r** command is indispensable for fixing these errors.
- When you need to replace a single word, the **cw** is extremely effective.

Teaching Tips

Screen the slide, and use progressive display to individually display each line, the command, and then the resultant outcome.

Point out that the verb **c** is very much like the verb **d**.

14-14. 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.

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.

14-14. SLIDE: Searching for Text: /, n, N Instructor Notes

Key Points

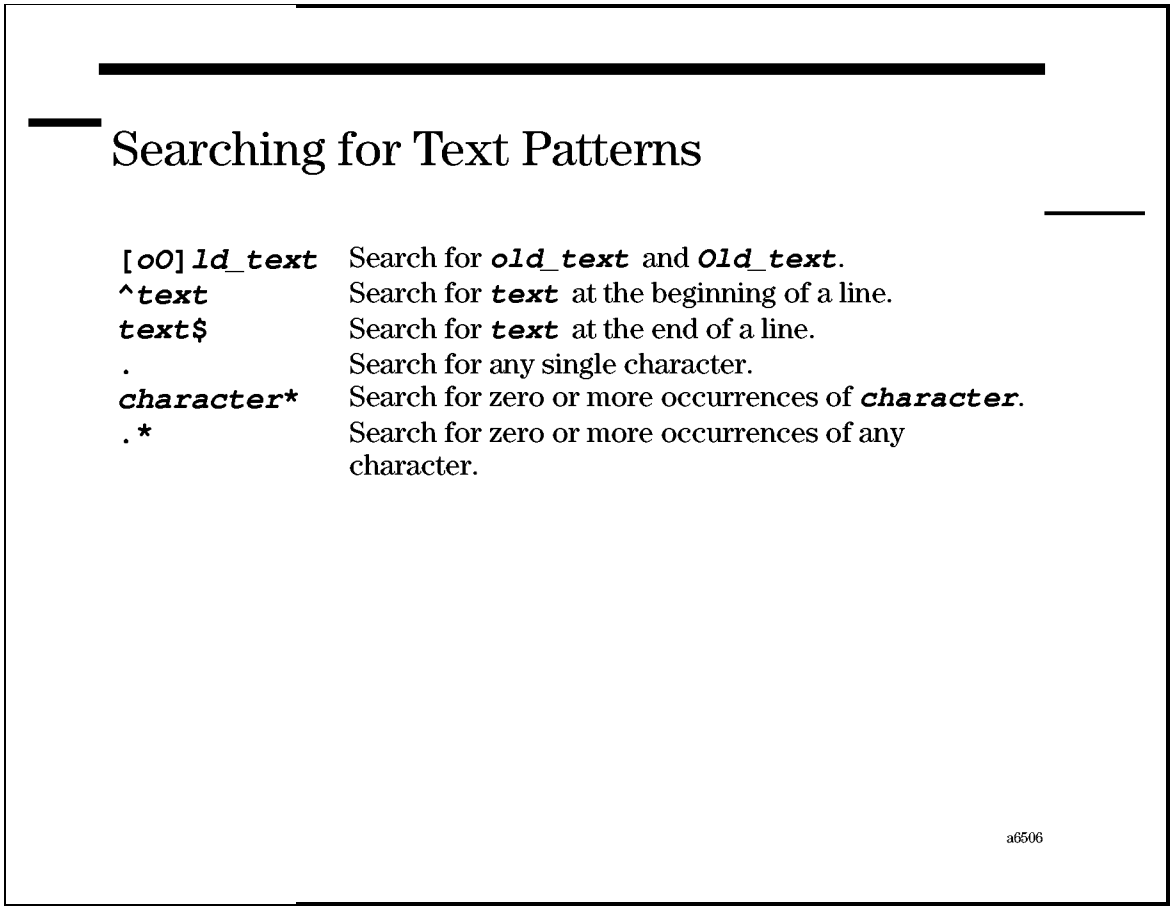
- Text searching is case sensitive
- Wrap around is supported

Teaching Tips

You can issue the **ex** command to ignore case:

```
:set ic
```

14-15. 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*.

Examples

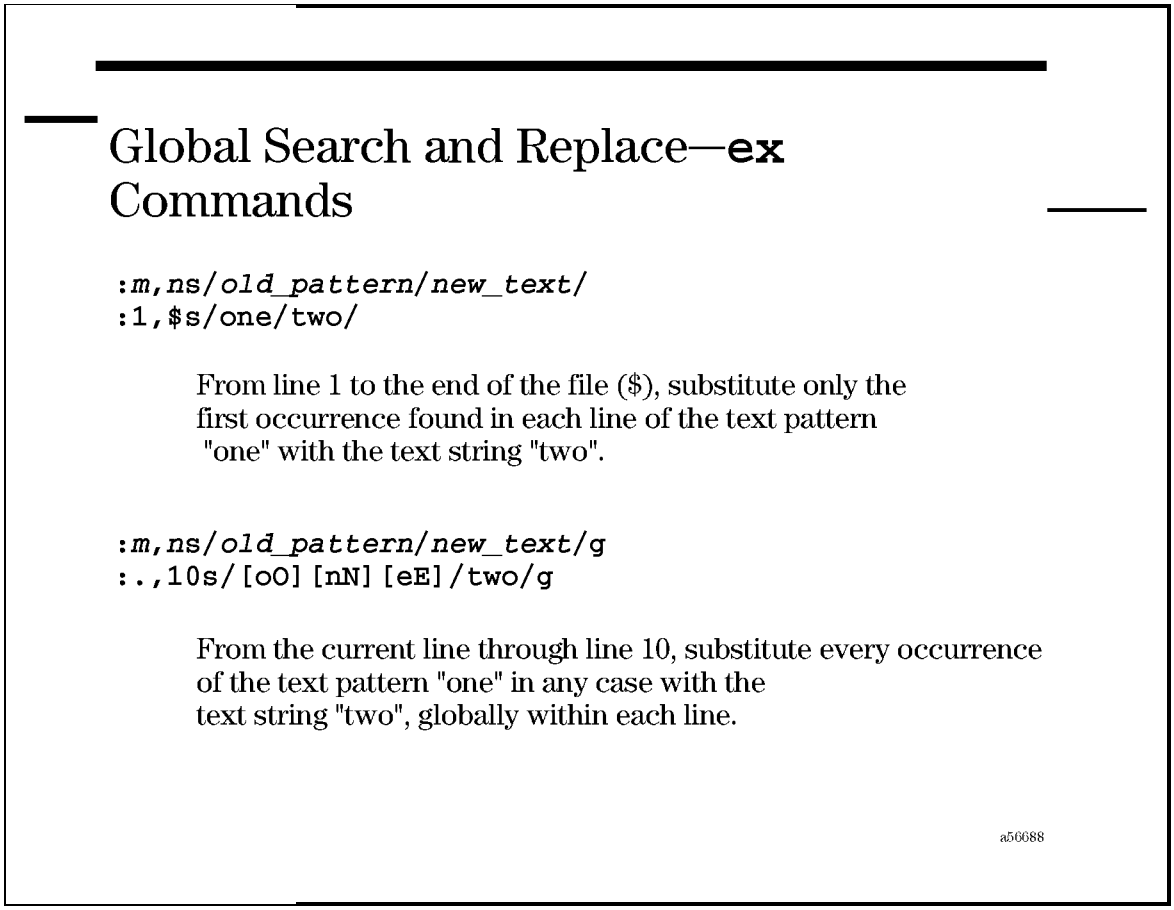
`/[Tt]he`	Searches for the next occurrence of the string *The* or *the* .
`/[oO][nN][eE]`	Searches for the string *one* with any character in any case.
`/bo*t`	Searches for the string *b* followed by zero or more *o*'s, followed by *t*. This would match *bt*, *bottom*, *boot*, *booot*, and so on.
`/^[abc].*`	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.* would match the pattern, as well as a line that ended with *finallyA* or *finallyZ*.

14-15. SLIDE: Searching for Text Patterns Instructor Notes

Key Points

- The most useful regular expression construct is the capability to search for a string by designating a character class `[]`, disabling the case sensitivity of a text search.
- There is usually confusion around the `*` and it's *zero or more* interpretation. You should go through the examples in the student notes.

14-16. SLIDE: Global Search and Replace — `ex` Commands



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,$`**).

```
:.,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.

14-16. SLIDE: Global Search and Replace — `ex` Commands Instructor Notes

Key Points

- Remember **`ex`** is a line oriented text editor. So the trailing **`g`** denotes global within the context of a line. The **`1,$`** provides global operation within the context of all of the lines of the file.

Teaching Tips

When presenting this command, verbalize it for your students to make it easier to comprehend. It really is a very cryptic command!

```
:1,$s/one/two/
```

On lines 1 through the end of the file (**`1,$`**), **substitute** the first occurrence found on each line of text string *one* with text string *two*.

```
:.,10s/[oO][nN][eE]/two/g
```

From the current line through line 10 (**`.,10`**), **substitute** the text pattern defined by **`[oO][nN][eE]`** with the text string *two*, globally *within each line*.

There is also a confirmation option that can be specified. The user responds with **`y`** or **`n`**. The default is **`n`** if you just enter a Return.

```
:.,10s/[oO][nN][eE]/two/gc
```

14-17. SLIDE: Some More `ex` Commands

Some More **ex** Commands

:w	write the current buffer to disk
:*m*,*n*w *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

a6284

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
```

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)
```

14-17. SLIDE: Some More `ex` Commands — Instructor Notes

Key Points

- Be sure to point out the following. They will be needed to complete the exercises:

  — **`:m,n w`** *file*
  — **`:e`** *file*
  — **`:se nu`**

Teaching Tips

You may point out any other options that you desire. Also, you may wish to discuss the abbreviations that are allowed.

One handy option when merging the output of commands with the contents of a file is to use the **`:r`** command in conjunction with **`!`***cmd*:

`:r !`*cmd*	read the output of the *cmd* into your file
`:r !date`	read the output of the **`date`** command into your file
`:r !ls`	read the output of the **`ls`** command into your file

You might also want to mention the capability to define keyboard macros. An example that is handy for users who do a lot of C programming would map the **{** in input mode to also provide the closing **}** and a blank line in between with an indentation. Similarly, you could define the opening **(** with the closing **)** and be left in input mode between the parentheses.

Macros

`:map` *keystroke vi_commands*	maps keystrokes while in *command* mode
`:map!` *keystroke vi_commands*	maps keystrokes while in *input* mode

Examples

Map an entered **{** to provide a closing **}** with an indented blank line between. When the user types in a **{**, **`vi`** will provide the rest.

The mapping will be displayed as follows:

```
:map { i{^M} ^[ko tab
```

You generate the mapping with the following keystrokes:

```
:map { i{ Ctrl + v Return } Space Ctrl + v ESC ko tab
```

Map an entered **(** to provide a closing **)** and put you in input mode between the parentheses.

The mapping will be displayed as follows:

```
:map ( i()^[i
```

You generate the mapping with the following keystrokes:

```
:map ( i() Ctrl + v ESC i
```

:map displays keystroke maps defined in *command* mode

:map! displays keystroke maps defined in *input* mode

14-18. TEXT PAGE: `vi` Commands — Summary

Table 14-1.

Cursor Control	Input Mode	Delete Text	Change Text	Write & 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		**Copy Text**	**Paste Text**	**Miscellaneous**
#G		yw	p (lowercase)	u
:#		yy	P (uppercase)	U
[Ctrl] + [g]		y$		.
[Return]				/ *text*
[Ctrl] + [b]				n
[Ctrl] + [f]				:!cmd
[Ctrl] + [u]				
[Ctrl] + [d]				
L				
M				
H				
[Ctrl] + [l]				

Student Notes

14-18. TEXT PAGE: vi Commands — Summary Instructor Notes

14-19. 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*.

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.

```
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.

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.

14-19. LAB: Modifying Text — Instructor Notes

Time: 30 minutes

Purpose

To practice the change text, file manipulation, and global search and replace features of **`vi`**.

Solutions

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*.

Answer:

```
:1,$s/bug/FEATURE/g
```

3. Write the first forty lines of the **`funfile`** out to another file called **`new.40`**.

Answer:

```
:1,40w new.40
```

4. Go to the last line in **`funfile`**.

Answer:

```
G
```

5. Find and execute the command to place your cursor midway down the window.

```
This file is silly.
```

Answer:

```
This file is silly.
```

ESC

6. Without quitting **`vi`**, write your new version of the file out to a file called **`funfile.123`**.

Answer:

```
:w funfile.123
```

7. Without leaving **`vi`**, load the file **`new.40`** into the buffer, overwriting the previous contents.

Answer:

```
:e new.40
```

8. Turn on line numbering with the **`ex`** *number* option.

Answer:

```
:set number
```

9. Search for an occurrence of **`FEATURE`** in **`new.40`**.

Answer:

```
/FEATURE
```

10. Change all occurrences of *FEATURE* to *BUG* and save it into **`new.new.40`**.

Answer:

```
cwBUG ESC
```

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*.

Answer:

```
1G
/[Ss]ystem
cwXXXXX ESC
n
.
n
n
.
n
.
n
.
```

12. Write your current edit session and quit the editor.

Answer:

```
:wq
```

or

```
ZZ
```

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 which 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.

Overview of Module 15

Audience

general user — General system user

Product Family Type

open sys — Open systems environment

Abstract

This module is designed to teach about submitting background jobs, and the **`ps`**, **`nohup`**, and **`kill`** commands.

Time

Lab — 30 minutes

Lecture — 30 minutes

Prerequisites

m1306m — Input and Output Redirection

Instructor Profile

UX — General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000 — Logon access to an HP-UX system

Software Requirements

UX11 — HP-UX release 11.0

Material List

P/N B2355-90033(T)	*HP-UX Reference Manual* , one per terminal

Lab Instructions

setup1	Create user logon of **user1**, **user2**, ... **user***n*, where *n* is the number of students in the class. Set up one user per student.
compile	Compile **infinite.c** and copy the object file **infinite** to the users' home directories.

Lab Files

```
-rw-r--r-- 1   karenk  users    868   May 28 16:12 infinite.c
```

15-1. SLIDE: The ps Command

The ps Command

Syntax:

ps [-efl] Reports process status

Example:

```
$ ps
  PID TTY      TIME COMMAND
 1324 ttyp2    0:00 sh
 1387 ttyp2    0:00 ps

$ ps -ef
  UID    PID  PPID  C    STIME   TTY     TIME COMMAND
 root     0    0    0    Jan  1  ?       0:20 swapper
 root     1    0    0    Jun 23  ?       0:00 init
 root     2    0    0    Jun 23  ?       0:16 vhand
 root     3    0    0    Jun 23  ?      12:14 statdaemon
user3  1324    1    3   18:03:21 ttyp2  0:00 -sh
user3  1390 1324   22   18:30:23 ttyp2  0:00 ps -ef
```

a65016

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 **e**very process running on the system, not just your own. The **-f** and **-l** options report **f**ull and **l**ong 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 a **ps** giving a full (**-f** option) listing of *every*
(**-e** option) process on the system.

NOTE: Be aware that the **ps** command is CPU intensive, and you may notice a slower response while it is executing.

15-1. SLIDE: The ps Command — Instructor Notes

Key Points

- **ps** output
- PID
- PPID
- Show the parentage of the **sh** and **ps** commands with the output of the **ps** command on the slide.

NOTE: If the fair share scheduler has been installed on your system, the following options are also available for the **ps** command:

-F	Print the fair share group process association.
-G fglist	Restrict listing to data about processes whose fair share group ID numbers or fair share group names are given in **fglist**.

Additionally, the following column is added to the standard output:

FSID (f,l)	The fair share group ID of the process; the fair share group ID under the **-l** option and the fair share group name under the **-f** option. If neither the **-l** option nor the **-f** option are specified when the **-F** option is specified, the fair share group name is printed.

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
```

a566161

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* 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 backgrounded 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.

15-2. SLIDE: Background Processing — Instructor Notes

Key Points

- Execute commands that take a long time to complete in the background, so that you can continue other activities.
- If the **monitor** option is **set**, the job number and process ID will be displayed when a background job is submitted. A message will also be displayed when the background command is complete.

```
[1]+ Done grep user * > grep.out &
```

- You should redirect output of background commands.
- You must redirect input of background commands.
- You cannot interrupt background commands through the keyboard, you must use the **kill** command.
- If you are using CDE, closing a window will kill all the processes running in this window. Exiting CDE will kill all processes running in all windows (background or foreground processes). You can protect process execution by using **nohup** prefix command.

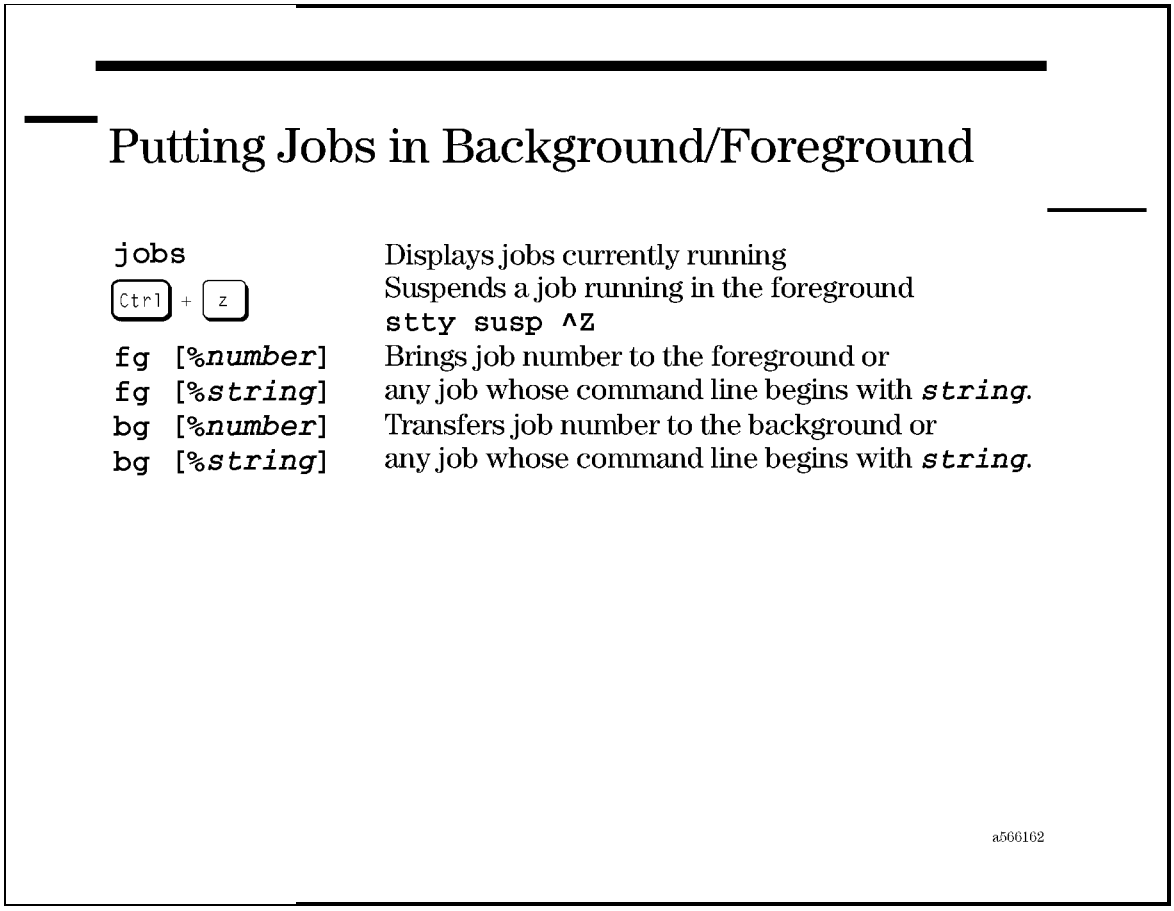
Teaching Tips

You might want the students to experiment with background commands using the **sleep** command, since most commands complete too quickly to observe in the background:

```
$ sleep 120 &
[1]     1389
$ ps -f
    UID           FSID PID  PPID  C STIME    TTY   TIME COMMAND
  user3 default_system 1324    1  2 18:03:21 ttyp2 0:00 -sh
  user3 default_system 1390 1324 15 18:30:23 ttyp2 0:00 ps -ef
  user3 default_system 1389 1324  3 18:30:23 ttyp2 0:00 sleep 120
```

Also, you might want to mention that the **set -o** command displays the current status of all of the **set** options.

15-3. SLIDE: Putting Jobs in Background/Foreground



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.

15-3. SLIDE: Putting Jobs in Background/ Foreground — Instructor Notes

Key Points

- **`jobs`** allows you to see the jobs running under your current session.
- You can get control of your terminal while it is running a command by issuing the *susp* character, normally Ctrl + **`z`**.
- You must issue **`stty susp ^Z`** to map the suspend character. This is normally done in the **`.profile`** file.
- You can use the **`bg`** command to send a suspended job to the background.
- You can use the **`fg`** command to bring a backgrounded job to the foreground.

15-4. SLIDE: The nohup Command

The nohup Command

Syntax:

nohup ***command line &*** Makes a command immune to hangup (logout)– no hangup

Example:

```
$ nohup cat * > bigfile &
   [1] 972
$Ctrl + d
login: user3
Password:
  .
  .
  .
$ ps -ef | grep cat
UID    PID   PPID          COMMAND
user3  972     1    ....   cat * > bigfile &
```

a68912

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**.

15-4. SLIDE: The nohup Command — Instructor Notes

Key Points

- **nohup** protects backgrounded commands from being terminated when you log out.
- The output of the command must be redirected. If the user does not designate an output file, **nohup** will redirect to the file called **nohup.out**.
- When the **nohup** command's parent shell is terminated, it will be adopted by process 1 (**init**).
- The **nohup** command is useful for users who want to start the execution of a command in the background and be able to log out before the command has completed running.

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 &
$
```

a6288

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.

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. Negative increments can only be used by the **root** user.

15-5. SLIDE: The `nice` Command — Instructor Notes

Key Points

- The **`nice`** command is a prefix command.
- The lower the priority number displayed from the **`ps`** command the higher the priority (more important) of the process.
- A command with a higher **`nice`** value has a lower priority (less important).
- The **`nice`** command *does not* literally add that amount to the process priority. It is only one of several factors that determine the actual process priority.
- The super-user can make a process *mean* or *not nice* by designating a negative increment to the nice command: **`nice --10 cmd`**
- **`nice`** and **`nohup`** can be combined in a single command line.

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
```

a566165

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 kill is issued by the super-user.

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.

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 **`-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
```

15-6. SLIDE: The `kill` Command — Instructor Notes

Key Points

- The **`kill`** command is used to terminate processes.
- The **`kill`** command is the only way to terminate a background process.
- One user cannot kill another user's processes.
- **`kill`** does not actually kill the process; it sends a signal to which the default action is usually for the process to die. The **`KILL`** signal (signal 9) cannot be ignored by the process.

Teaching Tips

The EXIT signal (signal number 0) does not actually send a signal to a process, but it can be used to check for a valid process ID number:

```
$ sleep 2000 &
[1]    13288
$ kill -s EXIT 13288
$ ps
  PID TTY   TIME COMMAND
13288 ttypa 0:00 sleep
13289 ttypa 0:00 ps
13264 ttypa 0:01 sh
$ kill -s EXIT 99999
kill: 99999: no such process
```

The signal names can be specified in upper or lower case, as in

```
kill -s INT 1234
kill -s int 1234
```

Also, the signal name may be specified with the *sig* prefix, as in

```
kill -s SIGINT 1234
kill -s sigint 1234
```

The **`kill`** command can be invoked without the **`-s`** option by preceding the signal name or signal number with a minus sign (note that both of these forms are considered obsolete):

```
kill -INT 1234
kill -int 1234
kill -9 1234
```

Note that the shell variable **`$!`** stores the process ID number of the last backgrounded command. Therefore, you could issue the command: **`$ kill $!`**

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
hello
hello
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. Re-run 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?

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.

15-7. LAB: Process Control — Instructor Notes

Time: 30 minutes

Purpose

To practice executing background commands and using the **ps**, **nohup**, and **kill** commands.

Notes to the Instructor

Exercise number 1 assumes that *interrupt* is mapped to Ctrl + **c**.

Exercise number 5 assumes that *suspend* is mapped to Ctrl + **z**.

Students can issue **stty -a** to confirm these settings.

Solutions

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
hello
hello
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.

Answer:

The PID (process ID number) of the shell (-sh) will be the PPID (parent process ID number) of the **infinite** command. When you log out, terminating the parent process, all child processes (including **infinite**) are killed.

3. The **nohup** command protects a process from terminating upon the death of its parent process. Re-run 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?

Answer:

When the parent process (your shell) dies, the child process (**infinite**) becomes an **orphan** process. Orphan processes are **adopted** by PID 1 (**init**). When you log back in, you will see **infinite** still running.

4. Use the **kill** command to terminate your **infinite** program.

Answer:

```
$ kill PID                          PID is returned from the
                                    ps command
```

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.

Answer:

```
$ infinite > infin.out
Ctrl + z
[1] + Stopped infinite > infin.out
$ bg %1
[1] infinite > infin.out &
$ kill %1
[1] + Terminated infinite > infin.out
```

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.

Overview of Module 16

Audience

general user General system user

Product Family Type

open sys Open systems environment

Abstract

This module will introduce shell programming, passing data to shell programs through the environment and command line arguments, and requesting input from the user.

Time

Lab 75 minutes

Lecture 60 minutes

Prerequisites

m51m Shell Advanced Features

Instructor Profile

UX General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000 Logon access to an HP-UX system

Software Requirements

UX11 HP-UX release 11.0

Material List

P/N B2355-90033(T) — *HP-UX Reference Manual* , one per terminal

Lab Instructions

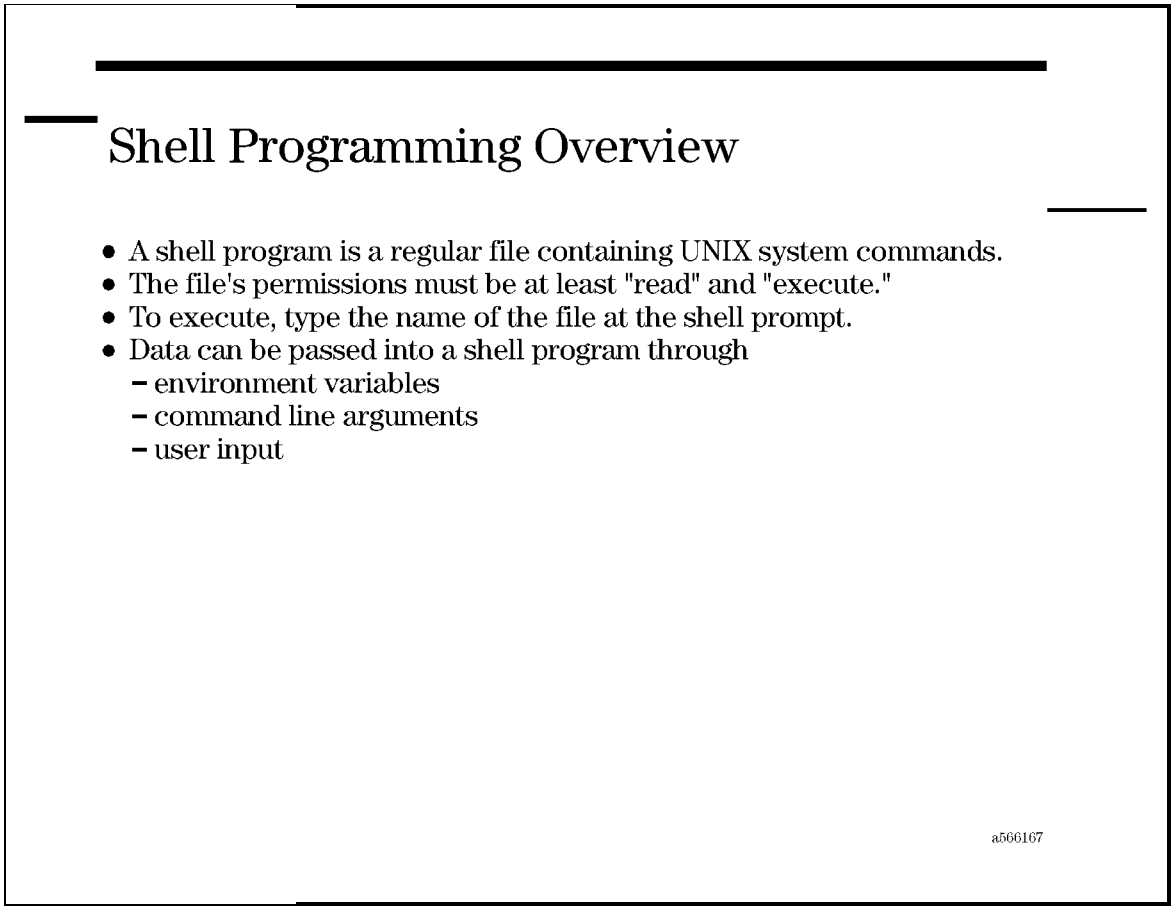
setup1 — Create user logon of **user1**, **user2**, ... **user***n*, where *n* is the number of students in the class. Set up one user per student.

copyfiles — Copy the lab files to the users' home directories.

Lab Files

```
-rwxr-xr-x   1 karenk  users    89 May 28 16:12 color1
-rwxr-xr-x   1 karenk  users   227 May 28 16:12 color2
-rwxr-xr-x   1 karenk  users   245 May 28 16:12 color3
-rwxr-xr-x   1 karenk  users   101 May 28 16:12 color4
-rwxr-xr-x   1 karenk  users   293 May 28 16:12 color5
-rwxr-xr-x   1 karenk  users   164 May 28 16:12 color6
-rwxr-xr-x   1 karenk  users    40 May 28 16:12 myprog
-rw-r--r--   1 karenk  users    86 May 28 16:12 myprog.c
```

16-1. SLIDE: Shell Programming Overview



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.

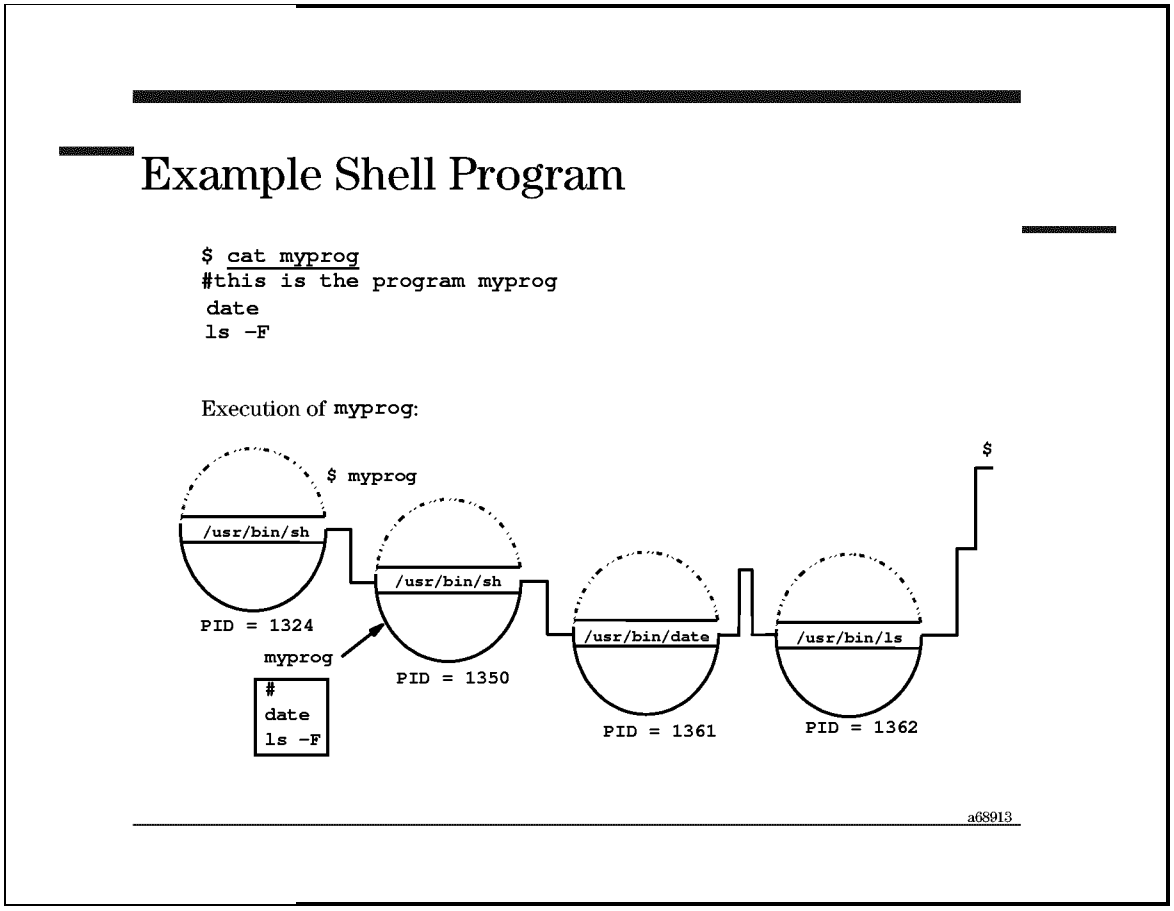
Rather complex shell scripts can be developed because the shell supports variables, command line arguments, interactive input, tests, branches, and loops.

16-1. SLIDE: Shell Programming Overview Instructor Notes

Key Points

- The focus of this module will be command line arguments and user input.
- Users will be developing shell programs in this module.

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.

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.

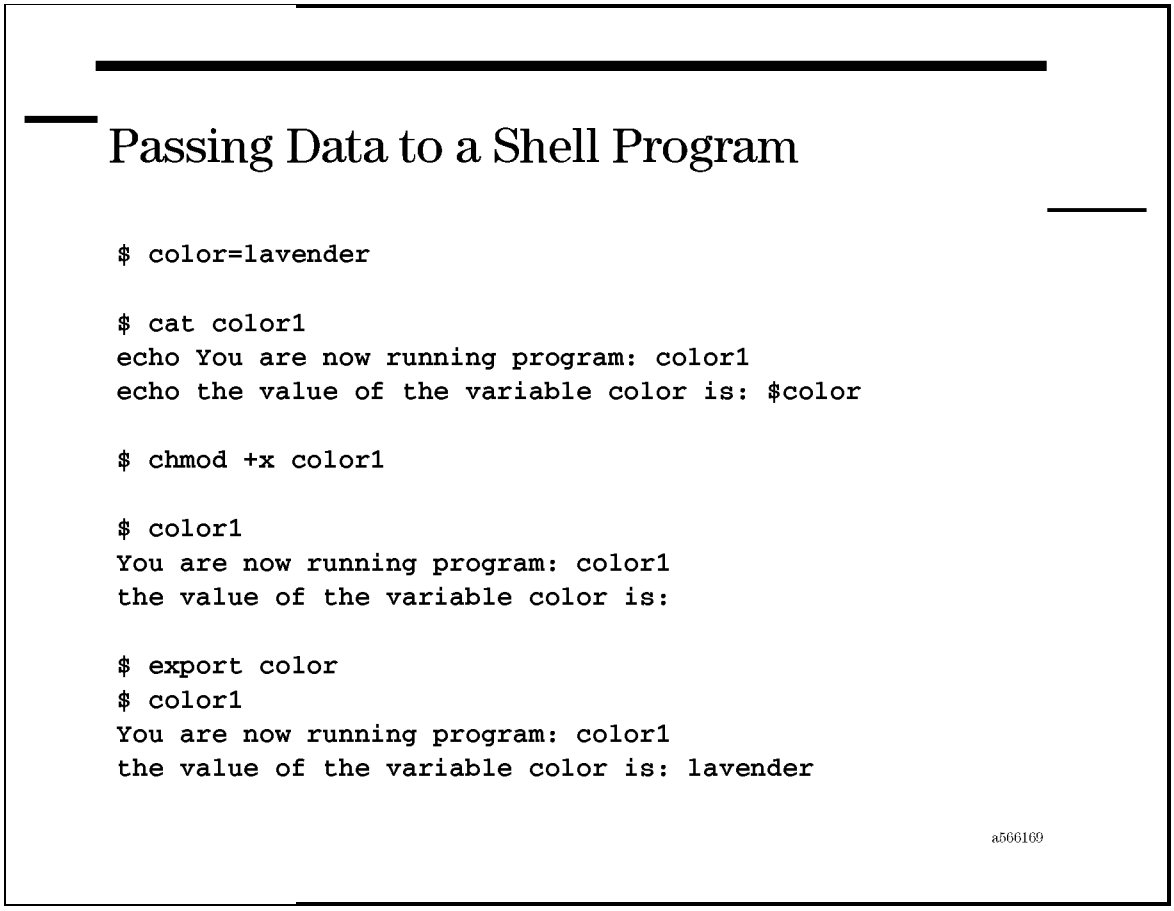
NOTE: You should never call a shell program **`test`** because test is a built-in shell command.

16-2. SLIDE: Example Shell Program — Instructor Notes

Key Points

- A shell program or shell script just holds a collection of shell commands.
- The permissions must be at least read and execute. (This provides a good opportunity to review the **chmod** command.)
- Point out the importance of properly commenting your shell scripts.

16-3. SLIDE: Passing Data to a Shell Program



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:

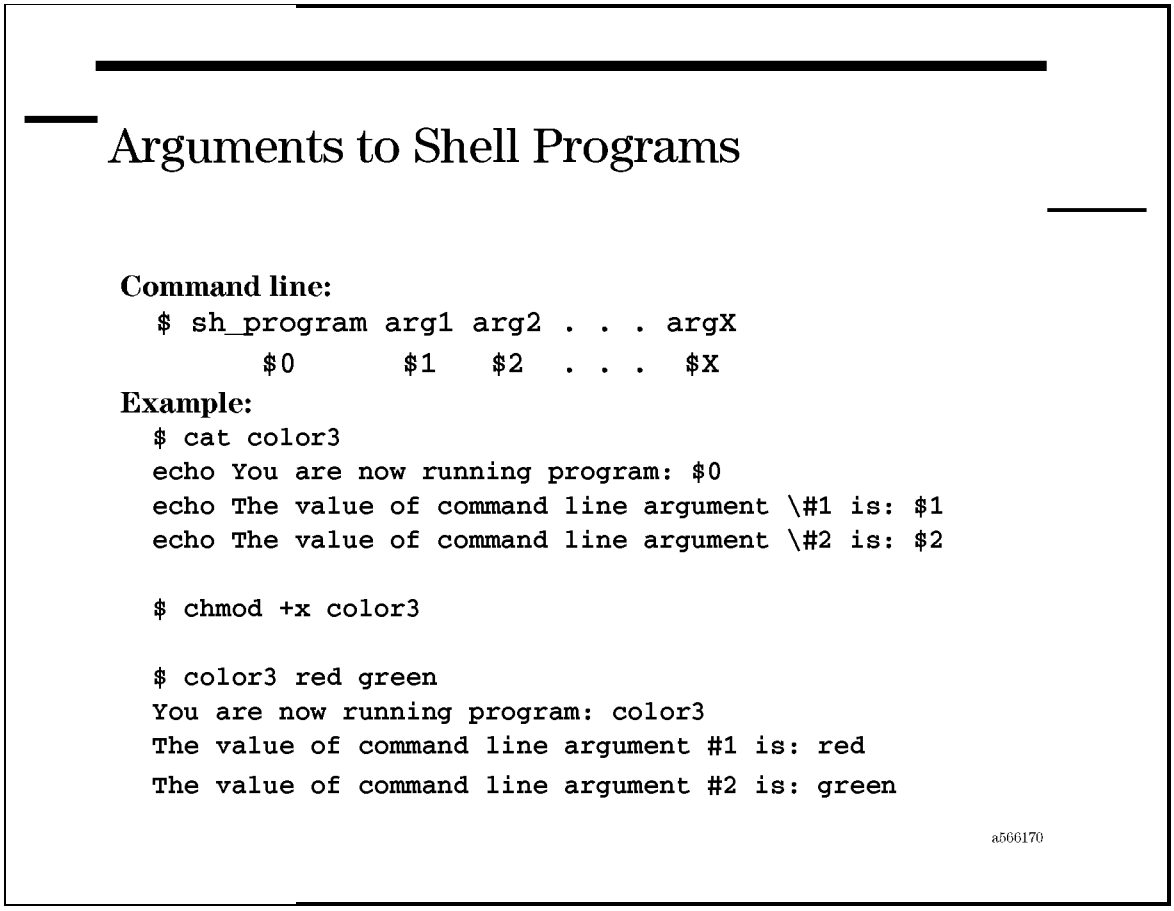
```
echo The original value of the variable color is $color
echo This program will set the value of color to amber
color=amber
echo The value of color is now $color
echo When your program concludes, display the value of the color variable.
```

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
```

16-3. SLIDE: Passing Data to a Shell Program Instructor Notes

16-4. SLIDE: Arguments to Shell Programs



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 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.

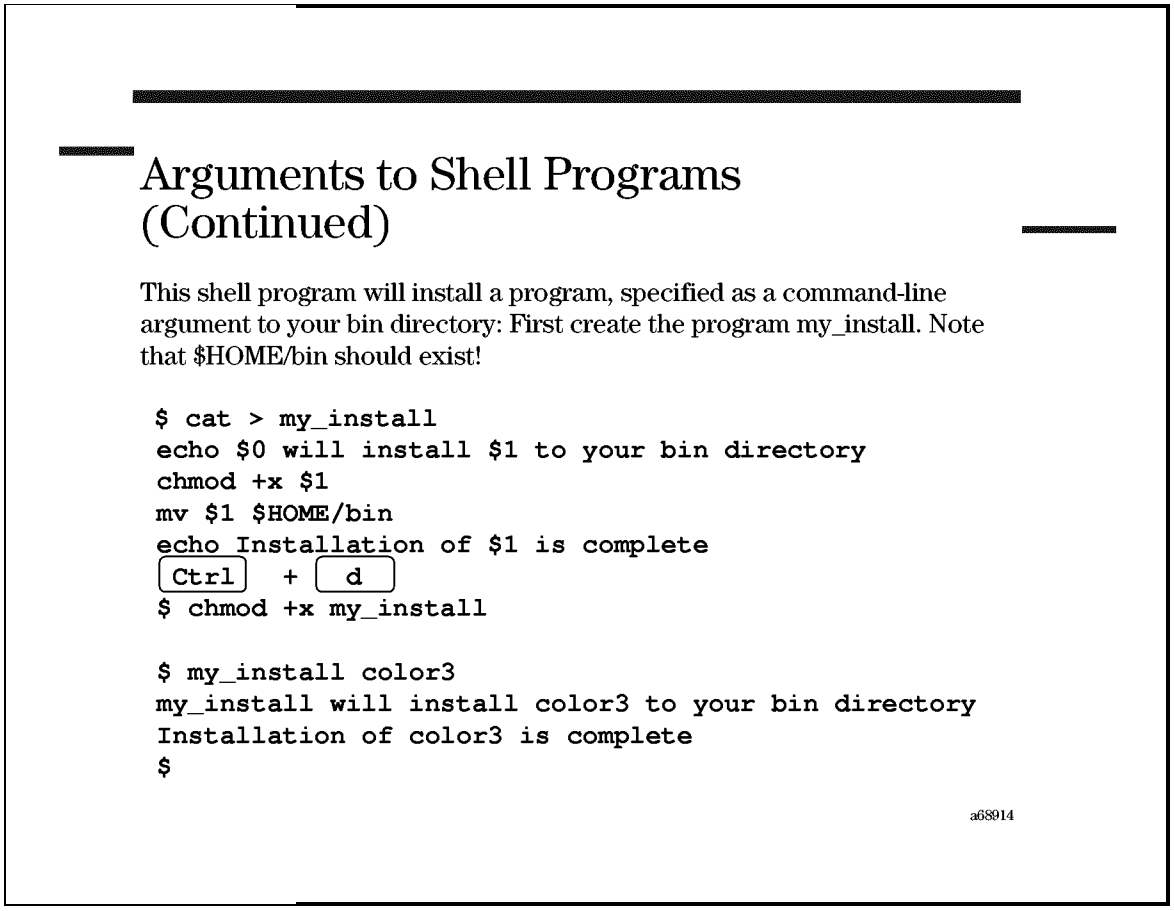
The only disadvantage to developing a program that accepts command-line arguments is that the users must know the proper syntax and what the command-line arguments represent. For example, how do you know that the **cp** command can copy one file to another file or several files to a directory? What happens 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.

16-4. SLIDE: Arguments to Shell Programs Instructor Notes

Key Points

- Command-line arguments allow you to pass information into your shell program.
- Unlike the Bourne shell, the POSIX and Korn shells support positional parameters for *all* of the command-line arguments, not just the first ten. The curly brace notation must be used after `$9` , however.
- `$0` always references the command name.
- Their relevance to the shell program is entirely dependent on the contents of the shell program.
- You must provide syntax or usage messages to users so that they know how to invoke the program properly. Use **`cp`**, **`banner`**, and **`ls`** as examples.

16-4. SLIDE: Arguments to Shell Programs (Continued)



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.

16-4. SLIDE: Arguments to Shell Programs (Continued) Instructor Notes

Teaching Tips

The objective of this slide is to show a more realistic example of a shell script. It will be embellished as we progress through this module.

16-5. 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
$
```

a566172

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.

16-5. SLIDE: Some Special Shell Variables — # and * Instructor Notes

Key Points

- These special variables are useful when supporting a variable number of command-line arguments.
- Each command-line argument still maintains its individual identity.

Teaching Tips

Tips for remembering the special variables:

- **#** usually represents a number symbol, as in **#1** (number 1).
- ***** in the UNIX system usually means *everything* . In **echo *** the ***** will generate *all* file names, in **echo $***, the ***** represents all command-line arguments.

  There are other special shell variables that you may wish to mention:

  $ The process ID of the current process

  ! The process ID of the last background process

  - The options specified for the current shell

16-5. 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
```

a68915

Student Notes

The installation program is now more flexible. If you have several scripts that need to be installed, you only have to execute the program 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 to more than one directory, the program will incur an error.

16-5. SLIDE: Some Special Shell Variables — # and * (Continued) Instructor Notes

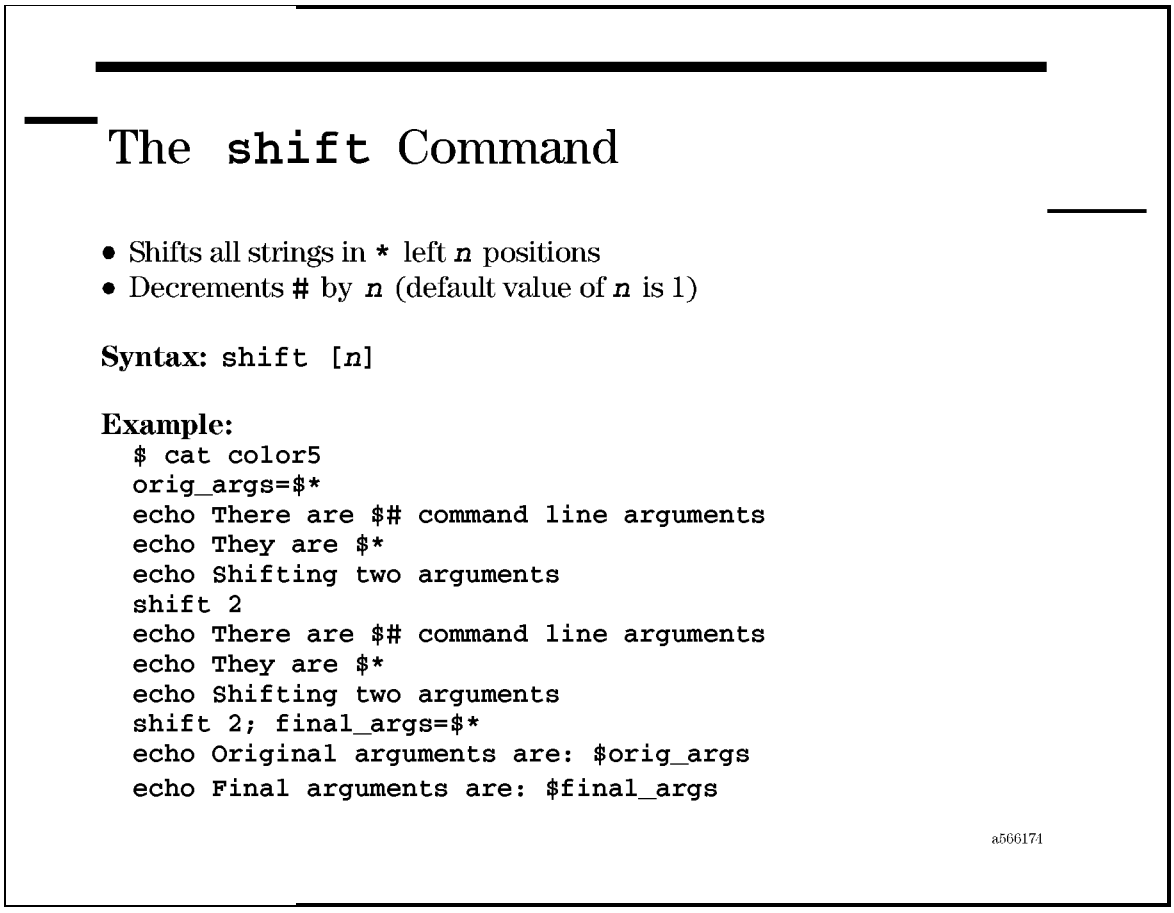
Key Points

- Some commands do not accept a variable number of arguments.
- The *Shell Programming — Branching* module shows how to verify that the correct number of command-line arguments has been entered.

Teaching Question

Ask the students what mechanism they would use to verify the number of arguments. They should at least be aware that they can compare **$#** with the number of required arguments, even though they do not yet know the syntax.

16-6. SLIDE: The `shift` Command



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

Example

The following shows the output that would be generated if the shell program illustrated in the slide were executed:

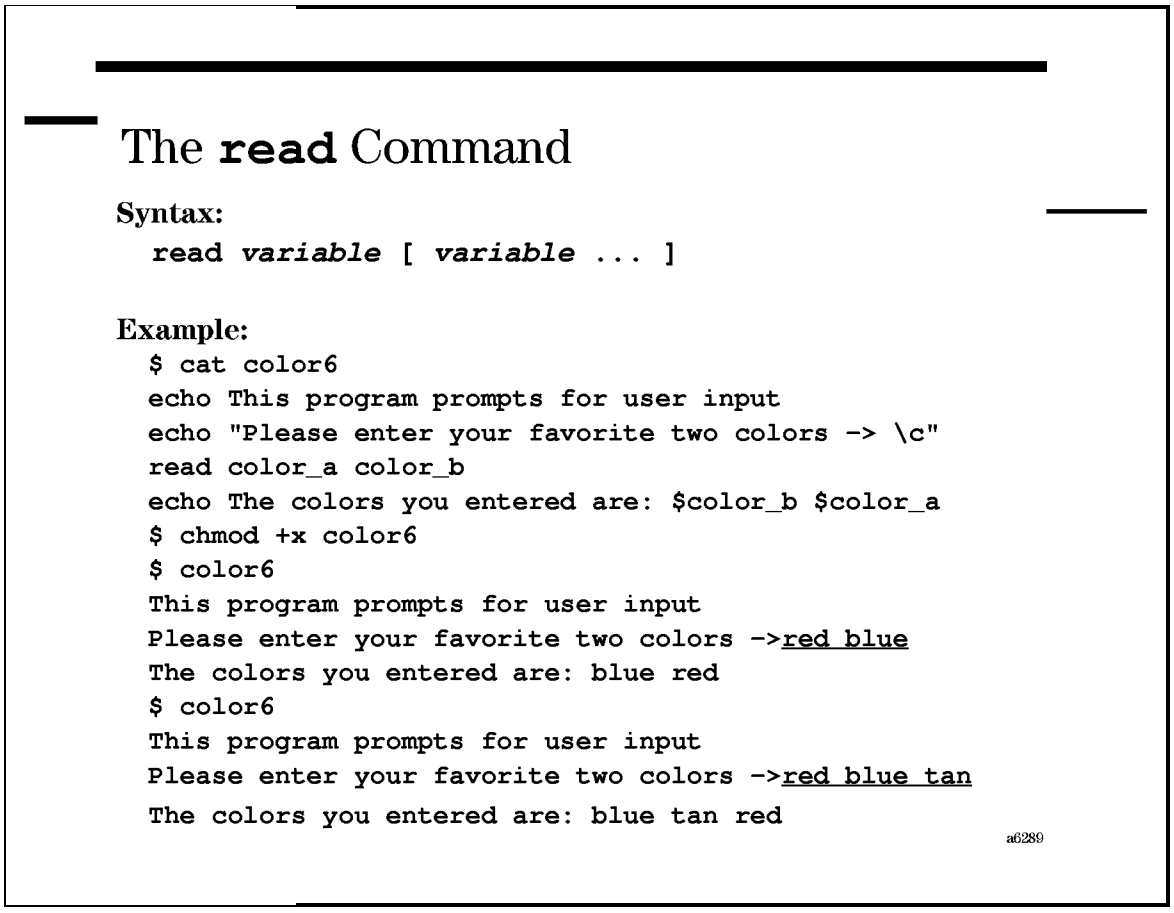
```
$ color5 red green yellow blue orange black

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
$
```

16-6. SLIDE: The `shift` Command — Instructor Notes

Key Points

- Shifted arguments are lost. They must be saved prior to the shift if you will need them later in your program.
- Review the program code on the slide and the corresponding output included in the student notes.

16-7. SLIDE: The `read` Command



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, each **read** statement should be preceded by an **echo** 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.

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 a zero.

16-7. SLIDE: The `read` Command — Instructor Notes

Key Points

- **`read`** allows the program to prompt the user for input. It makes the program more interactive.
- Make sure to distinguish **`read`** input from command-line arguments.
- The variables assigned by the **`read`** command are separate from the variables assigned by the positional parameters.
- Leftover input is assigned to the last variable specified on the **`read`** command line.

Teaching Tips

- You might want to mention the use of the **`\c`** in the prompt **`echo`** which suppresses the line feed so that the input is provided on the same line as the prompt.
- Go over the examples slowly. Make sure the class understands the difference between command-line arguments and read-in variables. Many students will try to use the **`read`** command and then access the read-in variables using **`$1, $2, $3`**, and so on.

Teaching Question

How would you separate the words in the last variable?

Answer: Use the **`cut`** command on the variable value, using a blank space as a delimiter between fields.

Remember **`cut`** will only operate on standard input or the contents of a file; it cannot directly access the value of a variable. How do you cut the value of a variable?

Answer: Send the value of the variable to standard output using the **`echo`** command. For example,

```
echo $varname | cut -f1 -d " "          Displays value to stdout

param1=$(echo $varname | cut -f1 -d " ")
                                        Assigns value to variable param1
```

16-7. 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
```

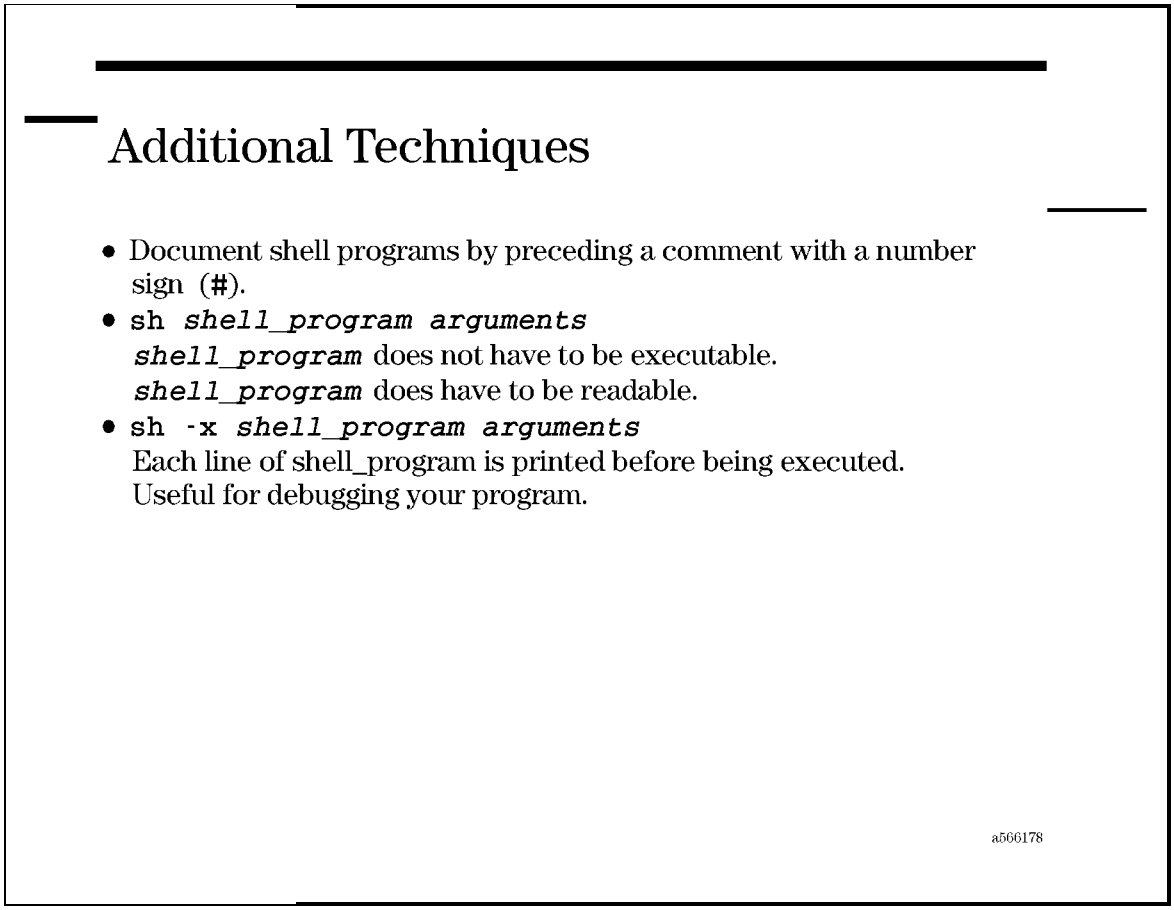
a68916

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*.

16-7. SLIDE: The `read` Command (Continued) Instructor Notes

16-8. SLIDE: Additional Techniques



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 **#!/usr/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: **`#!/usr/bin/csh`**.

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.

16-8. SLIDE: Additional Techniques — Instructor Notes

Key Points

- This is a useful debugging tool, especially to view lines of code in which variables are not properly dereferenced.
- Mention how to specify a command interpreter.
- **`sh -x`** can also be invoked with a shell program via **`set -x`** and stopped with **`set +x.`**

16-9. 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 **banner** 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 into the system

2. Change to the **/tmp** directory. Invoke the program **color1**. Does the shell find the **color1** program?

3. Change to $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.

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 **banner** command, such as *welcome username*, so that if user3 was logged in it would banner *welcome user3* or if user2 was logged in it would banner *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
```

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.

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 which 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**.

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.

16-9. LAB: Introduction to Shell Programming — Instructor Notes

Time: 75 minutes

Lab Objective

To practice developing shell programs that accept command-line arguments and prompt the user for input.

Notes to the Instructor

Introductory exercises 1–10

Advanced exercises 11–18

Students who need to practice with basic command-line arguments and the **`read`** command should start with the introductory exercises.

Other students should start with introductory exercise number 5. Exercise number 6 requests that students write a script that will create a backup file when they start a **`vi`** editing session, and many students find it useful, since **`vi`** does not save any old version of the file that is being edited.

These exercises are intended to get students thinking about the types of utilities that *they* can develop.

Solutions

1. Create a shell program called **`whoson1`** that will

- display a greeting to the user with the **`banner`** 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 into the system

Answer:

```
$ vi whoson1
banner Welcome to whoson1
MYNAME=your_name
echo $MYNAME
date
who
```

```
$ chmod +x whoson1
$ whoson1
```

2. Change to the **/tmp** directory. Invoke the program **color1**. Does the shell find the **color1** program?

Answer:

```
$ cd /tmp
$ color1
sh: color1: not found
```

The **color1** program is not found because it does not reside under one of the directories specified by the *PATH* variable.

3. Change to $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.

Answer:

```
$ cd
$ mkdir bin
$ mv color1 bin
$ PATH=$PATH:$HOME/bin
$ cd /tmp
$ color=lavender
$ export color
$ color1
You are now running program: color1
the value of the variable color is: lavender
$
```

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.

Answer:

```
$ date_var=$(date)
$ export date_var
$ cd $HOME/bin
$ vi date_tst
echo the value of date_var is $date_var
$ chmod +x date_tst
$ date_tst
```

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**?

Answer:

```
$ vi $HOME/bin/date_tst
date_var=$(date)
echo the value of date_var is $date_var
$ date_tst
```

In this case *date_var* does not need to be exported because it is being defined and used within the same process level. The permissions on **date_tst** do not need to be changed because the program was made executable in the previous exercise. Editing a file does not affect its permissions.

6. Create a shell program called **whoson2** that will

- Display a personalized greeting to the user with the **banner** command, such as *welcome username*, so that if user3 was logged in it would banner *welcome user3* or if user2 was logged in it would banner *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.

Answer:

```
$ vi $HOME/bin/whoson2
banner welcome $LOGNAME     or     banner welcome $(whoami)

date
who
echo your terminal session identification information is
who am i
echo thank you for using whoson2
$ chmod +x $HOME/bin/whoson2
$ whoson2
```

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
```

Answer:

```
6
-d
A blank line.
abc def -d -4 +900 xyz
myshellprog
```

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
```

Answer:

```
4
+900
A blank line.
-d -4 +900 xyz
myshellprog
```

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"
```

Answer:

```
Please type in your first, middle, and last names
James A. Smith, Jr.
Smith, Jr., James A.
```

Note that "Smith, Jr." is read into the last variable.

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.

Answer:

This is shell program **search1**:

```
echo "Please enter a string to search for: \c"
read string
echo The following files contain the string $string:
grep -l $string *
```

11. Write a shell program called **backwards** that will receive up to ten arguments and list the arguments in reverse order.

Answer:

```
#!/usr/bin/sh
# backwards: reverses command line arguments
# usage: reverse a b c d e f g h i
#
echo ${10} $9 $8 $7 $6 $5 $4 $3 $2 $1
```

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.
```

Answer:

```
#!/usr/bin/sh
# myecho: Display the number of command line arguments,
#         print the first three arguments on separate lines
#         and print the remaining arguments on one line
#         usage: myecho a b c
#
echo "The number of command line arguments is $#."
#
echo $1;echo $2;echo $3
shift 3
echo $*
$ myecho a b c d e f g h i j k l
The number of command line arguments is 12
a
b
c
d e f g h i j k l
$ myecho "I cannot" "seem to" "find my KEYS."
```

13. Create a program **my_vi** that will accept a command-line argument which 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.

Answer:

```
#!/usr/bin/sh
# my_vi: Create a backup file prior to starting a vi session
# usage: my_vi filename
#
echo Copying $1 to ${1}.bak
cp $1 ${1}.bak
vi $1
echo Edit of $1 is complete
echo You may recover your original file from ${1}.bak
```

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**.

Answer:

```
#!/usr/bin/sh
# my_recover: Recover a file from backup
# usage: my_recover filename
echo Restoring $1 from ${1}.bak
cp ${1}.bak $1
echo $1 is recovered
```

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.

Answer:

```
#!/usr/bin/sh
# info: Prompt user for mailing address
#
echo "Input your name: \c"
read name
echo "Input your street address: \c"
read address
echo "Input your City, State, and Zip Code: \c"
read where
echo;echo
```

```
echo "Your name is  $name"
echo "You live at   $address"
echo "              $where"
```

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.

Answer:

```
#!/usr/bin/sh
# home: Return the value of a user"s HOME directory
# usage: home
echo Select a user identifier from the following list:
cut -f1 -d: /etc/passwd | pr -4 -t
echo "Input user identifier: \c"
read user
home=$(grep $user /etc/passwd | cut -f6 -d:)
echo;echo "user:$user HOME directory: $home"
```

17. Write a shell program called **alpha** that will display the first and last command line arguments. Hint: use the **cut** command.

Answer:

```
#!/usr/bin/sh
# alpha: Displays the first and last command line arguments
#
last=$(echo $* | cut -f$# -d" ")
echo "The first command line argument is $1."
echo "The last command line argument is $last."
```

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.

Answer:

```
#!/usr/bin/sh
# file_copy: User interface for copying files
# usage: copy
#
echo Please enter the names of the file(s) you want to copy:
echo "-> \c"
read filenames
echo Please enter the destination.
banner NOTE!
echo If you entered more than one file, the destination must be a
directory.
echo "Enter destination here -> \c"
read dest
```

```
echo Copying files now ...
cp $filenames $dest
echo Done copying files "\a"
```

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.

Overview of Module 17

Audience

general user General system user

Product Family Type

open sys Open systems environment

Abstract

This module presents shell return codes and is designed to teach the use of the **`test`** command and the **`if`** and **`case`** constructs for conditional branching within a shell program.

Time

Lab 45 minutes

Lecture 60 minutes

Prerequisites

m1309m Introduction to Shell Programming

Instructor Profile

UX General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000 Logon access to an HP-UX system

Software Requirements

UX11 HP-UX release 11.0

Material List

P/N B2355-90033(T)	*HP-UX Reference Manual* , one per terminal
P/N B2355-90046(T)	*HP-UX Shells: User's Guide* , one per terminal

Lab Instructions

setup1	Create user logon of **`user1`**, **`user2`**, ... **`user`***n*, where *n* is the number of students in the class. Set up one user per student.

17-1. SLIDE: Return Codes

Return Codes

The shell variable `?` holds the return code of the last command executed:

0: command completed without error (true)
non-zero: command terminated in error (false)

Example:

```
$ true                  $ false
$ echo $?               $ echo $?
0                       1
$ ls                    $ cp
$ echo $?               Usage: cp f1 f2
0                              cp [-r] f1 ... fn d1
$ echo $?               $ echo $?
0                       1
                        $ echo $?
                        0
```

a566180

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.

17-1. SLIDE: Return Codes — Instructor Notes

Key Points

- *All* commands will return some value.
- Success returns **0**, Error returns non-zero.
- **true** returns **0**, **false** returns **1** (non-zero).
- **echo $?** will display the return value of the previous command.

Teaching Tips

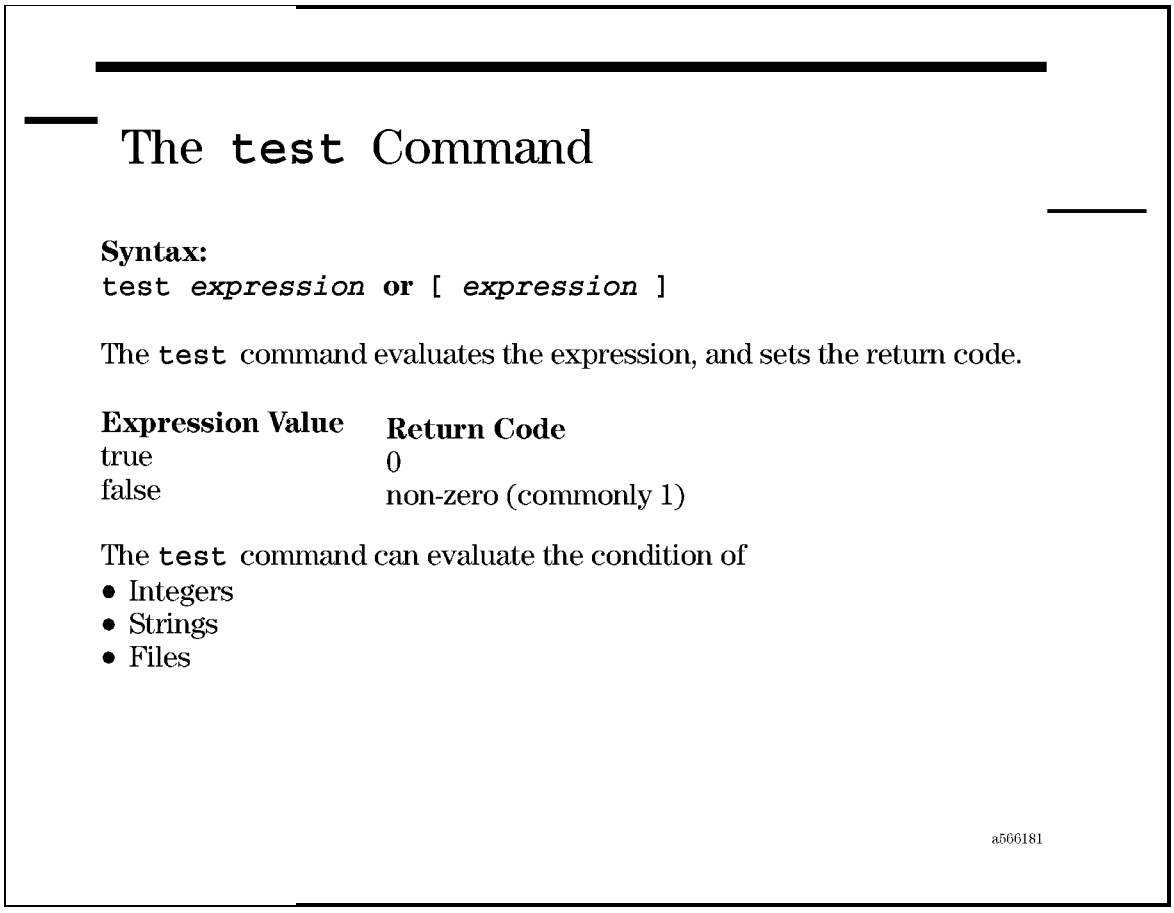
Students must understand the basic concept of return codes *before* progressing to the **test** command and conditional analyses.

Students who have previous programming experience will often find this logic contrary to their programming paradigms. For example in C, *true* is defined as non-zero, and *false* is defined as **0**.

Note that a return code of non-zero does not always indicate that an error has occurred—it can also indicate a special condition, as with the **grep** command:

```
# Pattern found - $? = 0
$ grep stu01 /etc/passwd
stu01:*:201:20::/home/stu01:/usr/bin/sh
$echo $?
0

# Pattern not found - $? = 1
$ grep xxxxx /etc/passwd
$echo $?
1

# Syntax error - $? = 2
$ grep -z stu01 /etc/passwd
grep: illegal option --- z
usage:
grep [-E|-F] [-cbilnqsvx] [-e expression [-e expression] ... | -f file]
[expression] [file ...]
$echo $?
2
```

17-2. SLIDE: The `test` Command



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 **`]`**.

17-2. SLIDE: The `test` Command — Instructor Notes

Key Points

- The **`test`** command will evaluate conditional expressions and conditional file tests.
- It will set the return code of the **`test`** command, based on whether the expression is true (**`0`**) or false (**`1`**) (including Bourne shell).
- This shell intrinsic is the reason you can never call a shell program **`test`**

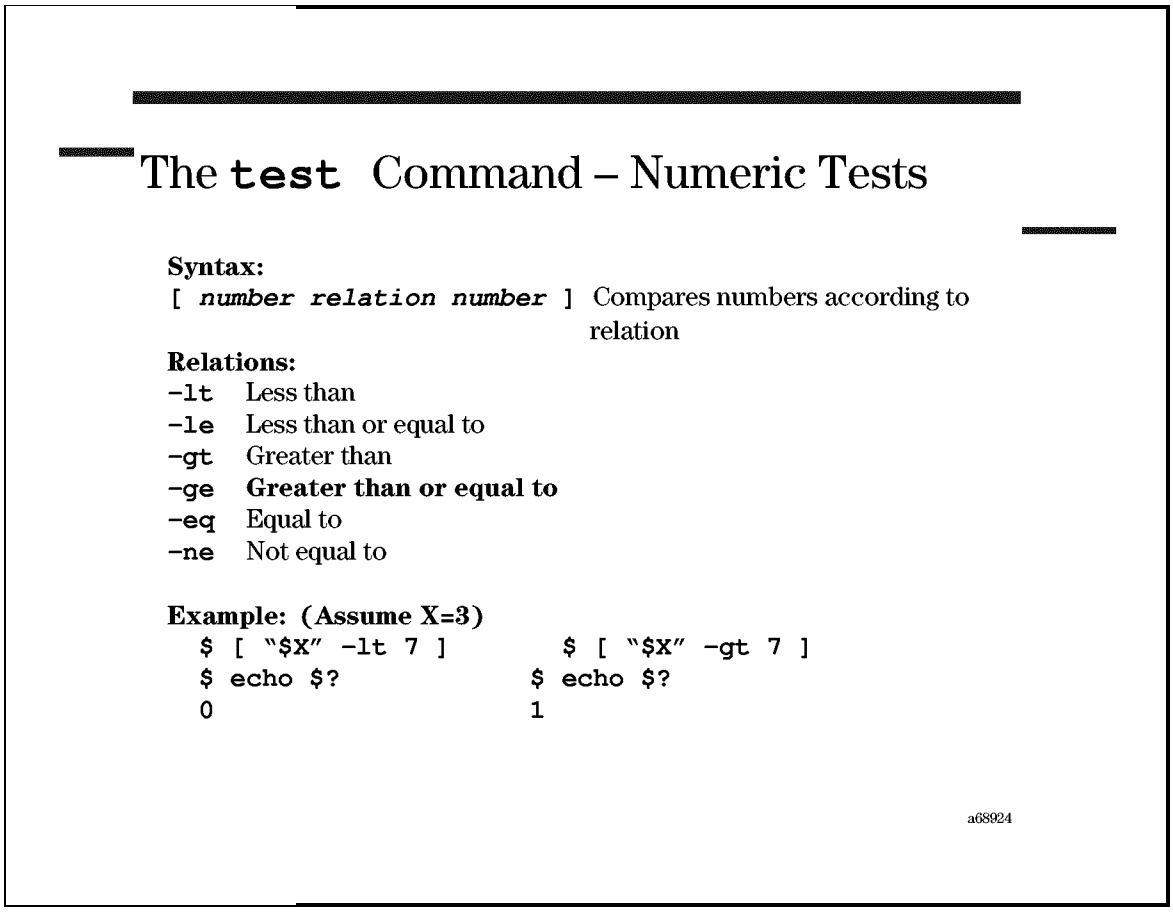
Teaching Tips

Point out that the test command has two syntaxes:

```
test args

[ args ]
```

NOTE: *There must be white space around each of the* **`[ ]`**.

17-3. SLIDE: The `test` Command — Numeric Tests



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
```

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 general rule, you should surround all **`$variable`** expressions with double quotation marks to avoid improper variable substitution by the shell.

17-3. SLIDE: The `test` Command — Numeric Tests — Instructor Notes

Key Points

- The shell is restricted to integer mathematics.
- You must interrogate the return value to determine if the condition is true or false.
- Shell variables that are not defined should be enclosed in quotation marks to guarantee that you do not receive a shell error message.

```
$ a=0
$ [ $a -eq $b ]
sh: test: argument expected
```

- Since both shell variables are not defined, *b* will be initialized to NULL, which is why we receive the syntax error.
- The shell references all variables as a text string unless the context indicates numerical usage. These test relational options will cause the shell to perform numerical operations on the operands.

Teaching Tips

Have the students type in the examples from the slide. Remind them to display the value of the return code with **`echo $?`**.

The following represents an equivalent expression:

```
$ x="03"
$ y="3"
$ [ "$x" -eq "$y" ]
$ echo $?
0
```

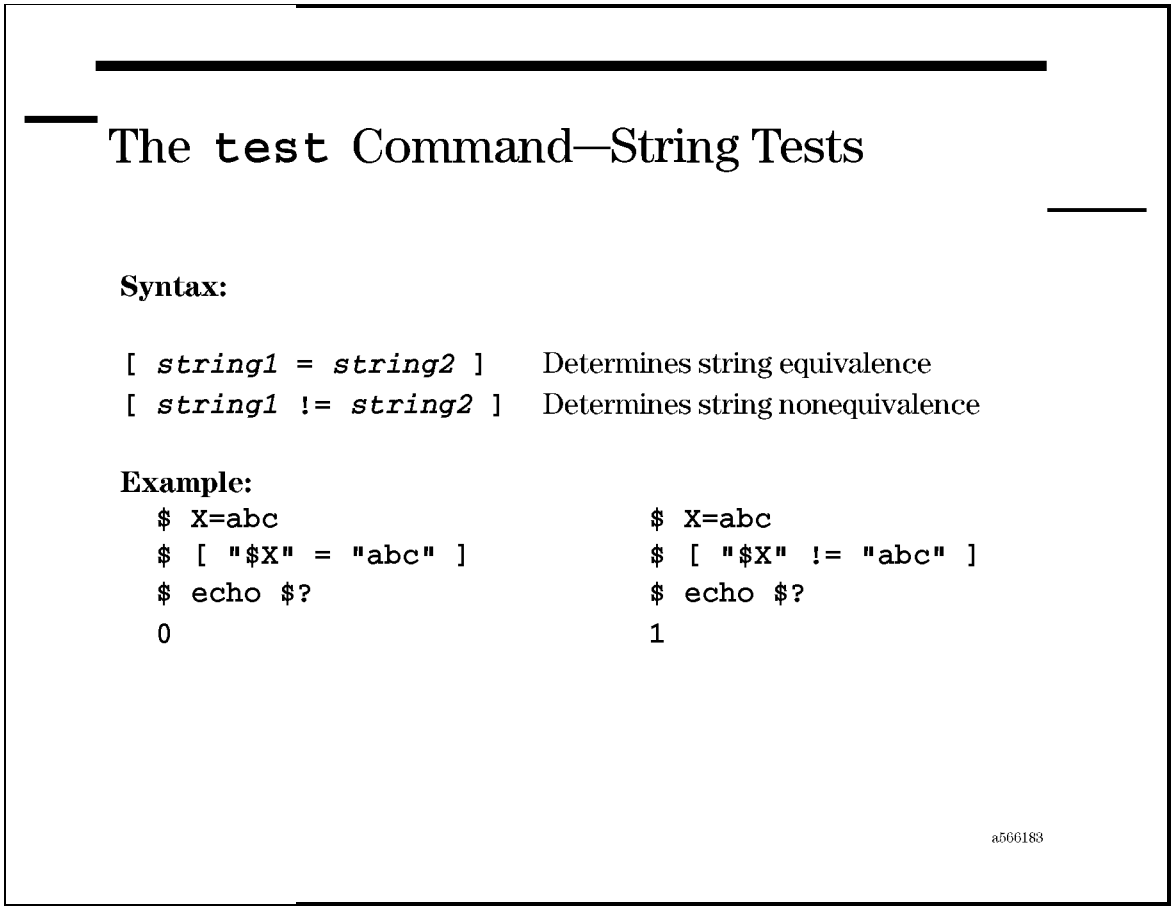
This example is contrasted with string comparisons in the Student Notes of the next slide.

Teaching Questions

QUESTION: What test would you use to guarantee that a program was invoked with three command line arguments?

ANSWER: [$# -eq 3]

17-4. SLIDE: The `test` Command — String Tests



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.

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
```

17-4. SLIDE: The `test` Command — String Tests — Instructor Notes

Key Points

- `=` and `!=` perform string comparisons.
- Spaces are required around *all* arguments.
- Provide quotes around variables that may be NULL or contain blanks.
- Numerical tests are not the same as string tests.

Teaching Questions

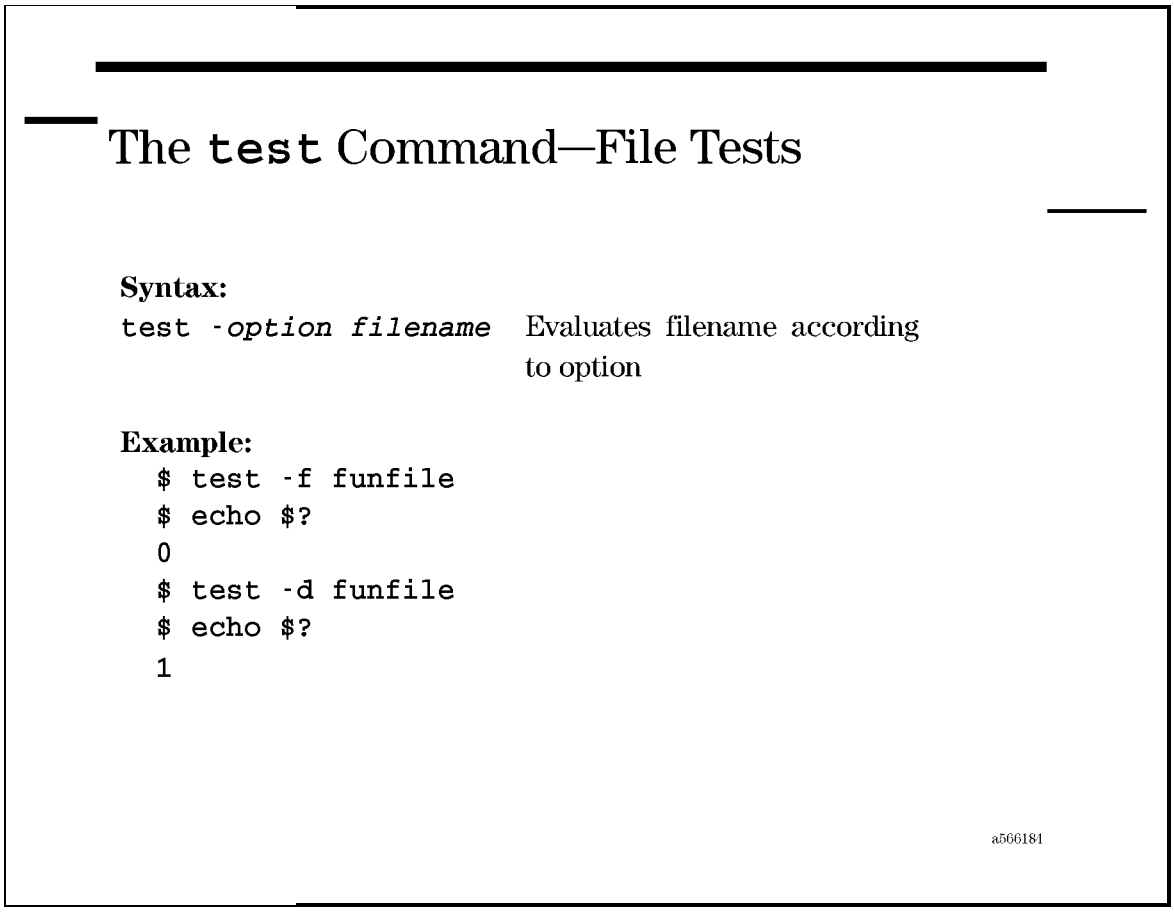
QUESTION: What happens if the `$` *variable* is surrounded with apostrophes?

ANSWER: The *variable* would *not* be replaced with its value.

QUESTION: How would you determine if the first command line argument is a `-m`?

ANSWER: `[ "$1" = "-m" ]`

17-5. SLIDE: The `test` Command — File Tests



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 writeable.
-x *file*	True if the *file* exists and is executable.
-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.

17-5. SLIDE: The `test` Command — File Tests Instructor Notes

Key Points

- These options test file characteristics.
- They provide useful options to test for file existence and file access.

Teaching Tips

The POSIX/Korn shell has several intrinsic testing options in addition to the capabilities provided by the **`test`** command. A couple of useful examples include

[**`-L`** *file*]	True if *file* is a symbolic link
[*file1* **`-nt`** *file2*]	True if *file1* is newer than *file2*
[*file1* **`-ot`** *file2*]	True if *file1* is older than *file2*
[*file1* **`-ef`** *file2*]	True if *file1* has the same device and inode number as *file2*, meaning that both refer to the same physical file.

POSIX Shell Only

[**`-e`** *file*]	True if *file* exists.

The conditional command **`[[`** *test_expression* **`]]`**can also be used, where *test_expression* is a combination of the above conditional primitives combined with the *and* operator, **`&&`**, the *or* operator, ||, and the *negation* operator, **`!`** (like in C). See **`test(1)`** for more information.

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.

a566185

Student Notes

Multiple conditions can be tested for by using the Boolean operators.

Table 17-1.

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)

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 \( -d "$2" -o -s "$2" \) ]
```

17-6. SLIDE: The `test` Command — Other Operators — Instructor Notes

Key Points

- The Boolean operators allow you to test for multiple conditions.
- The parentheses must be escaped.

Teaching Tips

You should go through the truth table for those students who have not previously done Boolean evaluations.

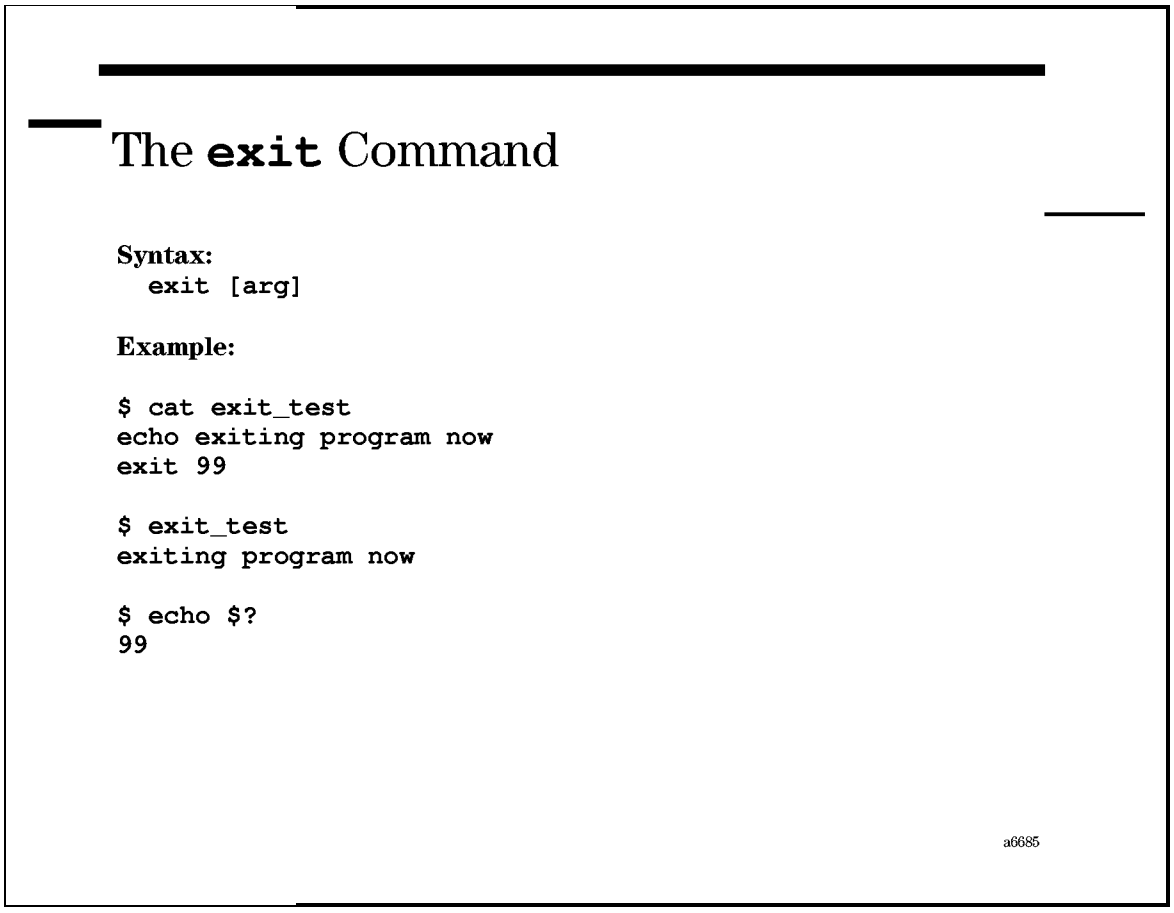
Go through the examples in the student notes.

The normal use for parentheses in the shell is for defining shell functions (presented in the module *Introduction to Shell Programming*) and command grouping. With respect to command grouping,

```
$ cmd1 ; cmd2 &          run cmd1 in foreground, then run cmd2 in background

$ ( cmd1 ; cmd2 ) &      run both cmd1 and cmd2 in background
```

The use of parentheses also causes the grouped commands to be run in a subshell. Backslashing the parentheses allows them to be used for expression evaluation.

17-7. SLIDE: The `exit` Command



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.

17-7. SLIDE: The `exit` Command — Instructor Notes

Key Points

- The **`exit`** command is used to set the return codes of shell programs and shell functions, respectively.
- It is commonly used with branching constructs to terminate the program or function upon recognition of an error condition. The argument will refer to the error encountered.

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
```

a566187

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* **0** (FALSE), jump to **fi** and continue with the first statement following the **fi**.

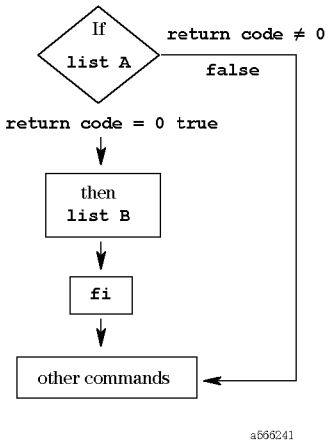


Figure 17-3. The if Construct Flowchart

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
```

17-8. SLIDE: The `if` Construct — Instructor Notes

Key Points

- The **`if`** construct is used to make decisions on what commands should be executed based on the outcome of some test.
- The test is based on the *return code* of a command.
- A list of commands can be provided, but the decision is based on the *last* command in list A.
- The **`test`** command is most commonly used to analyze the condition of variables, strings, files, etc.
- The **`if, then`** and **`fi`** keywords are required for any **`if`** construct.
- The branching constructs are concluded with the branch keyword in reverse.

Teaching Tips

Go through the example on the slide. Describe what would be printed when the test is true and when the test is false.

Also go through the examples in the student notes.

Inform the students that the **`if`** command can be issued interactively or programmatically. If entered at the command prompt, the shell will issue secondary prompts until the **`fi`** construct is entered, and then execute the entire **`if`** construct. Have the students try the example on the slide and the **`grep`** example from their notes interactively. Note that **`/dev/null`** is used to supress the intermediate display. This provides an opportunity to discuss uses of **`/dev/null`**.

Note that the **`grep`** command will return **`0`** if the pattern is found, **`1`** if the pattern is not found, and **`2`** if some other error was encountered.

17-9. SLIDE: The `if-else` Construct

The `if-else` Construct

Syntax: (used for multi-decision branch)

```
if
    list A
then
    list B
else
    list C
fi
```

Example:

```
if [ "$X" -lt 10 ]
then
   echo X is less than 10
else
   echo X is not less than 10
fi
```

a68917

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`**.

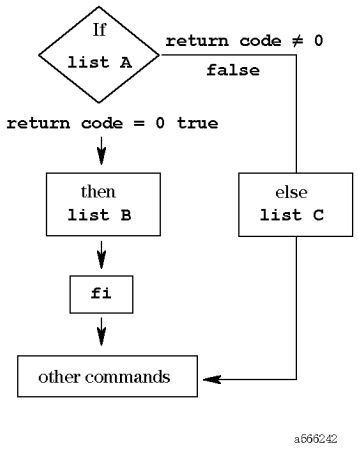


Figure 17-4. The if-else Construct Flowchart

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**.

17-9. SLIDE: The `if-else` Construct — Instructor Notes

Key Points

- The **`else`** construct allows for the generation of two-way or more conditional branches, depending on how many **`if`**s you nest.
- Every **`if`** requires an **`fi`**.

Teaching Tips

Be sure to present the example found in the student notes, demonstrating how an **`if`** statement can be nested within an **`if`** statement.

You might want to mention the **`if-elif-fi`** construct. **`elif`** is a contraction for **`else if`**, as in

```
if
    list A
then
    list B
elif
    list C
then
    list D
else
    list E
fi
```

This can be used to lead into the need for multi-decision branching and the case statement.

First, the nested **`if-then-else`**. Notice that each **`if`** must have a **`fi`**.

```
if [ "$color" = red ]
then
    echo apple
else
    if [ "$color" = yellow ]
    then
        echo banana
    else
        if [ "$color" = orange ]
        then
            echo kumquat
        else
            echo vegetable!
        fi
    fi
```

```
fi
```

Next, the **elif** contraction is applied. Notice only one **fi** needed.

```
if [ "$color" = red ]
then
    echo apple
elif [ "$color" = yellow ]
then
    echo banana
elif [ "$color" = orange ]
then
    echo kumquat
else
    echo vegetable!
fi
```

Now the **case**:

```
case "$color" in
red)    echo apple
        ;;
yellow) echo banana
        ;;
orange) echo kumquat
        ;;
*) echo vegetable!
        ;;
esac
```

17-10. SLIDE: The `case` Construct

The `case` Construct

Syntax: (multi-directional branching)

```
case word in
pattern1)  list A
             ;;
pattern2)  list B
             ;;
patternN)  list N
             ;;
esac
```

Example:

```
case $ANS in
     yes)  echo O.K.
           ;;
      no)  echo no go
           ;;
esac
```

```
case $OPT in
     1)  echo option 1  ;;
     2)  echo option 2  ;;
     3)  echo option 3  ;;
     *)  echo no option ;;
esac
```

a566189

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.
[...]	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.

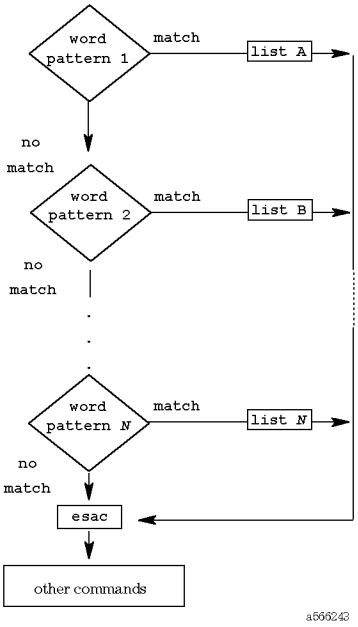


Figure 17-5. The case Construct Flowchart

17-10. SLIDE: The `case` Construct — Instructor Notes

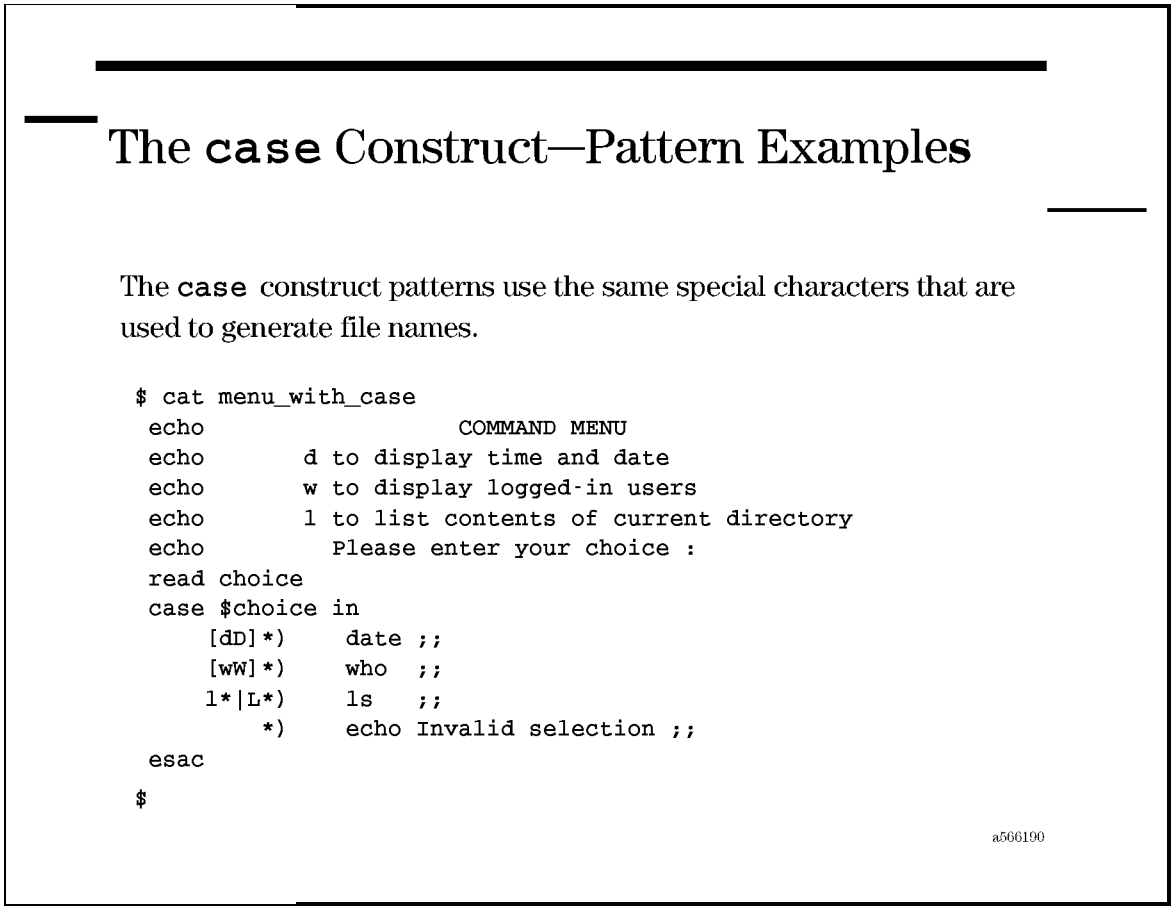
Key Points

- **`case`** is used for multi-way branching.
- **`case`** performs a string equivalence test for each pattern against the *word*.
- The *word* is commonly the value of a shell variable.
- **`case`** is commonly used to support a menu interface, where the user will enter one of many options.
- As illustrated in the second example, the semicolons can be placed at the end of each case, rather than on a separate line.
- In the final case only, the semicolons can be omitted.
- The default (**`*`**) case can appear anywhere in the list (not just at the end) and will be evaluated correctly.

The **`case`** example follows:

```
case "$color" in

red)    echo apple
        ;;
yellow) echo banana
        ;;
orange) echo kumquat
        ;;
*) echo vegetable!
        ;;
esac
```

17-11. SLIDE: The `case` Construct — Pattern Examples



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.

17-11. SLIDE: The `case` Construct — Pattern Examples — Instructor Notes

Key Points

- Patterns are supported to make the interface more flexible for user input.
- The patterns are generated with the same special characters as file name generation.
- It is common to conclude each case pattern list with **`*)`** to catch any response that was not supported by the available options. An error message should be displayed to the user.
- Note: the patterns *do not* support mathematical tests, such as less than, greater than, and so on. You must use **`if`** statements for mathematically-oriented decisions.

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
      command listA
then
      command listB
else
      command listC
fi
```

Decision based on *return code* of last command in *listA*

```
case word in
pattern1) command list
;;
pattern2) command list
;;
*)        command list
;;
esac
```

The string *word* is compared to each string pattern

a68918

Student Notes

17-12. SLIDE: Shell Programming — Branches — Summary Instructor Notes

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 which 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.)

5. Write a shell program which 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.)

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`** which 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`** which 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`**.

17-13. LAB: Shell Programming — Branches Instructor Notes

Time: 45 minutes

Purpose

To practice using the **test**, **if**, and **case** commands for branching within shell programs.

Notes to the Instructor

Introductory Exercises	1–6
Intermediate Exercises	7–10
Advanced Exercises	11

The *Introductory* exercises will be useful for students who need to practice the basic **test** and branching constructs. They provide exercises that are similar to the examples presented in the slides.

The *Intermediate* exercises are for students who feel reasonably comfortable with the basic constructs, and allow them to exercise their understanding of variables, pipelines, and command substitution along with the branching constructs. Most students will probably work on these four exercises.

The *Advanced* Exercise allows the students to create a shell program that accepts command line options.

Solutions

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.)

Answer:

```
$ X=xyz
$ test "$X" = "xyz"
$ echo $?
0
```

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.)

Answer:

```
$ Y=100
$ test "$Y" -gt 0    or    [ "$Y" -gt 0 ]
$ echo $?
```

```
0

$ Y=-100
$ test "$Y" -gt 0    or    [ "$Y" -gt 0 ]
$ echo $?
1
```

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.

Answer:

```
if
    [ "$1" = "abc" ]
then
    echo yes
else
    echo no
fi
```

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 which 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.)

Answer:

```
echo "please enter a number: \c"
read Y
if
    [ "$Y" -gt 0 ]
then
    echo Y is positive
    echo The value of Y is $Y
 else
    echo Y is not positive
    echo The value of Y is $Y
fi
```

5. Write a shell program which 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?)

Answer:

```
if
    [ "$#" -ne 3 ]
then
    echo "there are not exactly three command line arguments" >&2
else
    echo $*
fi
```

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.

Answer:

```
echo "Please input A, B, b, or C: \c"
read input
case $input in
         A) echo good morning
            ;;
      [Bb]) echo good afternoon
            ;;
    C|quit) exit
            ;;
         *) echo You entered an illegal option.
            exit 99
            ;;
esac
```

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.

Answer:

```
echo "Input a user login name -> \c"
read user
if
    grep $user /etc/passwd > /dev/null
then
    home=$(grep $user /etc/passwd | cut -f6 -d:)
    echo The HOME directory for $user is $home
else
    echo;echo "$user is not here!!!"
fi
```

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.)

Answer:

```
time=$(date | cut -c12-20)
hour=$(echo $time | cut -f1 -d:)

if [ $hour -lt 12 ]
then
    echo good morning
else
    if [ $hour -ge 12 -a $hour -lt 18 ]
    then
        echo good afternoon
    else
        echo good evening
    fi
fi
```

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.

Answer:

```
echo Would you like to see the contents of your current directory?
echo Please enter yes or no.
echo "----> \c"
read ans1
case $ans1 in
     yes) ls
          ;;
      no) echo What directory would you like to see?
          read ans2
          if test -d $ans2
          then
              ls $ans2
          else
              echo directory $ans2 does not exist
          fi
          ;;
       *) echo You have not entered a proper response.
          echo Please try again.
          ;;
esac
```

10. Create a program **mycp** which 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.

Answer:

```
#!/usr/bin/sh
# mycp file1 file2
if [ "$#" -ne 2 ]
    then
        echo "Usage: $0 file1 file2" >&2
        exit 1
fi
if [ "$1" = "$2" ]
then
    echo "$1 = $2"
    echo "No copy performed" >&2
    exit 2
fi

if test -d $2
then
    echo "Your target is a directory." >&2
    echo "No copy performed!" >&2
    exit 4
fi

if test -f $2
then
    echo "Your target file already exists."
    echo "Do you want to overwrite it? [y/n]: \c"
    read ans
    if [ "$ans" != "y" -o "$ans" != "Y" ]
    then
        echo "No copy performed!" >&2
        exit 3
    fi
fi

if test -f $1
then
    cp $1 $2
    echo "Copy complete"
else
    echo "Source file does not exist"
    echo "No copy performed"
    exit 4
fi
```

11. Write a shell program called **`options`** which 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**.

Answer:

```
if
    [ "$#" -lt 1 -o "$# "-gt 2 ]
then
    echo "usage: $0 -d" >&2
    echo " $0 -l directory" >&2
    echo " $0 -w" >&2
    exit 10
fi
case $1 in
     -d) date
         ;;
     -w) who
         ;;
     -l) if test -d $2
         then
             echo the contents of directory $2 are:
             ls -F $2
         else
             echo directory $2 does not exist
         fi
         ;;
      *) echo "bad option" >&2
         echo "usage: $0 -d" >&2
         echo " $0 -l directory" >&2
         echo " $0 -w" >&2
         exit 20
         ;;
esac
```

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.

Overview of Module 18

Audience

general user General system user

Product Family Type

open sys Open systems environment

Abstract

This module is designed to teach the student the **`while`**, **`until`** and **`for`** constructs for looping in a shell program. Also the **`let`** command, **`break`** and **`continue`** are discussed.

Time

Lab 90 minutes

Lecture 60 minutes

Prerequisites

m1310m Shell Programming—Branches

Instructor Profile

UX General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000 Logon access to an HP-UX system

Software Requirements

UX11 HP-UX release 11.0

Material List

P/N B2355-90033(T)	*HP-UX Reference Manual*, one per terminal

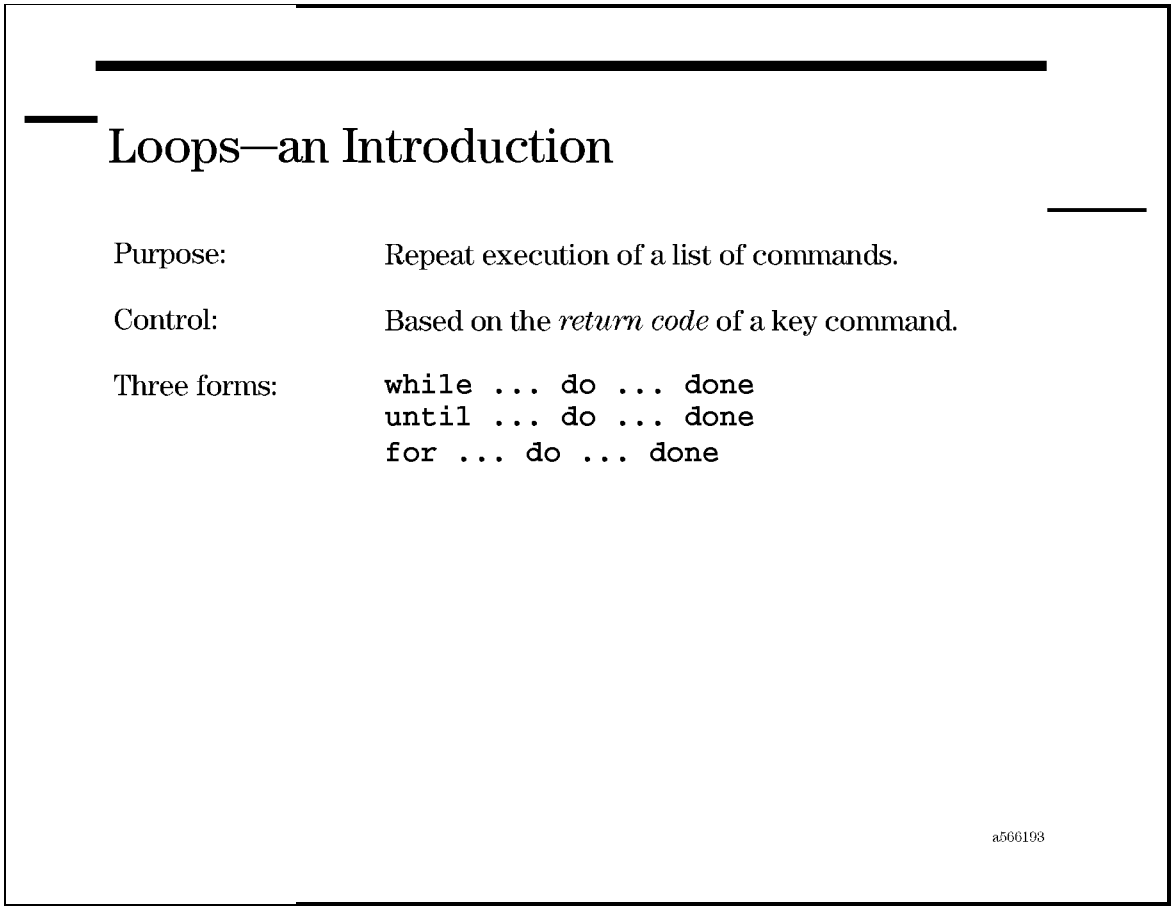
Lab Instructions

setup1	Create user logon of **user1**, **user2**, ... **user***n*, where *n* is the number of students in the class. Set up one user per student.
copyfiles	Copy the lab files to the users' home directories.

Lab Files

```
-rw-r----- 1 karenk users   3081 May 28 16:12 funfile
```

18-1. SLIDE: Loops — an Introduction



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`**.

18-1. SLIDE: Loops — an Introduction Instructor Notes

Key Points

- Similar to branching control, loop control is based on the return code of a command, commonly the **`test`** command.
- Point out that loops *do not* start with a keyword and end with the keyword in reverse. Loops are initiated with a keyword and a condition, and the body of the loop is surrounded by **`do/done`**.

18-2. SLIDE: Arithmetic Evaluation Using `let`

Arithmetic Evaluation Using `let`

Syntax:

`let` *expression* or `((` *expression* `))`

Example:

```
$ x=10                          $ x=12
$ y=2                           $ let "x < 10"
$ let x=x+2                     $ echo $?
$ echo $x                       1
12                              $ (( x > 10 ))
$ let "x = x / (y + 1)"         $ echo $?
$ echo $x                       0
4                               $ if (( x > 10 ))
$ (( x = x + 1 ))               >   then echo x greater
$ echo $x                       >   else echo x not greater
5                               > fi
                                x greater
```

a566194

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
== !=	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.

18-2. SLIDE: Arithmetic Evaluation Using `let` — Instructor Notes

Key Points

- The **`let`** command is used to perform mathematical operations.
- This is often required to increment a variable that is being tested to determine the continuance of a loop.

Teaching Tips

Note that the dollar sign can be used with variables with the **`let`** command, but it is not necessary. Either of the following is legal:

```
let x=x+1 OR let x=$x+1
```

Note that when special shell variables are referenced, the dollar sign must be used:

```
if (( $# <= 10 ))
```

The **`expr`** command may be used instead of **`let`** for Bourne shell portability:

```
x=12
$ expr $x + 1
13
$ x=`expr $x + 1`
$ echo $x
14
```

18-3. SLIDE: The `while` Construct

The `while` Construct

Repeat the loop while the condition is true.

Syntax:

```
while
    list A
do
    list B
done
```

Example:

```
$ cat test_while
X=1
while (( X <= 10 ))
do
   echo hello X is $X
   let X=X+1
done

$ test_while
hello X is 1
hello X is 2
  .
  .
  .
hello X is 10
```

a566195

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
```

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.

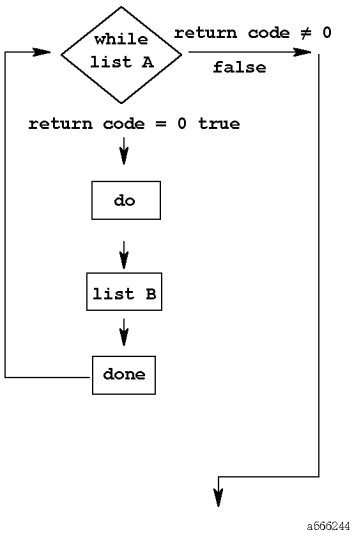


Figure 18-6. The while Construct Flowchart

WARNING: **Be careful of infinite `while` loops. These are loops whose controlling command *always* returns true.**

```
$ cat while_infinite
while
     true
do
     echo hello
done

$ while_infinite
hello
hello
  .
  .
  .
```

Ctrl + c

18-3. SLIDE: The `while` Construct — Instructor Notes

Key Points

- The **`while`** loop is controlled on the basis of a return code, just like the branching mechanisms.
- The body of the loop is encased in the **`do/done`** constructs.
- The while loop is commonly controlled by a *counter* variable that will be incremented with the **`let`** or **`expr`** command.
- When a false condition is encountered, the program continues with the first command beyond the **`done`**.
- Other branching and looping constructs can be nested within a **`while`** loop.
- Warn the students against infinite **`while`** loops.

Teaching Tips

You might want to let the students know about the POSIX/Korn shell's capability to recall a previous command and start a full **`vi`** session to modify the command entered. This is especially convenient on the multi-line commands that are entered in the branching and looping modules.

Recall a previous construct using the ESC and **`k`** keys. When the multi-line command is displayed, you will see many control characters, that represent the new lines for the input. Simply enter **`v`**, and a full **`vi`** session will be started on the command. You can now make any necessary modifications.

Use **`:wq`** or **`ZZ`** to quit the edit session. When your file has been written the associated commands will be executed.

NOTE: The editor that will be invoked is identified by the $FCEDIT or $EDITOR variable. If you are running under X Windows you could set the variable to **`ved`** and invoke a window-based editor to modify previous lines.

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
```

a566196

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.

18-4. SLIDE: The `while` Construct — Examples — Instructor Notes

Key Points

- Loops are commonly controlled by user input — Example A.
- A variable number of command line arguments can be handled by looping and shifting through the command line arguments — Example B.
- Note: once a command line argument is shifted from the command line, it cannot be retrieved again. If you need to preserve the original command line arguments, you may want to make the assignment:

```
args=$*
```

- The controlling variable in a **`while`** loops may need to be properly initialized to guarantee that the body of the loop will execute at least once, as seen in Example A.

18-5. SLIDE: The `until` Construct

The `until` Construct

Repeat the loop until the condition is true.

Syntax:

```
until
    list A
do
    list B
done
```

Example:

```
$ cat test_until
X=1
until (( X > 10 ))
do
   echo hello X is $X
   let X=X+1
done

$ test_until
hello X is 1
hello X is 2
   .
   .
   .
hello X is 10
```

a62810

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.

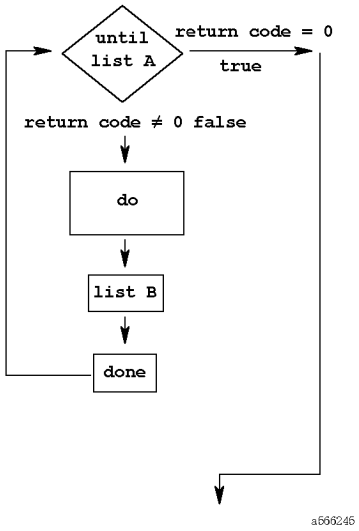


Figure 18-7. The until Construct Flowchart

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

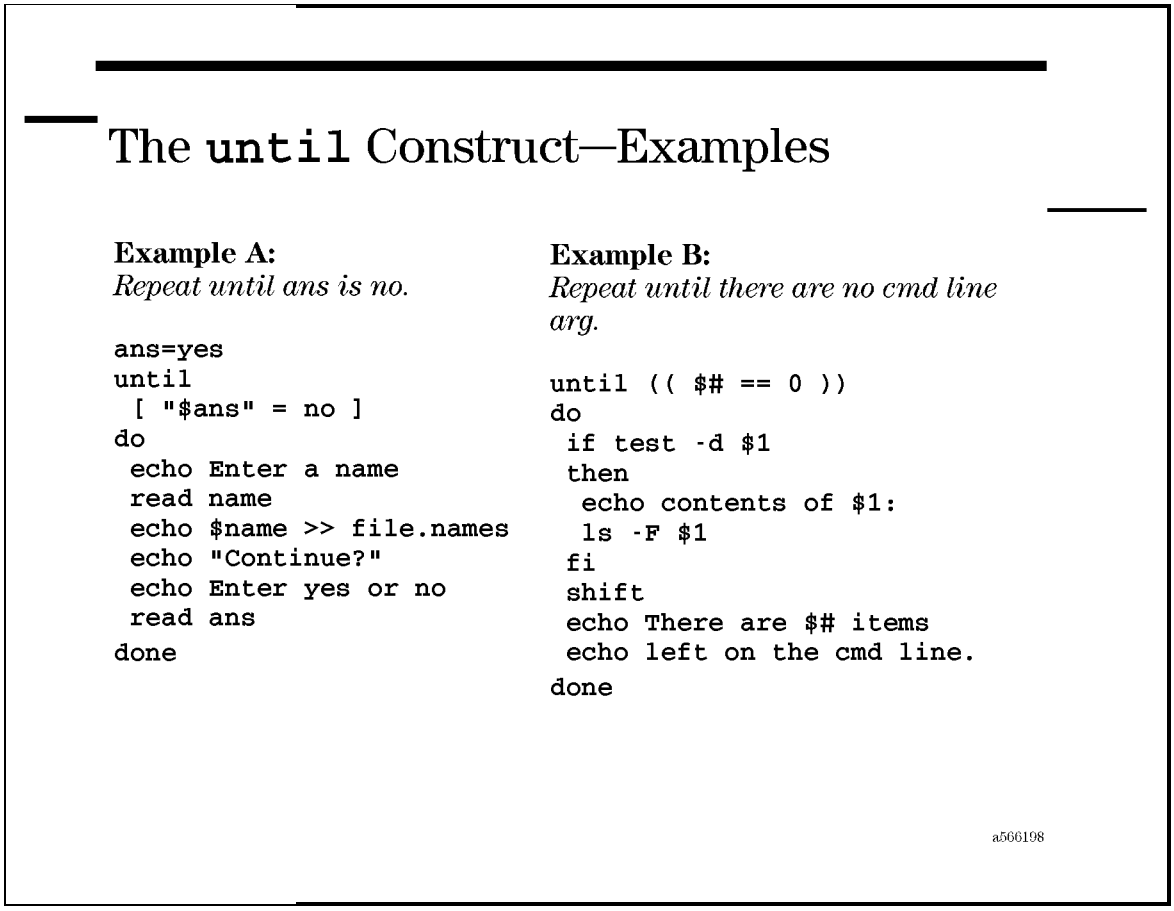
18-5. SLIDE: The `until` Construct — Instructor Notes

Key Points

- The **`until`** loop is controlled on the basis of a return code as are the **`while`** loop and the branching mechanisms.
- The body of the loop is encased in the **`do/done`** constructs.
- The **`until`** loop is commonly controlled by a *counter* variable that will be incremented with the **`let`** command.
- When a true condition is encountered, the program continues with the first command beyond the **`done`**.
- Other branching and looping constructs can be nested within an **`until`** loop.
- Warn the students against infinite **`until`** loops, as in the following:

```
$ X=1
$ until
>     [ $X -eq 0 ]
> do
>     echo hello
> done
hello
hello
hello
   .
   .
   .
Ctrl + c
```

18-6. SLIDE: The `until` Construct — Examples



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 a legal **`test`** syntax.

18-6. SLIDE: The `until` Construct — Examples — Instructor Notes

Key Points

- The logic for the **`while`** test conditions is reversed from the logic of the **`repeat until`** test conditions. The same operation can usually be implemented with either construct.
- Note the change in the sensitivity of input required between Example A of the **`repeat until`** loop and Example A of the **`while`** loop.
- The controlling variable of an **`until`** loop may need to be initialized to guarantee that the body of the loop will execute at least once, as in Example A.

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
```

a566199

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.

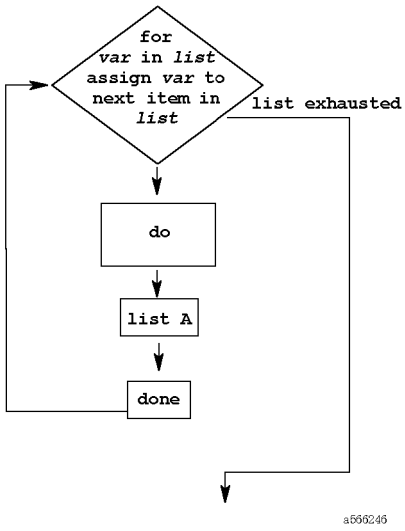


Figure 18-8. The for Construct Flowchart

18-7. SLIDE: The `for` Construct — Instructor Notes

Key Points

- The **`for`** construct is very powerful, since lists can be generated through command substitution and pipelines. Examples will be presented on the next slide.
- Point out that there is no easy way to count in a **`for`** loop. For example,

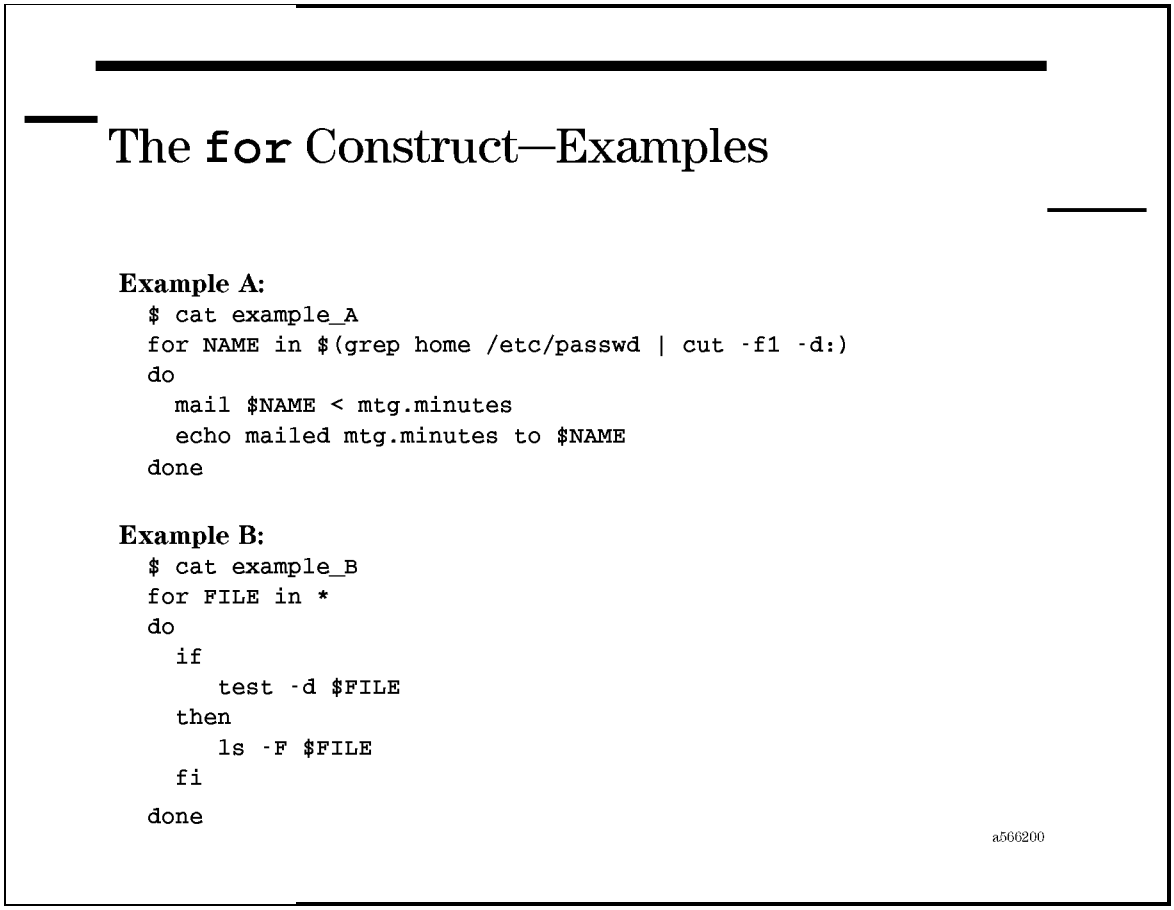
```
for i in 1 2 3 4 5 6 ... 200
do
...
done
```

  is not practical. Show the students that the construct

```
X=1
while ((X <= 200 ))
do
    ...
    let X=X+1
done
```

  is more efficient.

18-8. SLIDE: The `for` Construct — Examples



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

$ 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 $*
do
    cp $i $HOME/backups
done
```

or

```
for i
do
    cp $i $HOME/backups
done
```

18-8. SLIDE: The `for` Construct — Examples Instructor Notes

Key Points

- Lists can be provided explicitly, as seen on the previous slide or generated through other shell mechanisms:

  — command substitution
  — filename generation
  — command line arguments

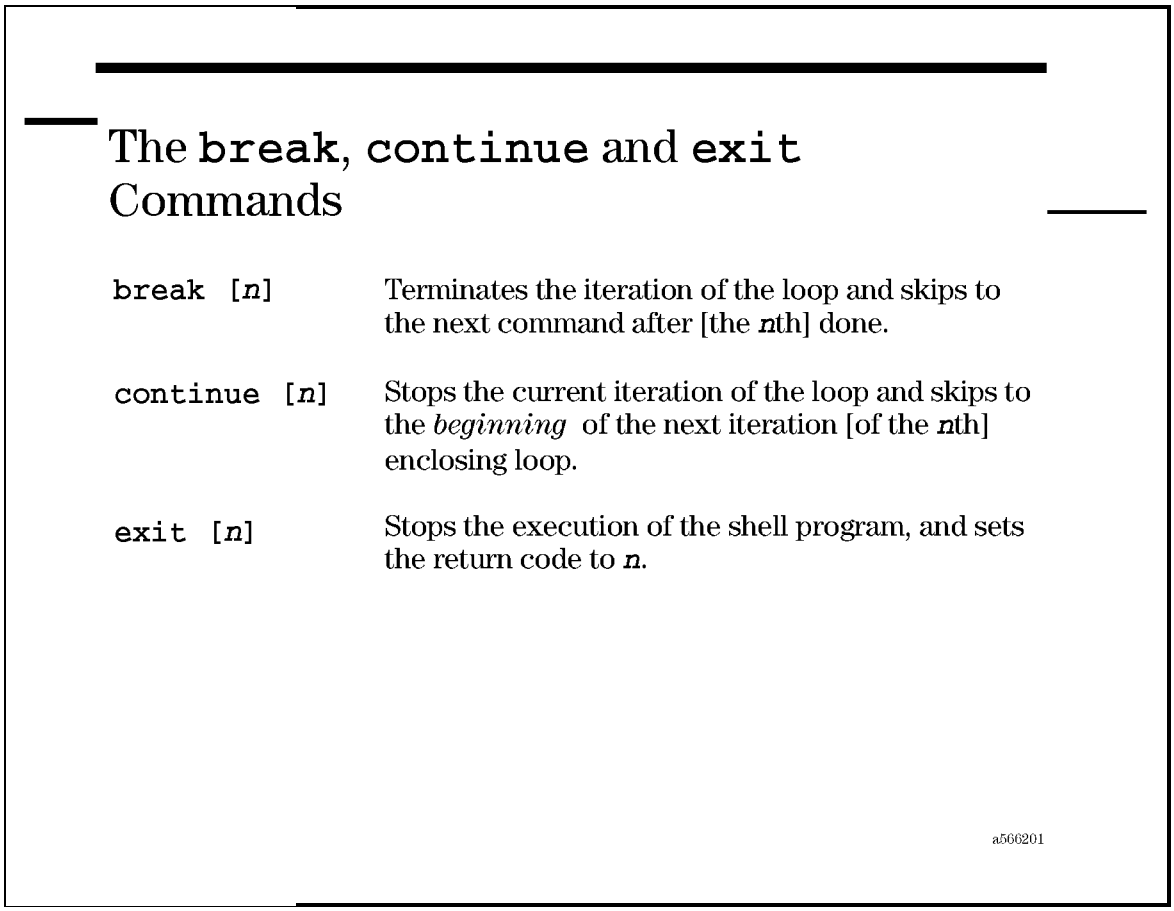
- The construct **`for x in $*`** is equivalent to **`for x`**.

- Saving a list in a file is a convenient method for repeated access.

Other Examples

```
$ cat example_C
for i in $(lsf | grep / | cut -d/ -f1)
do
    ls $i > $i.ls
done

$ cat example_D
for file in *
do
    size=$(wc -c $file | tr -s " " | cut -d " " -fl)
    echo $file has $size bytes
done
```

18-9. SLIDE: The `break`, `continue` and `exit` Commands



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.

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**?

18-9. SLIDE: The `break`, `continue` and `exit` Commands — Instructor Notes

Key Points

- The **`break`** will break out of the loop completely. Execution continues with the next command *after* the loop.
- The **`continue`** will just terminate the current iteration of the loop. Execution of the loop continues.
- The **`exit`** terminates the entire program.
- **`break`** and **`continue`** with arguments greater than **`1`** are not recommended. It makes following the flow of the code difficult. Use the example in the notes to illustrate.

Teaching Tips

Illustrate the differences of the break and continue with the code model provided in the student notes. An example is provided on the next page.

What command will be executed following the **`break 2?`**	cmdX
What if it is replaced simply with a **`break`**?	cmdJ
What about a **`continue 2?`**	**`while`** cmd1
What about a simple **`continue`**?	**`while`** cmdC

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
```

a566202

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.

18-10. SLIDE: `break` and `continue` — Example Instructor Notes

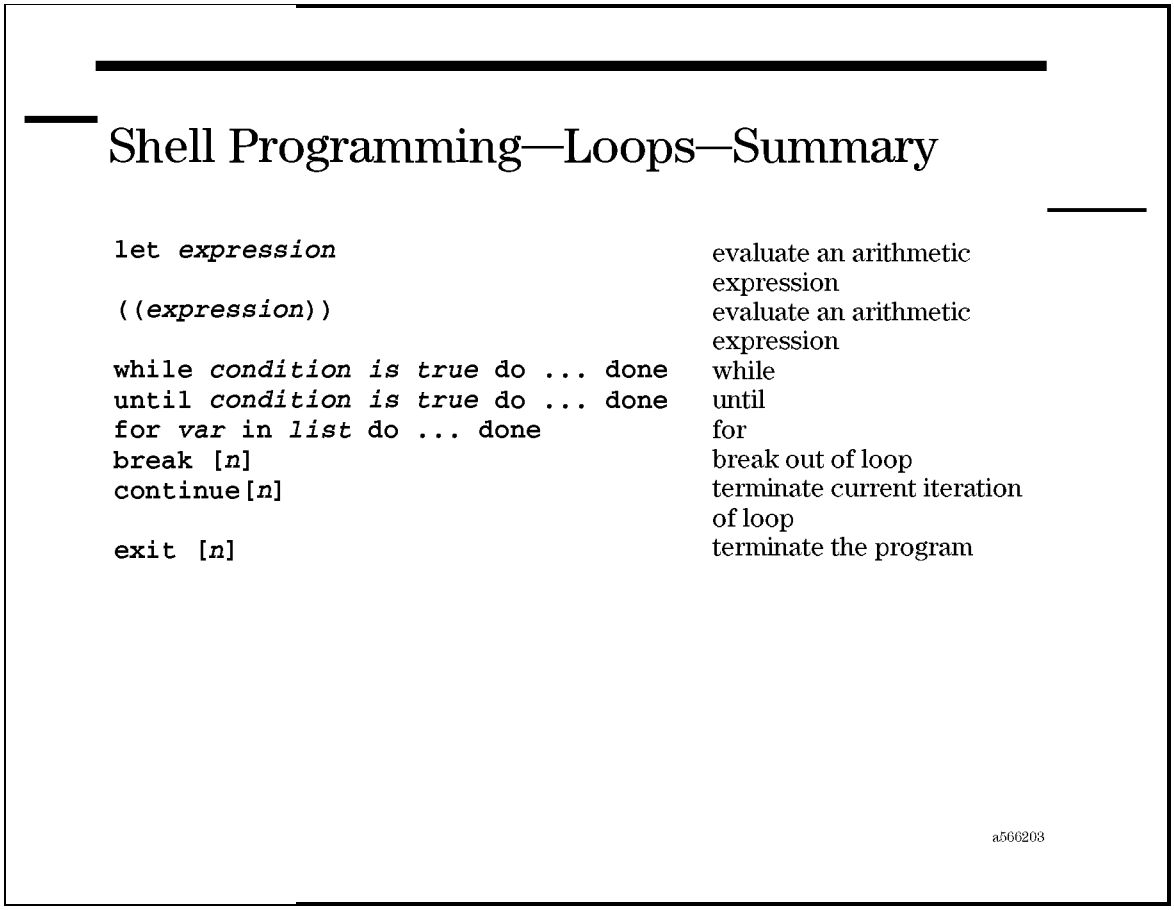
Key Points

- The **`continue`** will cause the current iteration of the loop to terminate, and the process will continue at the top of the loop where the user is prompted for a file name.
- The **`break`** command will cause the current iteration of the loop to terminate, breaking out of the loop altogether.
- Caution that the slide example **`safe remove`** will allow the user to enter file name-generating characters (including **`*`**) when prompted for "Enter a filename to remove." You might suggest including a confirmation question such as "Are you sure you want to remove *filename*?"

Teaching Tips

Be sure the students are aware of the dangers of using infinite loops.

18-11. SLIDE: Shell Programming — Loops — Summary



Student Notes

18-11. SLIDE: Shell Programming — Loops — Summary Instructor Notes

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

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.

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

18-12. LAB: Shell Programming — Loops — Instructor Notes

Time: 90 minutes

Purpose

To practice the looping constructs available in the shell.

Notes to the Instructor

Introductory Exercises	1–6, Recommend: 1, 2, 3, 4 or 5, 6
Intermediate Exercises	7
Advanced Exercises	8–10 Recommend: 8, 9 or 10

The *Introductory* exercises will be useful for students who need to practice with the basic **`let`** and looping constructs. They provide exercises that are similar to the examples presented in the slides.

The *Intermediate* exercise gives students practice creating menus.

The *Advanced* exercises Advanced Exercises, number 8 requires the user to send a message to a terminal. Exercise 9 directs the student to create a *background process*. Exercise number 10 is useful for students who would like to develop a script that protects them from improperly deleting files. This shell script provides the capability to recover *removed* files.

Solutions

1. Create a program called **`double_it`** that will prompt the user for a number and then display two times the number.

Answer:

```
#!/usr/bin/sh
# double_it: Prompt the user for a number and then display 2 times
#            its value.
#
echo "Input an integer value: \c"
read num
echo "Two times the number you entered is \c"
let num=num*2
echo $num
```

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.

Answer:

```
#!/usr/bin/sh
# sum_them: Prompt the user for 10 numbers and add them together
#
sum=0
count=1
echo You will be prompted to enter 10 numbers.
echo Their sum will be displayed after all 10 numbers have been entered.
while
     [ $count -le 10 ]
do
     echo "Please enter a number ($count): \c"
     read num
     let sum=sum+num
     let count=count+1
done
echo The sum of the 10 numbers you entered is: $sum
```

Optional solution supporting a command line argument identifying the number of numbers to enter:

```
#!/usr/bin/sh
# sum_them2: The user will provide the number of numbers to
#            add together as a command line argument
#
if
    [ $# -ne 1 ]
then
    echo Usage: $0 number >&2
    exit 99
fi

count=1
echo You will be prompted to enter $1 numbers.
echo Their sum will be displayed after all $1 numbers have been entered.
while
    [ $count -le $1 ]
do
    echo "Please enter a number ($count): \c"
    read num
    let sum=sum+num
    let count=count+1
done
echo The sum of the $1 numbers you entered is: $sum
```

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.)

Answer:

```
#!/usr/bin/sh
# words_in: prompt the user to input words until "quit" is entered
#
until
    echo Please enter a word. Enter "quit" when you are done.
    read input
    [ "$input" = quit ]
do
    words="$words\n$input"
done
echo $words
#Print words out in alphabetical order
echo $words | sort
```

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

Answer:

```
for name in Adir Bdir Cdir Ddir Edir
do
    mkdir $name
    cp $HOME/funfile $name
    ls $name
    echo done with $name
done
```

an alternative method could be:

```
for name in A B C D E
do
    mkdir ${name}dir
    cp $HOME/funfile ${name}dir
    ls ${name}dir
    echo done with $name
done
```

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.

Answer:

```
#!/usr/bin/sh
# new_files: create new files as provided by the command line arguments
# Usage: new_files f1 f2 f3 f4 ...
#
for i in $*
do
    echo creating file $i
    touch $i
done
```

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.)

Answer:

```
$ for i in $(who | cut -f1 -d" ")
> do
>     mail $i < mailtest
> done
```

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.

Answer:

```
#!/usr/bin/sh
# my_menu: A menu interface
# Usage: my_menu
#
until
   [ $ans -eq 3 ]
   clear
   echo
   echo
   echo
   echo 1) Double a number.
   echo 2) Add together 10 numbers.
   echo 3) Quit
   echo
   echo "Enter your selection number ->\c"
   read ans
do
   case $ans in
     1) double_it
```

```
        ;;
     2) sum_them
        ;;
3|quit|q|Q) exit
        ;;
     *) echo You have not entered a legal option.
        echo Please try again.

        ;;                              screen clears before displaying menu
   esac
   sleep 3
done
```

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.

Answer:

```
#!/usr/bin/sh
# msg_me: display a message to your terminal every 5 seconds
#
term=$(who am i | cut -c12-18)
count=1
while
    [ $count -lt 12 ]
do
    cat msg.file > /dev/$term
    sleep 5
    let count=count+1
done
```

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.

Answer:

```
#!/usr/bin/sh
# ison: Check for a user to log into the system
#       Usage: ison username
#
if [ "$#" -ne 1 ]
then
    echo "usage: $0 user_id" >&2
    exit 99
fi

until who | grep $1 > /dev/null
```

```
do
    sleep 60
done

# When you reach this point, the user has logged in

echo $1 has logged on
who | grep $1
```

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

Answer:

```
#!/usr/bin/sh
# myrm: WasteBasket
#
if [ "$1" = "" ]
   then
   echo "Usage: $0 file [file ...]" >&2
   echo " or: $0 [-l] | [-d]" >&2
   exit 5
fi
opt=$(echo $1 | cut -c1)
if [ "$opt" = "-" ]
then
   case $1 in
     -l) echo;echo "The WasteBasket includes the following files:"
         ls $HOME/.waste;;
     -d) echo;echo "The WasteBasket is being dumped!"
         rm $HOME/.waste/*;;
     -*) echo "$0: $1 invalid argument" >&2
         exit;;
   esac
else
   echo "Are you sure you want to remove $*? [y/n]: \c"
   read ans
   if [ "$ans" = "y" -o "$ans" = "Y" ]
   then
      for i in $*
      do
        if test -f $i
        then
          mv "$i" $HOME/.waste
        else
          echo "$i: Does not exist" >&2
```

```
        fi
      done
   else
      exit
   fi
fi
```

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**.

Overview of Module 19

Audience

general user	General system users

Product Family Type

open sys	Open systems environment

Abstract

The purpose of this module is to teach the basic skills needed for a general user of HP-UX to create tape archives of files. This module does not teach system backups, only "backups" for individual users.

Time

Lab	30 minutes
Lecture	45 minutes

Prerequisites

m46m	Navigating the File System

Instructor Profile

UX	General UNIX knowledge

The instructor of this module must have general UNIX or HP-UX knowledge.

Hardware Requirements

HP9000	Logon access to an HP-UX system
TAPE	Access to tape drive

Software Requirements

UX11	HP-UX release 11.0

Material List

P/N B2355-90033(T) — *HP-UX Reference Manual*, one per terminal

Lab Instructions

setup1 — Create user logon of **user1**, **user2**, ... **user***n*, where *n* is the number of students in the class. Set up one user per student.

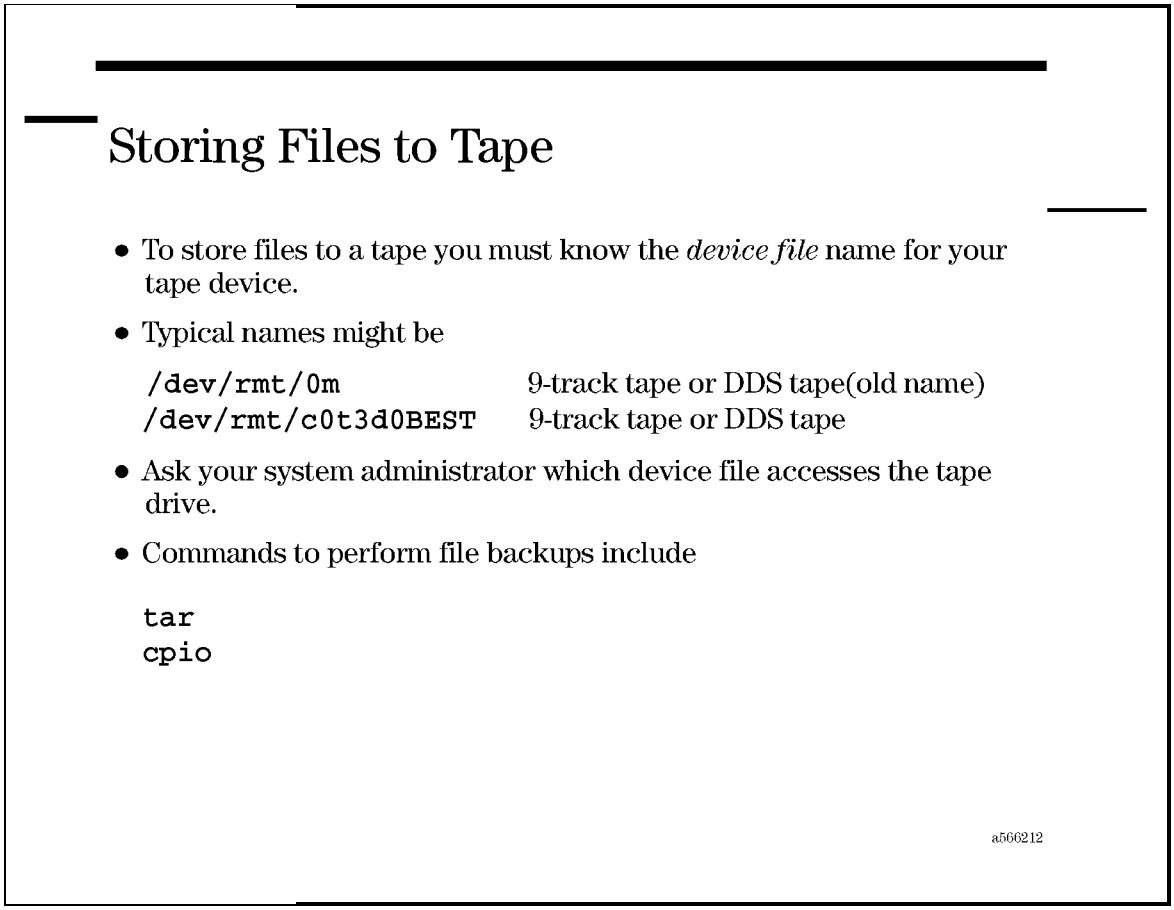
copyfiles — Copy the lab files to the users' home directories.

Lab Files

```
-rw-r--r-- 1 karenk users     34 May 28 16:12 abcdefXlmnop
-rw-r--r-- 1 karenk users     34 May 28 16:12 abcdefYlmnop
./tree:
total 14
drwxr-xr-x 5 karenk users   1024 May 28 16:12 car.models
-rw-r--r-- 1 karenk users     17 May 28 16:12 cherry
-rw-r--r-- 1 karenk users     17 May 28 16:12 collie
drwxr-xr-x 4 karenk users   1024 May 28 16:12 dog.breeds
-rw-r--r-- 1 karenk users     17 May 28 16:12 poodle
-rw-r--r-- 1 karenk users     17 May 28 16:12 probe
-rw-r--r-- 1 karenk users     17 May 28 16:12 taurus

./tree/car.models:
total 6
drwxr-xr-x 2 karenk users     24 May 28 16:12 chrysler
drwxr-xr-x 4 karenk users   1024 May 28 16:12 ford
drwxr-xr-x 2 karenk users     24 May 28 16:12 gm

./tree/car.models/chrysler:
total 0

./tree/car.models/ford:
total 4
drwxr-xr-x 2 karenk users     24 May 28 16:12 sedan
drwxr-xr-x 2 karenk users   1024 May 28 16:12 sports

./tree/car.models/ford/sedan:
total 0

./tree/car.models/ford/sports:
total 2
-rw-r--r-- 1 karenk users     18 May 28 16:12 mustang

./tree/car.models/gm:
total 0
```

```
./tree/dog.breeds:
total 4
drwxr-xr-x 2 karenk users   1024 May 28 16:12 retriever
drwxr-xr-x 2 karenk users     24 May 28 16:12 shepherd

./tree/dog.breeds/retriever:
total 6
-rw-r--r-- 1 karenk users     27 May 28 16:12 golden
-rw-r--r-- 1 karenk users     29 May 28 16:12 labrador
-rw-r--r-- 1 karenk users     26 May 28 16:12 mixed

./tree/dog.breeds/shepherd:
total 0
```

19-1. SLIDE: Storing Files to Tape



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.

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:

Figure 19-9. DDS Format Tape Logo

19-1. SLIDE: Storing Files to Tape — Instructor Notes

Key Points

- Deleted files can only be recovered through a tape backup.
- There are several backup commands available. We will present **`tar`** and **`cpio`**.
- You *must* find out what the designated device file is for your tape drive. You should be able to get this information from your system administrator.

Media Recognition System for DDS Tape

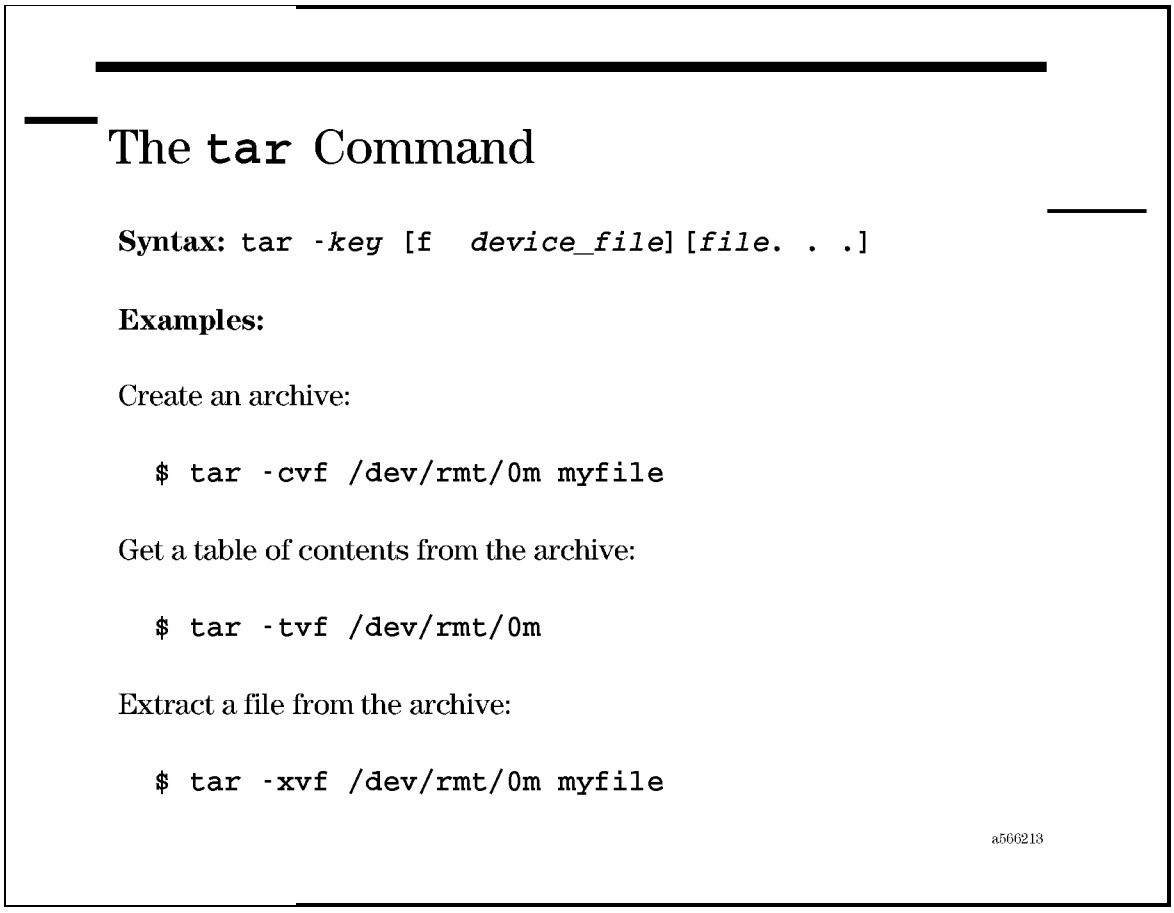
The DDS Manufacturer's Group has devised a system to prevent writing to non-DDS qualified tape. The new tape will have stripes located at the splice between the leader and the tape. Media Recognition System compatible drives detecting these stripes treat all other DAT tapes as write protected. All new cartridges factory shipped from HP have these stripes and will carry the modified DDS logo shown in the student notes.

HP will be ready to implement the Media Recognition System feature in future drives. All existing HP DDS drives will support the new tape as DDS compatible.

Teaching Tips

The objective for the remainder of this module is to provide an introduction to the backup commands so that users can complete simple file backups to tape.

19-2. SLIDE: The `tar` Command



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.

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 to. 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**.

19-2. SLIDE: The `tar` Command — Instructor Notes

Key Points

- The **`tar`** command is the oldest of the UNIX backup commands.
- It is also slow relative to other backup commands.
- Other operating environments can read **`tar`** formatted tape archives.
- The minus sign is not necessary in front of the key but is presented here for continuity.
- Point out that **`tcio`** should be used with cartridge but not with 9-track tape or DDS format DAT tape. Then you must specify the **`f -`** option so that the **`tar`** output is sent to standard output:

```
$ tar cvf - filename myfile | tcio -o /dev/rct/c4t1d0
```

  Likewise when restoring from a cartridge tape **`tar`** archive, you must use **`tcio`** and pipe the output to the **`tar xvf -`** command. In this case **`f -`** directs **`tar`** to read from standard input:

```
$ tcio -i /dev/rct/c4t1d0 | tar xvf -
```

Example:

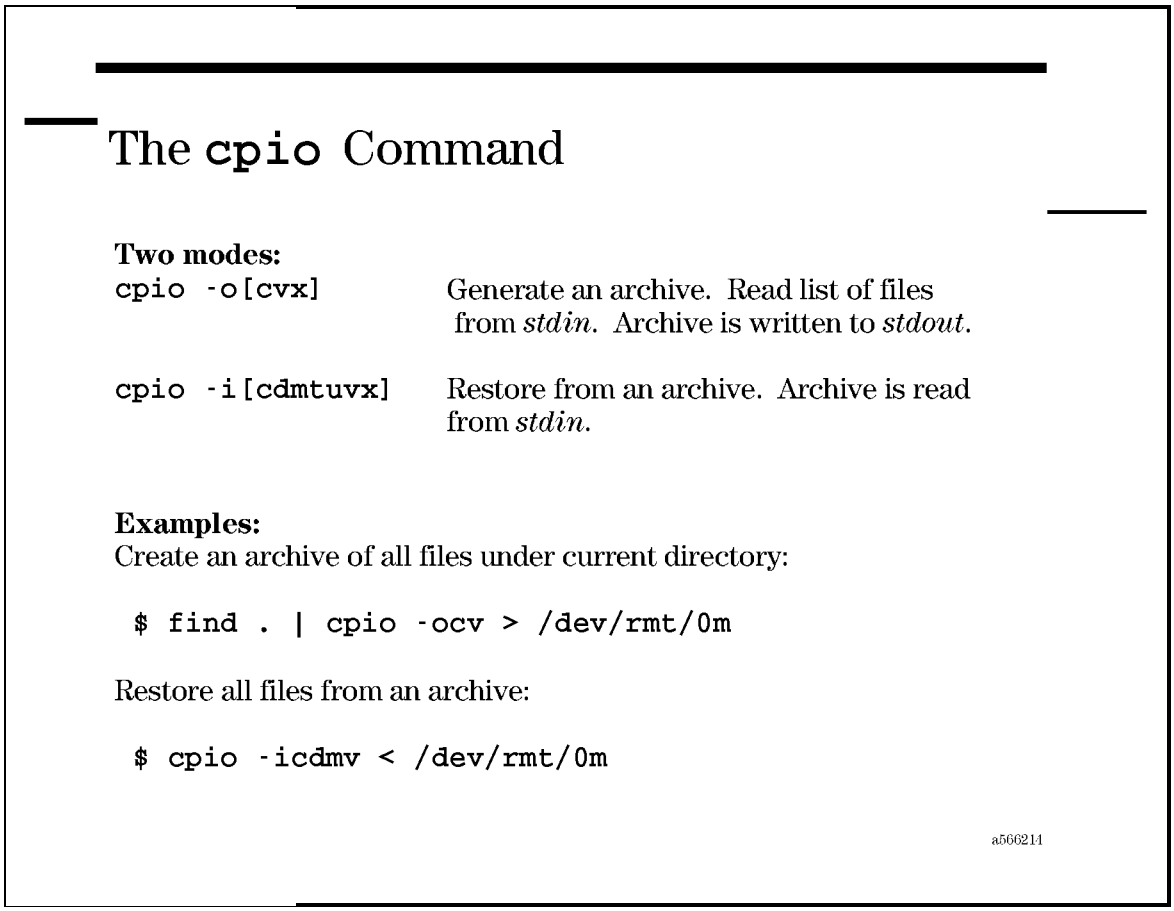
Create a **`tar`** archive on cartridge tape:

```
$ tar -cvf - filename myfile | tcio -o /dev/rct/c4t1d0
```

- You might want to warn your students of the different cartridge tape formats, and that they are *not* interchangeable:

9144 Tape Drive	16-track tapes
9145 Tape Drive	32-track tapes (can *read* 16-track tapes)

19-3. SLIDE: The `cpio` Command



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.

There are several options which 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)**.

19-3. SLIDE: The `cpio` Command — Instructor Notes

Key Points

- **`cpio -o`** creates an archive and sends its **out** to standard output.
- Use **`find`** to generate a list of file names for **`cpio -o`**.
- The output of **`cpio -o`** is commonly redirected to the special file that represents the tape drive.
- **`cpio -i`** reads an archive **in** from standard input.
- The input is commonly redirected from the special file that represents the tape drive.
- Stress the use of the **`-d`** option when using **`cpio -i`** to support re-creation of directories.
- The * must be escaped to avoid premature interpretation by the shell.
- If the **`-m`** option is not specified, the timestamps will be the date and time the files were restored from the tape.
- The **`find`** command can be used to generate a list of file names to backup.
- The **`find`** command can generate a relative list of file names or an absolute list of file names.
- The **`find`** command supports many expressions that can specify file characteristics to find.

WARNING: **Anyone can restore a file from tape, if he or she has access to the tape!**

Using Cartridge Tapes

Using redirection causes excess wear and tear on a cartridge tape drive because the data transfer rates between the host computer and the tape drive are not in sync. The **`tcio`** command is a Hewlett-Packard specific utility that was written to *buffer* the data transfer between **`cpio`** and the cartridge tape drive. Instead of redirecting the output of **`cpio`** straight to the device, the output is piped through **`tcio`** to enable this streaming to take place.

The **`tcio`** command, like **`cpio`**, has two major options:

`-o` Send **out** to a device.

`-i` Read **in** from a device.

The **`-o`** and **`-i`** options to **`tcio`** correspond to those used with **`cpio`**.

Examples

- Create a **cpio** archive, write out to cartridge tape:

```
$ find . | cpio -ocv | tcio -o /dev/rct/c4t1d0
```

- Restore a **cpio** archive, read in from cartridge tape:

```
$ tcio -i /dev/rct/c4t1d0 | cpio -idmcv
```

Teaching Tips

It is helpful if the students type these commands in, but you will see a noticeable degradation in the system if everyone is on the same system.

- Use **find .** to list everything under the current directory (**.**).
- Use **find** *path1 path2 path3* ... to list everything under several directories.
- Use **find** */path1 /path2 /path3* ... to list the *absolute pathnames* associated with all files and directories.
- Use **find /** to list the entire system.

Show examples of sending an archive to a regular file:

```
$ find . | cpio -ocv > dir.archive
```

This is useful when transferring an extensive directory structure through the network. You create the archive on one end, transfer the archive, and unpack the archive at the other end of the network.

The **-p** option of **cpio** is not discussed here. It can be used to copy a directory structure to another directory. The **cp** command supports a recursive copy (**-r**) from one directory to another, but **cpio -p** is generally faster.

```
$ cd
$ mkdir newtree
$ cd tree
$ find . | cpio -pdv $HOME/newtree
```

Remind the class how to use the **find** command. Walk them through the examples. It is sometimes helpful if they type the commands in. It is also useful to do a **man** on the **find** command at this point to show them all the various search criteria that are available, such as searching for names, sizes, and so on.

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 commands syntax by 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**.

19-4. LAB: Offline File Storage — Instructor Notes

Time: 30 minutes

Lab Objective

To practice using the **tar** and **cpio** commands to produce backups of user files.

Notes to the Instructor

It is preferable to have access to a tape drive, but the lab can be performed by writing the commands that would be appropriate to accomplish the tasks or by backing up to a file.

Provide the students with the device file name of the tape drive available on your system. If you are using a Series 800, you might take the whole class to the computer room and let them work on the lab in groups. If adequate tape drives are not available, you should specify which exercises to just write the answers for, and which they should actually create tapes for.

Solutions

1. Using **tar**, create an archive of all files in your *HOME* directory that start with **abc**.

   Answer:

```
$ tar cf /dev/rmt/0m abc*
```

2. Obtain a table of contents listing of this tape archive.

   Answer:

```
$ tar tf /dev/rmt/0m
```

3. Using **find** and **cpio**, make a backup of your whole directory structure from your *HOME* directory on down.

   Answer:

```
$ cd
$ find . | cpio -ocv > /dev/rmt/0m
```

4. Remove the file **backup** from your current directory. Then restore the file from tape using the **cpio** command.

   Answer:

```
$ rm backup
$ cpio -iumc "backup" < /dev/rmt/0m
$ ll backup
```

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**.

Answer:

```
$ mkdir $HOME/tree.cp
$ cd $HOME/tree
$ find . | cpio -pcduv $HOME/tree.cp
```

Appendix A — Commands Quick Reference Guide

Objectives

- To provide a list of frequently used commands along with an explanation of proper use.

Overview of Appendix A

Abstract

This module provides a list of frequently used commands along with an explanation of proper use.

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)
ll	shorthand for **ls -l**

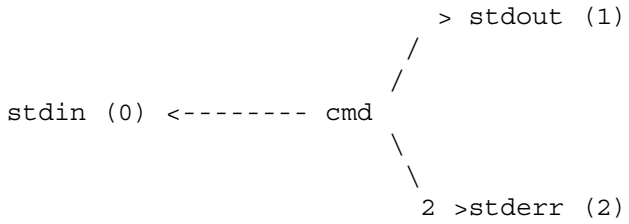
lsf	shorthand for **ls -F**
lsr	shorthand for **ls -R**
lsx	shorthand for **ls -x**
cat [*file*]	display contents of *file*
more [*file*]	display contents of file one screen at a time space — next screen Return — next line **q** — quit **more**
tail - *n file*	display the last *n* lines of a file
pr *file*	format file for printing
lp *file*	queue file to be printed
pr *file* | **lp**	format and print file
lpstat -t	display status of the printer(s) and print system
cancel *jobnumber*	cancel print job
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*
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	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*
umask *mode*	remove *mode* from default permissions
chown *username file*	change ownership of file to *username* refer to **/etc/passwd**
chgrp *groupname file*	change group access of file to *groupname* refer to **/etc/group**
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

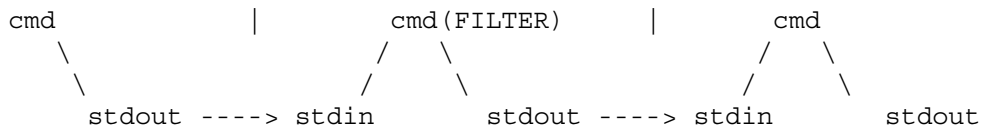
*****	Match zero or more characters
?	Match any single character
[amqp]	Match specific characters, in this case **a, m, q, p**
[a-z]	Match a range of characters, in this case **a** through **z**
[!a-z]	Do NOT match a character in the range

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
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
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 arg4 ... arg9 command line arguments
$0 $1 $2 $3 $4 ... $9            variables for command line args
```

shift *n*	shift through command line arguments
echo $#	display number of command line arguments
echo $*	display all 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
"*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 | lp
```

sort [**-ndt** *X*] [+*field*] [*file*] — **-n** — numeric sort
-d — dictionary sort
-t *X* — use *X* as the delimiter between fields
+*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)

Branching

```
if                          if RETURN VALUE of LAST cmd is true do cmds following then
     cmd(s)                 if RETURN VALUE of LAST cmd is false do cmds following else
then
     cmdtrue(s)
else
     cmdfalse(s)
fi
case $vara in               compare value of vara to patterns
                            execute commands that follow matching pattern
     pat1) cmdsa
```

```
            ;;
      pat2) cmdsb
            ;;
         *) cmds default
            ;;
esac
```

Looping

```
while                    while RETURN VALUE of LAST cmd is true do cmds following do
     cmd(s)
do
     cmdtrue(s)
done
until                    until RETURN VALUE of LAST cmd is true do cmds following do
     cmd(s)
do
     cmdfalse(s)
done
for vara in a b c d e    assign vara to each item in list, do cmds
do
     cmd(s)
done
```

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.
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	A 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.
VISUAL	Defines the edit mode to be used for command stack. Associated with **set -o vi**.

A-1. Commands Quick Reference Guide Instructor Notes

Solutions

2-21. LAB: General Orientation

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

Answer:

You may have had a problem if you made a mistake while typing in your user name or password and tried correcting it with the Backspace key. Remember, the # key is used to erase while logging in.

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

Answer:

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
$ banner
$ banner hello
$ BANNER hello
```

Answer:

```
$ echo                          correct
$ echo hello                    correct
$ echohello                     incorrect
$ echo HELLO WORLD              correct
```

The **echo** command will work with zero or more arguments. As the arguments are just seen as strings of characters, and echoed back to the screen, it does not matter whether they are uppercase or lowercase.

The shell needs white space (spaces or tabs) to separate commands from arguments. The third command line doesn't work because the shell is trying to execute a command called **echohello** instead of executing the **echo** command and passing the argument *hello* to it.

```
$ banner                        incorrect
$ banner HELLO                  correct
```

```
$ BANNER hello                          incorrect
```

The **banner** command requires at least one argument, unlike the **echo** command. Therefore, the second entry is legal, because **banner** does not care if the string(s) to be echoed are uppercase or lowercase. In the third instance the shell will look for a command called **BANNER**, which is not a legal shell command. Remember, the shell is case sensitive, and therefore **banner banner** is not the same as **BANNER**.

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

Answer:

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

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?

Answer:

```
$ who
$ who am i
$ whoami
```

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

Answer:

```
$ mesg
```

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?

Answer:

```
$ who                                   Confirm that your partner is logged in.
$ write partner
message contents
message contents
```

Ctrl + d *Conclude conversation.*

If your partner is not logged on, you will get the message:

```
partner is not logged on.
```

If your partner has disabled messaging on his or her terminal, you will get the message :

```
Permission denied.
```

8. Read the system's news. What command did you use? Can you display the news *after* you have read a message?

Answer:

```
$ news
this is a news message
$ news
no new news
$ news -a
this is a news message
```

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)**.

Answer:

```
$ date +%m-%d-%y
```

10. Using the *UNIX Reference Manual*, find the **cp** command. What is its function? What is the minimum number of arguments that it requires?

Answer:

The **cp** command is used to copy one or more files. It requires at least two arguments: a source file name and a destination file name.

11. Using the *HP-UX Reference Manual*, find the **ls** command. What is its function? What is the minimum number of arguments that it requires?

Answer:

The **ls** command is used to display file names. It requires no arguments. Notice it has many options available. Each option will extend the capability of the **ls** command, and each option is identified as a single letter.

12. Issue the command **ll /usr/bin**. You will see several screens worth of data scroll by. 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 **ll** command?

Answer:

ll /usr/bin should generate several screens worth of output. Issuing the **clear** command after moving the cursor to the top of the current screen will clear only the last screen of output. Scrolling down will display the previous screens.

13. Log out of your terminal session. Log back in with the CAPS lock on. How can this situation be corrected without logging off and then back in again. (Hint: Look at the manual page for the **stty** command.)

Answer:

Notice that if you hit the Caps Lock key, it has no effect. You must use the **stty** command to disable the Caps lock:

```
$ STTY -LCASE
```

then hit the Caps Lock key on your keyboard. You will now be able to enter uppercase and lowercase letters. This interface is provided for terminals that support only uppercase input, so that they can interpret the commands properly that are normally defined as all lowercase.

3-19. LAB: Using CDE

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

Answer:

1. Position the mouse cursor over the directory **class** and double click.
2. Position the cursor over the file **cde_intro**.
3. Choose **Copy To** from the **Selected** menu. A pop-up dialog box will appear prompting you for the file. Type in **cde_intro2** .
4. Press OK.

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

Answer:

1. Position the cursor over the file **cde_intro2** .
2. Choose **Move To** from the **Selected** menu. A pop-up dialog box will appear prompting you for the destination folder. Type in **cde_dir**.
3. Press OK.

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

Answer:

1. Position the cursor over the folder **cde_dir** and double click to change to that directory.
2. Highlight the file **cde_intro2**.
3. A pop-up dialog menu will appear with the current ownership and permission information, including the size and last modified information.
4. Click *off* the write permission.
5. Press OK.

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**.

Answer:

1. Click on **File** menu and choose **Find...**.
2. Type *graphical environment* in the **File Contents** field.
3. Type **class** following your home directory in **Search Folder:** field.
4. Click Start

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

Answer:

1. Click on **File** menu and choose **Find...**.
2. Type **data*** in the **File or Folder Name:** field.
3. Type **class** following your home directory in **Search Folder:** field.
4. Click Start

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.

Answer:

1. Click on **File** menu and choose **Find...**.
2. Type **data?** in the **File or Folder Name:** field.
3. Type **class** following your home directory in **Search Folder:** field.
4. Click Start

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

   Answer:

   1. Open the **cde_dir** folder.
   2. Highlight the file **cde_intro2**.
   3. Either choose the **Put in Trash** menu item from the **Selected** menu *OR* click and hold mouse button 1 while dragging the icon down to the Trash Can. Once the file icon is over the trash can release the mouse button.

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

   Answer:

   1. Double click on the Trash Can icon in the Front Panel to open Trash Can window.
   2. Select the file **cde_intro2** to restore.
   3. Click on **File** to open the menu bar, then select **Put Back**. The file will return to its original location.

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

   Answer:

   1. Open the **cde_dir** folder.
   2. Highlight the file **cde_intro2**.
   3. Either choose the **Put in Trash** menu item from the **Selected** menu *OR* click and hold mouse button 1 while dragging the icon down to the Trash Can. Once the file icon is over the trash can release the mouse button.
   4. Double click on the Trash Can icon in the Front Panel to open Trash Can window.
   5. Select the **cde_intro2** file to restore.
   6. Click on **File** to open the menu bar, then select **Shred**. The file will be permanently removed.

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.

    Answer:

    1. Position the mouse cursor to the beginning of the first paragraph. Press and hold mouse button 1 while dragging the cursor across the area to be copied. Release the button.
    2. Choose **Copy** from the Edit menu. A copy of the tet is stored on a clipboard.
    3. Position the cursor to the end of the file.

4. Choose **Paste** from the Edit menu.

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

Answer:

1. Choose **Find/Change ...** from the Edit menu.
2. Type **CDE** in the **Find** field.
3. Type **Common Desktop Environment** in the **Change** field.
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.

12. Correct all misspelled words in the file.

Answer:

1. Choose **Check Spelling** from the **Edit** menu. The Spell dialog box will be displayed.
2. Type the correct work 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.

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

Answer:

1. Position the mouse cursor on a portion of the workspace switch that is not occupied by other controls or workspace buttons and press mouse button 3. Choose **Add Workspace**
2. Position the mouse cursor on the new workspace called **New** and press mouse button 3. Choose **Rename**.

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

Answer:

1. Click on the up arrow above the Text Editor control on the Front Panel.
2. Position the cursor next to the Terminal icon and press mouse button 3.
3. Select **Copy to Main Panel**.

4. Click on the down arrow above the Terminal control on the Front Panel to close the Personal Applications subpanel menu.

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

Answer:

1. Open the Application Manager.
2. Double-click the Desktop_Apps group icon to display its contents.
3. Scroll down until you see the **Man Page Viewer** action icon.
4. Double-click the action icon to execute the application.
5. Type in **ls** in the dialog window to execute the action.

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

Answer:

1. Choose **New Message** from the Compose Menu
2. Enter your mail partner's email address in the **To** field and the subject in the **Subject** field.
3. Once your have addressed the message, press Return to go to the text area and compose the message.
4. Click the Send button.

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

Answer:

To send the reply:

1. Select the message for reply.
2. From the Compose menu choose **Reply to Sender**
3. Enter reply.
4. Click Send.

To forward the message:

1. Select the message to forward.
2. Choose **Forward Message** from the Compose menu.
3. Enter the mail address for the recipients in the To: field.
4. Include your own comments if desired.
5. Click Send.

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.

Answer:

Once you have created the file using Text Editor, do the following to create the template:

1. In the Mailer, click **Mail Options** from the **Mailbox** menubar.
2. Click the **Category** button and choose **Templates**. The template dialog box will appear.
3. Type the name **monthly** in **Menu Label** field.
4. Enter the name of the file **status** in the **File/Path:** field.
5. Click Add to include the template in the list of templates.

To use the template:

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

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.

Answer:

1. Open the calendar to display the current month. Click on 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.
5. If you want this to occur on a regular basis, click More .

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.

Answer:

1. Choose **Menu Editor** from the Browse Menu.
2. Type **calendar-name@hostname** in the **User Name** field.
3. Click Add Name.
4. Click OK.
5. Repeat for your calendar and your mail partner's.

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.

Answer:

1. Select **Compare Calendars** from the Browse menu.
2. Select the name of the calendars you want to view.

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

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.

Answer:

1. Choose **Options...** from the File menu.
2. From the Category menu, choose **Access List** to display the Access List and Permissions dialog box.
3. In the **User Name** field, type **calendar-name@hostname** for the calendar to which you want to grant access.
4. Select View, Insert, and Change permissions.
5. Click Add to add the calendar to the Access List with the permissions you've chosen.
6. Click Apply.
7. Repeat for yourself so that you can overlay your calendar with your mail partner's.

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

Answer:

1. Browse your menu together with your mail partner's menu

2. Click on an unshaded area available to both your calendars

3. Click Schedule. The Calendar Group Appointment Editor will be displayed. A **Y** in the Access column means that you have insert access to update their schedule. An **N** means that you do not. If you do not have insert access, remind your mail partner to grant you access.

4-14. LAB: The File System

1. What is the name of your *HOME* directory?

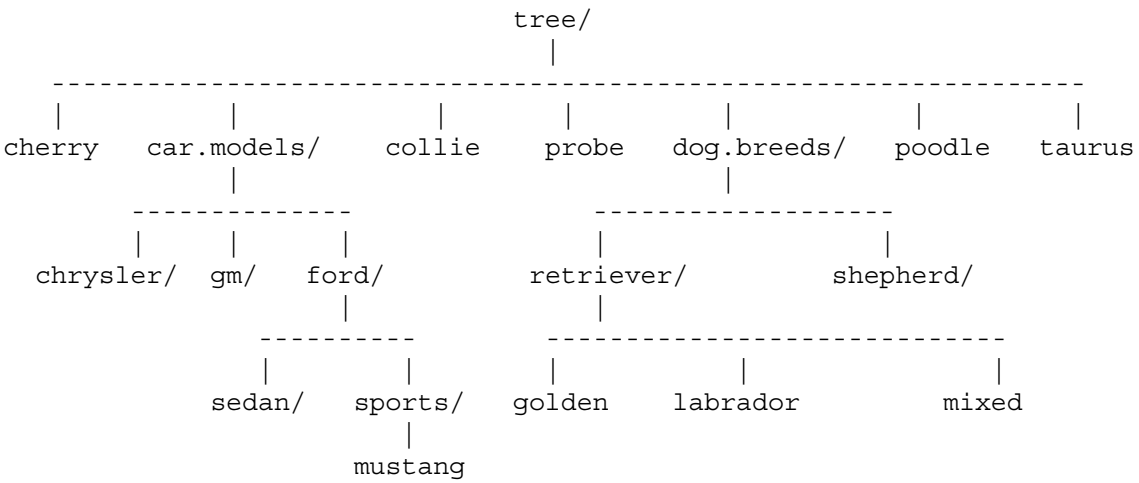
Answer:

Login and then issue the **pwd** command. It should display something similar to **/home/*YOUR_USER_NAME*** .

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.

Answer:

The exercise consists of a lot of **ls** (**lsf**) commands. Or, as an alternative, you could have used the **-R** (recursive) option. The directory map should look like



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?

Answer:

Full path name **/home/ *YOUR_USER_NAME* /tree/dog.breeds/retriever/ labrador**

Relative path name **tree/dog.breeds/retriever/labrador**

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?

Answer:

```
$ cd
$ cd tree/dog.breeds/retriever
$ cd ../shepherd
$ cd ../../car.models
$ cd
```

To verify each destination

```
$ pwd
```

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?

Answer:

```
$ cd
$ mkdir junk
$ cd junk
$ pwd
/home/YOUR_USER_NAME
/junk
```

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

Answer:

```
$ mkdir junk/dirA junk/dirA/dir1 junk/dirA/dir2 junk/dirA/dir1/dirc
```

or

```
$ mkdir -p junk/dirA/dir1/dirc junk/dirA/dir2
```

If you entered the directory names in the order in which they are presented in the exercise, it will fail, because the command executes the arguments from left to right.

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?

Answer:

```
$ ls junk/dirA
$ ls /home/YOUR_USER_NAME/junk/dirA
```

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

Answer:

```
$ rmdir junk/dirA/dir1/dirc
$ rmdir junk/dirA/dir1
$ rmdir junk/dirA/dir2
$ rmdir junk/dirA

$ rmdir junk/dirA/dir1/dirc junk/dirA/dir1 junk/dirA/dir2 junk/dirA
```

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.

Answer:

```
$ cd
$ pwd
/home/YOUR_USER_NAME
$ ls -l /usr/bin/cp /usr/bin/vi            Absolute path names
$ ls -l ../../usr/bin/cp ../../usr/bin/vi
                                           Relative path names
```

5-16. LAB: File and Directory 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?

Answer:

```
$ cat funfile
```

The file is too long for one screen. The **more** command provides screen scrolling control. For example:

```
$ more funfile
```

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?

Answer:

```
$ more tree
****** tree is a directory ******
```

more knows that **tree** is a special directory file, not a normal text file, so its contents cannot be displayed to the screen in a readable format. You use the **ls** command to display the contents of a directory. For example:

```
$ ls tree
```

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?

Answer:

```
$ more /usr/bin/ls
****** /usr/bin/ls: Not a text file ******
```

more knows that **/usr/bin/ls** is a compiled program, not a normal text file, so its contents cannot be displayed to the screen in a readable format.

```
$ cat /usr/bin/ls
```

This command produces what appears to be garbage. In fact, this is what happens when you use the **cat** command to display a binary (compiled) program. Your terminal settings may have been changed by this. To reset your HP terminal:

- Hit the Break key.
- Simultaneously press Shift + Ctrl + Reset.
- Press Return to get the shell prompt.
- At the prompt, type the commands:

```
$ tset -e -k                          -e: sets erase to ^H, -k: sets kill to ^X
$ tabs
```

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.

Answer:

```
$ cp names names.cp
$ cat names names.cp
```

5. If the file **names** is modified, will this 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**?

Answer:

The files **names** and **names.cp** are individual entities. The content of **names** was overwritten with the content of the file **funfile**. The file **names.cp** is not affected.

```
$ cp funfile names
$ more names names.cp
```

names now contains the same contents as **funfile**, while **names.cp** still contains the content that was in **names**.

6. How do you restore the file **names** ? Issue the command to restore **names**.

Answer:

To restore the contents of the file **names**, copy or move from the file **names.cp**.

```
$ cp names.cp names
```

or

```
$ mv names.cp names
```

7. Make another copy of the file **names** called **names.new**. Change the name of **names.new** to **names.orig**.

Answer:

```
$ cp names names.new
$ mv names.new names.orig
```

8. How do you create two files (called **names.2nd** and **names.3rd**) that reference the contents of the file **names**?

Answer:

```
$ ln names names.2nd
$ ln names names.3rd      or     $ln names.2nd names.3rd
```

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**?

Answer:

The files **names**, **names.2nd**, and **names.3rd** are all referencing the same data on the disk. If one is modified, all three will be modified. From the long listing, you see that their link count has gone up to three, since there are now three names referencing the same data. **names.orig** is still an individual entity, as seen by its link count still being one.

```
$ cp funfile names
$ ls -l names.orig names names.2nd names.3rd
-rw-r--r-- 1 user3 class 37   Jul 24 11:06 names.orig
-rw-r--r-- 3 user3 class 125  Jul 24 11:08 names
-rw-r--r-- 3 user3 class 125  Jul 24 11:10 names.2nd
-rw-r--r-- 3 user3 class 125  Jul 24 11:12 names.3rd
```

If you do an **ls -i** of the **names** files, their **inode** numbers will be displayed. The inode stores each file's characteristics, such as permissions, number of links, and ownership. Files that are linked together share the same inode.

```
$ ls -i names.orig names names.2nd names.3rd
102 names.orig
322 names
322 names.2nd
322 names.3rd
```

10. Remove the file **names**. What happens to **names.2nd** and **names.3rd**?

Answer:

```
$ rm names
```

The files **names.2nd** and **names.3rd** are unaffected except that their link count will be reduced by one, which can be seen with the **ls -l** command:

```
$ ls -l names.orig names names.2nd names.3rd
names not found
-rw-r--r-- 1 user3 class 37   Jul 24 11:06 names.orig
-rw-r--r-- 2 user3 class 125  Jul 24 11:10 names.2nd
-rw-r--r-- 2 user3 class 125  Jul 24 11:12 names.3rd
```

11. Use the interactive option for **rm** to remove **names.2nd** and **names.3rd**.

Answer:

```
$ rm -i names.2nd names.3rd
names.2nd? y
names.3rd? y
$
```

12. Copy the file **names.orig** back to **names**.

Answer:

```
$ cp names.orig names
```

---

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

Answer:

```
$ cd
$ mkdir fruit
$ mv lime grape orange fruit
```

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

Answer:

```
$ cd
$ mv apple fruit/APPLE
$ mv peach fruit/Peach
$
```

3. Look at the **tree** directory structure in your *HOME* directory. It requires a little organization.

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 could make these changes from any directory, but what directory do you think you should be in?

Answer:

```
$ cd
$ cd tree
```

```
$ pwd
/home/YOUR_USER_NAME/tree
$ mv collie poodle dog.breeds
$ mv probe car.models/ford/sports
$ mv taurus car.models/ford/sedan
$ mkdir horses
$ cp car.models/ford/sports/mustang horses
$ mv cherry ../fruit
```

4. Move the **fruit** directory from your *HOME* directory to the **tree** directory.

Answer:

```
$ cd
$ mv fruit tree
```

A directory called **fruit** is created under **tree**.

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.

Answer:

```
$ cd
$ cd tree/fruit
$ mv ../../banana ../../lemon .
```

---

1. Under your *HOME* directory, you will find a directory **scavenger** and a file **scaveng.README** providing the first clue for a scavenger hunt. 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!

Answer:

```
$ cd
$ cd scavenger
$ more scaveng.README
north, 1 mile
$ cd north/1_mile
$ more README
east, 2 miles
$ cd ../../east/2_mile
$ more README
You are on the right track!
south, 3 miles
$ cd ../../south/3_mile
```

```
$ more README
You have to keep going
south, 2 miles
$ cd ../2_mile
$ more README
You are almost there
west 1 mile
$ cd ../../west/1_mile
$ more README
CONGRATS
You have found the end of the trail.
The code word is ________
```

---

1. List the current status of the printers in the **lp** spooler system and find the name of the default printer.

Answer:

```
$ lpstat -t
scheduler is running
system default destination: rw
device for rw: /dev/rw
rw accepting requests since Jul 1 10:56:20 1994
printer rw is idle. enabled since Jul 4 14:32:52 1994
fence priority : 0
```

2. Send the file named **funfile** to the line printer. Make a note of the request ID that is displayed on your terminal.

Answer:

```
$ lp funfile
request id is rw-58 (1 file)
```

3. Verify that your requests are queued to be printed.

Answer:

```
$ lpstat
rw-58 ralph   3967  Jul 4 16:57:25 1994
rw-59 ralph   1331  Jul 6 13:01:19 1994
```

4. How can you tell what files other users are printing? Try it.

Answer:

You can tell by using **lpstat -t**.

5. Use the **cancel** command to remove your requests from the line printer system queue. Confirm that they were canceled.

Answer:

```
$ cancel rw-58 rw-59
request "rw-58" canceled
request "rw-59" canceled
$ lpstat
$
```

6-14. LAB: File Permissions and Access

1. Look under your *HOME* directory for a file called **mod5.1**. Who has what access to this file? Can you display the contents of **mod5.1**?

Answer:

```
$ ls -l
-rw-r--r-- 1 YOUR_LOGNAME class     20 Jan 24 13:13 mod5.1
```

YOUR_LOGNAME has read and write access.
Members of group *class* have read access.
All other users have read.

```
$ cat mod5.1
```

This is successful since you have read permission.

2. Modify the permissions on **mod5.1** so that they are: **-w-------**. Can you display the contents of **mod5.1**?

Answer:

```
$ chmod a-rwx,u=w mod5.1
$ cat mod5.1
```

You no longer have read access to the file **mod5.1**, so the **cat** will fail.

3. Modify the permissions on **mod5.1** so that they are: **rw-------**. Can you display the contents of **mod5.1**? Can your partner display the contents of your **mod5.1**?

Answer:

```
$ chmod u=rw mod5.1
```

You can display the contents of **mod5.1**.
Your partner cannot display the contents of **mod5.1**.

4. How can you modify the permissions on **mod5.1** so that your partner *can* read the file?

Answer:

```
$ chmod g+r mod5.1
```

The file allows read access to all members of the group *class*, and your partner is also a member of the group class. Therefore, you provide the group read access to the file.

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?

Answer:

```
$ cp mod5.1 mod5.2
$ chmod -w mod5.2
$ rm mod5.2
mod5.2: 444 mode ? (y/n)
```

mod5.2 is removed!
You would have to remove the write permissions from your *HOME* directory as well.
If you remove write permissions from your *HOME* directory and then try to remove the file, you will get a message "permission denied".

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?

Answer:

The owner is root. The group is other. Only the super-user can change the ownership or group. You have read access only.

7. If you wanted to make changes to the file **root_file**, how would you go about it?

Answer:

Since you have read permission, you can make a copy of **root_file** . You will own the copy, and can therefore, modify the copy's contents.

```
$ cp root_file my_root_file
$ ls -l my_root_file
-rw-r--r-- 1 user3 class   3967 Jan 24 13:13 my_root_file
```

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?

Answer:

```
$ who am i
user3 tty03 Jul 9 11:10
$ mesg y
$ ll /dev/tty03
crw--w--w- 1 user3 class      0 Jan 24 13:13 /dev/tty03
```

```
$ mesg n
crw------- 1 user3 class      0 Jan 24 13:13 /dev/tty03
```

You own the device file associated with your terminal connection. The **mesg** command is essentially running a **chmod** on your terminal device file to grant or deny write access of others to your terminal.

---

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**)

Answer:

```
$ cd
$ mkdir mod5.dir
$ cp mod5.1 mod5.dir
$ ls mod5.dir
mod5.1
$ ls -ld mod5.dir
drwxrwxrwx 3 YOUR_LOGNAME class   1024 Jul 24 13:13 mod5.dir
$
```

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**?

Answer:

```
$ chmod a-rwx,u+rw mod5.dir
$ cd mod5.dir
sh: mod5.dir: Permission denied.
$ ls mod5.dir
mod5.1
$ ls -l mod5.dir/
mod5.dir/mod5.1 not found
total 0
$ cat mod5.dir/mod5.1
cat: cannot open mod5.dir/mod5.1: Permission denied
$
```

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**?

Answer:

```
$ chmod u+wx mod5.dir
$ ls mod5.dir
mod5.dir unreadable
```

```
$ cat mod5.dir/mod5.1
This is the contents of mod5.1
$ cd mod5.dir                          cd is successful
$ pwd
/home/user3/mod5.dir
$ ls
. unreadable
```

4. Can other users copy files into your *HOME* directory? How do you display the permissions for your *HOME* directory?

Answer:

```
$ cd
$ ls -ld .
drwxr-xr-x 3 YOUR_USER_NAME class   1024 Jul 24 13:13 .
```

Other users can display the contents of your *HOME* directory, and change to your HOME directory, but they cannot modify the contents of your *HOME* directory. Therefore, other users cannot copy files to 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** ?

Answer:

```
$ ls -ld /usr/bin
dr-xr-xr-x 3 bin other    1024 Jul 24 13:13 /usr/bin
```

Write access for others is not set on **/usr/bin**, so your copy should fail.

6. Can you copy the file **/usr/bin/date** to your *HOME* directory?

Answer:

```
$ cd
$ ls -l /usr/bin/date
-r-xr-xr-x 1 bin bin   16384 Nov 15 13:13 /usr/bin/date
$ cp /usr/bin/date .
```

Since **/usr/bin/date** has read permission for others, you are able to make a copy of the file.

---

1. Look under your *HOME* directory, you should find 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?

Answer:

```
$ ls -l YOUR_LOGNAME
-rw------- 1 YOUR_LOGNAME class   3967 Jan 24 13:13 YOUR_LOGNAME
```

You have read and write access.
Your partner is also in the group *class*.
Your partner has no access 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?

Answer:

```
$ ls -l YOUR_LOGNAME
-rw------- 1 YOUR_LOGNAME class   3967 Jan 24 13:13 YOUR_LOGNAME
You initially have read and write access.
$ chown partner_login_name YOUR_LOGNAME
$ ls -l YOUR_LOGNAME
-rw------- 1 partner class 3967 Jan 24 13:13 YOUR_LOGNAME
```

- You no longer have access to this file.
- You need a minimum of read access to copy a file, so you cannot make a copy with the current permissions.
- You are a member of the group *class*, but the group permissions are disabled.

The only way you can get ownership back, is to have your partner **chown** the file back to you. (Have you been nice to your partner?) You could also **su** to your partner's account (if he or she will share his or her password) and **chown** the file yourself.

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?

Answer:

```
$ cp mod5.1 mod5.3
$ chmod a-rwx mod5.3
$ chown partner mod5.3
```

You can change the ownership because the permissions are associated with your access to the contents of the file, *not* the ownership and group identifiers assigned to the file.

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**?

Answer:

```
$ cp mod5.1 mod5.4
$ chmod a-rwx,u+w,ug+r mod5.4
```

```
$ chgrp class2 mod5.4
$ chown root mod5.4
$ cat mod5.4cat: cannot open mod5.4: Permission denied
```

Since you are not currently a member of the group **class2**, you cannot access the file mod5.4.

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.)

Answer:

```
$ newgrp class2
$ cat mod5.4
This is the contents of mod5.1
$ newgrp
```

Once you change your effective group to class2, you can then access the file mod5.4.

---

1. What are the permissions when you create a new file? Hint: Create a new file by using the editor, and copy or **touch** an existing file. Examine the permissions on the new files. How about a new directory? What is your current file creation mask?

Answer:

```
$ touch new_file
$ ls -l new_file
-rw-rw-rw- 1 YOUR_USER_NAME class     0    Jul 24 13:13 new_file
$ mkdir new_dir
$ ls -ld new_dir
drw-r---r-- 3 YOUR_USER_NAME class  1024 Jul 24 13:13 new_dir
$ umask
000
```

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.

Answer:

```
$ umask a-rwx,u=rw,g=r,o=r
$ touch new_file2
$ ls -l new_file2
-rw-r--r-- 1 YOUR_USER_NAME  class    0    Jul 24 13:13 new_file2
$ mkdir new_dir2
```

```
$ ls -ld new_dir2
drw-r--r-- 3 YOUR_USER_NAME  class 1024 Jul 24 13:13 new_dir2
```

7-21. LAB: Exercises

1. Create an alias called **h** that executes the **history** command.

   Answer:

   ```
   $ alias h=history
   ```

2. Check the commands in the **.shrc** file in your home directory. Add your **h** alias to the list.

   Answer:

   ```
   vi .kshrc
   ```

   add the line

   ```
   alias h=history
   ```

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 when separated with a semicolon.)

   Answer:

   ```
   $ alias go="cd /home/user3/tree; ls -F"
   $ go
   car.models/ dog.breeds/ fruit/ horses/
   $ pwd
   /home/user5/tree
   $ cd
   ```

4. Log out and then log back in to test your aliases. Why did you have to log out?

   Answer:

   You had to reread the **.profile** and **.kshrc** files. The easiest way is to log out and then log back in.

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**?

   Answer:

   Typing the command line given puts **more f** on the command line, and the shell beeps because there is more than one file starting with **f**. If you type an **u** and then Esc Esc again, the file name **funfile** will be completed for you.

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.

Answer:

```
$ cp fr[ESC] [ESC] tree/ca [ESC] [ESC]ford/sports
$ cp frankenstein tree/car.models/ford/sports
```

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).

Answer:

```
$ cd /user/spol/ko/interface[Esc]
```

Using [Backspace] and the space bar to position the cursor, use **vi** commands **x**, **a**, **cw** to make the appropriate changes. Remember to use [Esc] whenever you need to leave input mode.

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.

Answer:

```
$ ls -F
$ [Esc] [k]
```

Now use the **r** command to change **ls -F** to **ls -l** and press [Return].

9. Using the command stack, recall the previous copy command, and change **frankenstein** to **funfile**.

Answer:

```
[ESC] [k]
$ cp frankenstein tree/car.models/ford/sports
[l] [l] [l]  or  [w]
 [c] [w] funfile [Return]
$ cp funfile tree/car.models/ford/sports/
```

10. Recall the previous copy command, and modify it so that you display the contents of the sports directory.

Answer:

```
ESC k
$ cp funfile tree/car.models/ford/sports
c w ls ESC                              change word cp to ls
w
d w                                     to delete funfile
Return
$ ls tree/car.models/ford/sports
```

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.

Answer:

```
ESC k
$ ls tree/car.models/ford/sports
c w cd ESC                              change word ls to cd
w w                                     repeat until cursor is under sports
c w sedan ESC                           change word ls to cd
Return
$ cd tree/car.models/ford/sedan
$ pwd
tree/car.models/ford/sedan
```

12. Change back to your *HOME* directory, and 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.

Answer:

```
$ cd
$ history
$ r 'cd t'    or    r cmd_number
$ r p
```

8-12. LAB: The Shell Environment

1. Using command substitution, assign today's date to the variable *today*.

Answer:

```
$ date
Fri Apr 2 11:57:21 EST 1993
$ today=$(date)
echo $today
Fri Apr 2 11:57:21 EST 1993
```

2. What is an easy way to list the contents of another user's home directory?

Answer:

If the other user's name was *mike*, you could get a listing of his home directory using:

```
$ ls ~mike
```

3. Set a shell variable named *MYNAME* equal to your first name. How do you see the contents of that variable?

Answer:

```
$ MYNAME=user3
$ echo $MYNAME
user3
```

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*?

Answer:

The *MYNAME* variable was set in the parent shell's local data area. When the child shell was spawned, it inherited only the parent's environment variables.

When the child shell is dead, the parent wakes up and remembers all that it knew. You can test this by typing

```
$ echo $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?

Answer:

```
$ export MYNAME

$ env
```

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?

Answer:

```
$ MYNAME=user2

$ env
```

MYNAME is still an environment variable in the child shell.

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?

Answer:

```
$ unset MYNAME
```

MYNAME will no longer exist in the child shell because the **unset** command removes it.

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?

Answer:

$ Ctrl + c

Return

The removal of the variable in the child shell does not have an effect on the variable in the parent shell. Therefore, *MYNAME* still exists in the environment of the parent shell. To verify this, you can display the environment variables in the parent shell.

```
$ env
```

9. Modify your shell prompt so that it displays: *good_day$*. What happens to your prompt when you log out and log back in?

Answer:

```
$ PS1=good_day$
good_day$
```

When you log out and log back in the prompt reverts to the default.

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?)

Answer:

```
$ PS1=$LOGNAME    or    $ PS1=$(whoami)
user3                   user3
```

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 arguments?

Answer:

```
$ dir=/usr/bin/ls
$ $dir
```

$dir will be substituted with **/usr/bin/ls** , and then execute the command **/usr/bin/ls** . Yes this will accept command line arguments. Try by executing:

```
$ $dir $HOME /tmp /var
```

9-8. LAB: File Name Generation

1. Change to your *HOME* directory, then type the command **ls *** and explain the output.

Answer:

Remember that the **ls** command can act two different ways. If given a file name as an argument, **ls** displays the file's name. If given a directory name, **ls** lists the contents of that directory. Thus when given a list of file and directory names, such as that generated by the asterisk (*), **ls** will list the names of all files under the current directory and list the contents of all immediate subdirectories.

2. If the command **echo ???XX** produces the output **???XX**, what does it mean?

Answer:

There are no files in the current working directory that match the pattern **???XX**. Therefore the file name generation characters are taken literally, and the echo command echoes them out.

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**.

Answer:

a. **ls *.c**
libtest.c math.c mod1.c mod2.c mod3.c myprog.c part1.c part2.c xdbtest.c

b. **ls mod*.c** or **ls mod[0-9].c** or **ls mod?.c**
mod1.c mod2.c mod3.c

c. **ls *file***
file.1 file.2 herfile myfile my_root_file ourfile root_file yourfile filegen/: Abc Abcd abc abcdemf e35f efg fe3f fe3fg

d. **ls *.[cfp]**
libtest.c math.c math.f math.p mod1.c mod2.c mod3.c myprog.c myprog.f myprog.p xdbtest.c part1.c part2.c

4. Create a directory called **c_source**. Move all of your **.c** files to the **c_source** directory using the file name generating characters.

Answer:

```
$ mkdir c_source
$ mv *.c c_source
```

5. Create a directory called **dir_1** under your *HOME* directory. What happens when you issue the command: **cd dir*** ?

Answer:

```
$ cd dir*                          Expands to: cd dir_1
```

You will effectively change to the *dir_1* directory.

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.

Answer:

```
$ cd                               Expands to: cd dir_1 dir_2 dir_3 dir_4
$ mkdir dir_2 dir_3 dir_4

$ cd dir*
```

This is not a legal usage for the **cd** command and will fail.

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.

Answer:

```
$ touch AXX
$ touch A.XX
$ touch B.XX
$ touch XX.AB
$ touch XX.A
$ echo ?XX
AXX
$ echo ?.XX
A.XX B.XX
$ echo *XX
A.XX AXX B.XX
```

```
$ echo XX.??
XX.AB
$ echo XX.*
XX.A XX.AB
```

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.)

Answer:

```
$ rm *XX*
```

10-7. LAB: Quoting

1. Type an echo command that will produce the following output:

```
$1 million dollars   ...   and that's a bargain !
```

Answer:

```
$ echo "\$1 million dollars   ...   and that's a bargain !"
$ echo '$1 million dollars   ...   and that'\''s a bargain !
```

There are several options to echo this string out, since the echo command uses blank spaces as delimiters between the strings. You really only need to escape the **$** and the **'**.

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.

Answer:

```
$ long_string="\$1 million dollars   ...   and that's a bargain !"
$ echo "$long_string"
```

This has fewer options because you must quote all of the blank spaces for the variable assignment to succeed. Note what happens when you don't quote **$long_string**.

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?

Answer:

```
$ banner good day                    2 command line arguments
```

```
GOOD
DAY
$ banner "good day"                    1 command line argument
GOOD DAY
```

The quotation marks escape the space as an argument delimiter, so the second command will see only one command line argument.

4. Assign to your prompt the string: *Way to go YOUR_USER_NAME $*

Answer:

```
$ PS1="Way to go $LOGNAME $ "
Way to go user3 $
```

5. How would you display the following message?

```
Exercises #1, #2, and #3 are now complete.
```

Answer:

```
echo Exercises \#1, \#2, and \#3 are now complete.
```

The **#** symbol precedes comments, and must therefore be escaped.

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"
```

Answer:

```
$ echo '$abc'
$abc
```

The single quotes (**'**) do *not* allow variable substitution to occur, so the literal string *$abc* will be echoed back to your terminal.

```
$ echo "$abc"

sh: abc: parameter not set.
```

When the **$** appears within the double quotes (**"**), the shell will try to de-reference the variable. Since the variable does not hold a value, the shell will generate an error message when it tries to evaluate the variable value.

Note: The POSIX/Korn Shell will generate error messages when referencing a variable that has not been defined and the **set -u** option is set. You can enter **set -o** to view the options configured for your shell. If **-u** is not set, no error will be generated, and the variable value will be substituted with NULL. You can disable this option with: **set +u**. See the man page **sh-posix(1)** for more details.

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?

Answer:

```
$ touch "White Space"
$ touch " "
$ touch " "
```

When you do an **ls** command, the file names with the leading blanks will appear at the beginning of your directory listing. One way to determine how many blanks are in the file names is to make the files executable and then execute an **ls -F** and observe how many blanks occur before the asterisk, you can also try an **ls -b** command.

Notice that with the quoting mechanism you can create a file name that contains *any* special character. You will always have to use the escape characters though whenever you want to reference the file name on your command line.

11-11. LAB: Input and Output Redirection

1. Redirect the output of the **date** command to a file called **date.out** in your *HOME* directory.

Answer:

```
$ cd
$ date > date.out
```

2. Append the output of the **ls** command to the file **date.out**. Look at the contents of **date.out**. What do you notice?

Answer:

```
$ ls >> date.out
$ more date.out
```

The output of the **ls** command is a list of files in the current directory. Each file name is on a separate line. The shell knows to put the output of the **ls** command in columns only when the output goes to the terminal. You can override this with the **-C** option to **ls**.

3. Using input redirection, mail the file **date.out** to your mail partner.

Answer:

```
$ mail mail_partner_login_name < date.out
```

4. Create two very short files called **f1** and **f2** using **cat** and output redirection.

Answer:

```
$ cat > f1
This is the file f1
Ctrl + d
$ cat > f2
This is the file f2
Ctrl + d
```

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?

Answer:

```
$ cat f1 f2
This is the file f1
This is the file f2
$ cat f1 f2 > f.join                    output of both files is sent to f.join
```

You will not see any output on the screen. All of the standard output has been sent to the file **f.join**.

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?

Answer:

```
$ cat f1 f2 f.new
This is the file f1
This is the file f2
cat: Cannot open f.new
```

It is not obvious that two output streams are being used, since all of the messages are sent to your display.

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.

Answer:

```
$ cat f1 f2 f.new 2> f.error
This is the file f1
This is the file f2
$ cat f.error
cat: Cannot open f.new
```

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.

Answer:

```
$ cat f1 f2 f.new > f.good 2> f.bad
$ cat f.good
This is the file f1
This is the file f2
$ cat f.bad
cat: Cannot open f.new
```

The files **f.good** and the file **f.bad** are created. You do not see any output to your screen because all output streams have been redirected to one file or the other.

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?

Answer:

```
$ cp
Usage: cp f1 f2
cp [-r] f1 ... fn d1
$ cp 2> cp.error
```

The **cp** command does not generate any standard output messages. It is normally silent when it succeeds.

10. Display the contents of the file **/etc/passwd** sorted out by user name.

Answer:

```
$ sort -d /etc/passwd
```

11. Sort the file **/etc/passwd** on the third field. What happens? Now do a numeric sort on the third field. Any difference?

Answer:

```
$ sort -t: -k 3 /etc/passwd          lexicographic sort
```

(Note that the numbers in the third field are not quite sorted. This is because an ASCII sort is being done on a numeric field.)

```
$ sort -nt: -k 3 /etc/passwd         numeric sort
```

(The results of this command are much better since the numbers in the third field are now arranged numerically.)

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*.

Answer:

```
$ grep user /etc/passwd > grepped
$ wc -l grepped
16 grepped
```

(Note that on the system you are using, this number may vary.)

13. Using redirection and filters, how many users are logged in on the system?

Answer:

```
$ who > whoson
$ wc -l whoson
```

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**.)

Answer:

```
$ wc -l /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**.

Answer:

```
$ sort names > names.sort
$ sort -r names > names.rev
$ sort names -o names
```

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?*

Answer:

You must first determine the device file associated with your mail partner's terminal:

```
$ who > whoson
$ grep mailpartner whoson
mailpartner tty03 Jul 16 8:02
```

Check out the **tty** designation. This tells you what device file is associated with your mail partner's terminal session.

```
$ banner good morning > /dev/tty03
```

If you get a *Permission denied* message, your mail partner has disabled the write permissions on his or her terminal with the command, **mesg n**.

12-13. LAB: Pipelines

1. Construct a pipeline that will count the number of users presently logged on.

Answer:

```
$ who | wc -l
```

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.

Answer:

```
$ grep home /etc/passwd | wc -l      Number of lines containing
                                     home
$ grep -v home /etc/passwd | wc -l   Number of lines not containing
                                     home
```

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.

Answer:

```
$ grep home /etc/passwd | tee all.users | wc -l
```

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.

Answer:

```
$ sort names | pr -3 -t
```

5. Create an alias called **whoson** that will display an alphabetical listing of the users currently logged into your system.

Answer:

```
$ alias whoson="who | sort"
```

6. Construct a pipeline to obtain a listing of just the user names of those users presently logged into the system.

Answer:

```
$ who | cut -c1-8

or

$ who | cut -f1 -d" "
```

7. Construct a pipeline to obtain a long listing of just file permissions and file names currently in your working directory.

Answer:

```
$ ll | cut -c2-10,58-
```

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.

Answer:

```
$ ll | cut -c16-24,34-44,58- | tee listing.out | wc -l
```

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**.

Answer:

```
$ cut -f1,3,6 -d: /etc/passwd | sort                              Alphabetical sort
$ cut -f1,3,6 -d: /etc/passwd | sort -n -t: -k 2    Numerical sort
```

13-11. LAB: Exercises

1. Use the **hostname** command to determine the name of your local system. What systems can you communicate with?

Answer:

The **hostname** command reports the local host name. By looking at the **/etc/hosts** file, you can see all of the computers your local computer can talk to.

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.

Answer:

```
$ telnet fred
Trying...
Connected to fred
Escape character is '^]'.

HP-UX fred 10.00 B 9000/715

login: enter your name
Password: enter your password
```

```
.
.
.
$ hostname
fred
$ exit
```

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.

Answer:

In **ftp**, you would use the **put** command, similar to the example given in the student notes.

4. Use **remsh** to list the contents of the remote directory to verify that the copy worked.

Answer:

```
$ remsh system ls
```

The **ls** command will list your *HOME* directory on *system*.

14-10. LAB: Adding and Deleting Text and Moving the Cursor

1. **vi** the file **tst**.

Answer:

$ **vi tst**

2. Insert the word *only* between the words *will be*.

Answer:

Move cursor to the second line, type **j** or Return or **2G**.
Move cursor to the last *l* in *will*, type **ee** or **2e** or several **l**s.
Append text after the **l**, type **a**; type **only** or
Move cursor to the first *b* in *be*, type **ww** or **2w** or several **l**s.
Insert text before the *b*, type **i**; type **only**.

3. Add the words *many, many* on the end of the line *It will be used for*.

Answer:

Go back to command mode, type ESC.
Move the cursor to the end of the line, type **$**.
Add (append) the text, **a**; enter text **many, many**.

4. Add a new blank line at the end of the file, and enter your name. DON'T PRESS THE ESC !

Answer:

Go back to command mode, type [ESC].
Move cursor to the last line: `G`.
Open a new line below, `o`.
Type your name.

5. Using the [Backspace], remove your name and enter your partner's name.

Answer:

[Backspace] over the first name.
Type in your partner's name.

6. Open a new line at the top of your file (Hint: `O`).

Answer:

Go back to command mode, type [ESC].
`1GO`

7. Enter `12345`.

Answer:

`12345`

8. [Backspace] 2 times. Do any of the numbers disappear from your display?

Answer:

[Backspace] [Backspace]
The cursor will be under the 4. No characters disappear.

9. Enter `1234`. What happens to the numbers that you backspaced over?

Answer:

The 4 and 5 will be typed over.

10. [Backspace] 3 times.

Answer:

[Backspace] [Backspace] [Backspace]
The cursor is now under the second 2.

11. Press [ESC] . What happens to the characters you backspaced over? Where does the cursor end up?

Answer:

The second 234 will disappear, and the cursor will back up so it is under the 1.

12. Type in 4 **a**'s. How many a's appear? Why?

Answer:

Three a's appear because the first a is taken to be the **vi append** command.

13. Backspace 5 times. What happens? Why?

Answer:

You can only Backspace three times, because you have only entered 3 letters in this input session.

14. Press ESC. What happens?

Answer:

All of the a's you just entered disappear. You are back in command mode.

15. Quit your **vi** session saving the changes you made to the file **tst**.

Answer:

Enter **:wq** or **ZZ**

14-19. LAB: Modifying Text

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*.

Answer:

```
:1,$s/bug/FEATURE/g
```

3. Write the first forty lines of the **`funfile`** out to another file called **`new.40`**.

Answer:

```
:1,40w new.40
```

4. Go to the last line in **`funfile`**.

Answer:

```
G
```

5. Find and execute the command to place your cursor midway down the window.

```
This file is silly.
```

Answer:

```
This file is silly.
```

ESC

6. Without quitting **`vi`**, write your new version of the file out to a file called **`funfile.123`**.

Answer:

```
:w funfile.123
```

7. Without leaving **`vi`**, load the file **`new.40`** into the buffer, overwriting the previous contents.

Answer:

```
:e new.40
```

8. Turn on line numbering with the **ex** *number* option.

Answer:

```
:set number
```

9. Search for an occurrence of **FEATURE** in **new.40**.

Answer:

```
/FEATURE
```

10. Change all occurrences of *FEATURE* to *BUG* and save it into **new.new.40**.

Answer:

```
cwBUG ESC
```

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*.

Answer:

```
1G
/[Ss]ystem
cwXXXXX ESC
n
.
n
n
.
n
.
n
.
```

12. Write your current edit session and quit the editor.

Answer:

```
:wq
```

or

```
ZZ
```

15-7. LAB: Process Control

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
hello
hello
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.

Answer:

The PID (process ID number) of the shell (-sh) will be the PPID (parent process ID number) of the **infinite** command. When you log out, terminating the parent process, all child processes (including **infinite**) are killed.

3. The **nohup** command protects a process from terminating upon the death of its parent process. Re-run 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?

Answer:

When the parent process (your shell) dies, the child process (**infinite**) becomes an **orphan** process. Orphan processes are **adopted** by PID 1 (**init**). When you log back in, you will see **infinite** still running.

4. Use the **kill** command to terminate your **infinite** program.

Answer:

```
$ kill PID                    PID is returned from the
                              ps command
```

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.

Answer:

```
$ infinite > infin.out
Ctrl + z
[1] + Stopped infinite > infin.out
$ bg %1
[1] infinite > infin.out &
$ kill %1
[1] + Terminated infinite > infin.out
```

16-9. LAB: Introduction to Shell Programming

1. Create a shell program called **whoson1** that will

- display a greeting to the user with the **banner** 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 into the system

Answer:

```
$ vi whoson1
banner Welcome to whoson1
MYNAME=your_name
echo $MYNAME
date
who
$ chmod +x whoson1
$ whoson1
```

2. Change to the **/tmp** directory. Invoke the program **color1**. Does the shell find the **color1** program?

Answer:

```
$ cd /tmp
$ color1
sh: color1: not found
```

The **color1** program is not found because it does not reside under one of the directories specified by the *PATH* variable.

3. Change to $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.

Answer:

```
$ cd
$ mkdir bin
$ mv color1 bin
$ PATH=$PATH:$HOME/bin
$ cd /tmp
$ color=lavender
$ export color
$ color1
You are now running program: color1
the value of the variable color is: lavender
$
```

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.

Answer:

```
$ date_var=$(date)
$ export date_var
$ cd $HOME/bin
$ vi date_tst
echo the value of date_var is $date_var
$ chmod +x date_tst
$ date_tst
```

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**?

Answer:

```
$ vi $HOME/bin/date_tst
date_var=$(date)
echo the value of date_var is $date_var
$ date_tst
```

In this case *date_var* does not need to be exported because it is being defined and used within the same process level. The permissions on **date_tst** do not need to be changed because the program was made executable in the previous exercise. Editing a file does not affect its permissions.

6. Create a shell program called **whoson2** that will

- Display a personalized greeting to the user with the **banner** command, such as *welcome username*, so that if user3 was logged in it would banner *welcome user3* or if user2 was logged in it would banner *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.

Answer:

```
$ vi $HOME/bin/whoson2
banner welcome $LOGNAME     or     banner welcome $(whoami)

date
who
echo your terminal session identification information is
who am i
echo thank you for using whoson2
$ chmod +x $HOME/bin/whoson2
$ whoson2
```

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
```

Answer:

```
6
-d
A blank line.
abc def -d -4 +900 xyz
myshellprog
```

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
```

Answer:

```
4
+900
A blank line.
```

```
-d -4 +900 xyz
myshellprog
```

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"
```

Answer:

```
Please type in your first, middle, and last names
James A. Smith, Jr.
Smith, Jr., James A.
```

Note that "Smith, Jr." is read into the last variable.

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.

Answer:

This is shell program **search1**:

```
echo "Please enter a string to search for: \c"
read string
echo The following files contain the string $string:
grep -l $string *
```

11. Write a shell program called **backwards** that will receive up to ten arguments and list the arguments in reverse order.

Answer:

```
#!/usr/bin/sh
# backwards: reverses command line arguments
# usage: reverse a b c d e f g h i
#
echo ${10} $9 $8 $7 $6 $5 $4 $3 $2 $1
```

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.
```

Answer:

```
#!/usr/bin/sh
# myecho: Display the number of command line arguments,
#         print the first three arguments on separate lines
#         and print the remaining arguments on one line
#         usage: myecho a b c
#
echo "The number of command line arguments is $#."
#
echo $1;echo $2;echo $3
shift 3
echo $*
$ myecho a b c d e f g h i j k l
The number of command line arguments is 12
a
b
c
d e f g h i j k l
$ myecho "I cannot" "seem to" "find my KEYS."
```

13. Create a program **my_vi** that will accept a command-line argument which 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.

Answer:

```
#!/usr/bin/sh
# my_vi: Create a backup file prior to starting a vi session
# usage: my_vi filename
#
echo Copying $1 to ${1}.bak
cp $1 ${1}.bak
vi $1
echo Edit of $1 is complete
echo You may recover your original file from ${1}.bak
```

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**.

Answer:

```
#!/usr/bin/sh
# my_recover: Recover a file from backup
# usage: my_recover filename
echo Restoring $1 from ${1}.bak
cp ${1}.bak $1
echo $1 is recovered
```

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.

Answer:

```
#!/usr/bin/sh
# info: Prompt user for mailing address
#
echo "Input your name: \c"
read name
echo "Input your street address: \c"
read address
echo "Input your City, State, and Zip Code: \c"
read where
echo;echo
echo "Your name is  $name"
echo "You live at   $address"
echo "              $where"
```

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.

Answer:

```
#!/usr/bin/sh
# home: Return the value of a user"s HOME directory
# usage: home
echo Select a user identifier from the following list:
cut -f1 -d: /etc/passwd | pr -4 -t
echo "Input user identifier: \c"
read user
home=$(grep $user /etc/passwd | cut -f6 -d:)
echo;echo "user:$user HOME directory: $home"
```

17. Write a shell program called **alpha** that will display the first and last command line arguments. Hint: use the **cut** command.

Answer:

```
#!/usr/bin/sh
# alpha: Displays the first and last command line arguments
#
last=$(echo $* | cut -f$# -d" ")
echo "The first command line argument is $1."
echo "The last command line argument is $last."
```

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.

Answer:

```
#!/usr/bin/sh
# file_copy: User interface for copying files
# usage: copy
#
echo Please enter the names of the file(s) you want to copy:
echo "-> \c"
read filenames
echo Please enter the destination.
banner NOTE!
echo If you entered more than one file, the destination must be a
directory.
echo "Enter destination here -> \c"
read dest
echo Copying files now ...
cp $filenames $dest
echo Done copying files "\a"
```

17-13. LAB: Shell Programming — Branches

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.)

Answer:

```
$ X=xyz
$ test "$X" = "xyz"
$ echo $?
0
```

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.)

Answer:

```
$ Y=100
$ test "$Y" -gt 0    or    [ "$Y" -gt 0 ]
$ echo $?
0

$ Y=-100
$ test "$Y" -gt 0    or    [ "$Y" -gt 0 ]
$ echo $?
1
```

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.

Answer:

```
if
    [ "$1" = "abc" ]
then
    echo yes
else
    echo no
fi
```

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 which 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.)

Answer:

```
echo "please enter a number: \c"
read Y
if
    [ "$Y" -gt 0 ]
then
    echo Y is positive
    echo The value of Y is $Y
 else
    echo Y is not positive
    echo The value of Y is $Y
fi
```

5. Write a shell program which 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?)

Answer:

```
if
    [ "$#" -ne 3 ]
then
    echo "there are not exactly three command line arguments" >&2
else
    echo $*
fi
```

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.

Answer:

```
echo "Please input A, B, b, or C: \c"
read input
case $input in
         A) echo good morning
            ;;
      [Bb]) echo good afternoon
            ;;
    C|quit) exit
            ;;
         *) echo You entered an illegal option.
            exit 99
            ;;
esac
```

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.

Answer:

```
echo "Input a user login name -> \c"
read user
if
    grep $user /etc/passwd > /dev/null
then
    home=$(grep $user /etc/passwd | cut -f6 -d:)
    echo The HOME directory for $user is $home
else
```

```
    echo;echo "$user is not here!!!"
fi
```

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.)

Answer:

```
time=$(date | cut -c12-20)
hour=$(echo $time | cut -f1 -d:)

if [ $hour -lt 12 ]
then
    echo good morning
else
    if [ $hour -ge 12 -a $hour -lt 18 ]
    then
        echo good afternoon
    else
        echo good evening
    fi
fi
```

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.

Answer:

```
echo Would you like to see the contents of your current directory?
echo Please enter yes or no.
echo "----> \c"
read ans1
case $ans1 in
     yes) ls
          ;;
      no) echo What directory would you like to see?
          read ans2
          if test -d $ans2
          then
              ls $ans2
          else
              echo directory $ans2 does not exist
          fi
          ;;
       *) echo You have not entered a proper response.
          echo Please try again.
          ;;
esac
```

10. Create a program **mycp** which 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.

Answer:

```
#!/usr/bin/sh
# mycp file1 file2
if [ "$#" -ne 2 ]
    then
        echo "Usage: $0 file1 file2" >&2
        exit 1
fi
if [ "$1" = "$2" ]
then
    echo "$1 = $2"
    echo "No copy performed" >&2
    exit 2
fi

if test -d $2
then
    echo "Your target is a directory." >&2
    echo "No copy performed!" >&2
    exit 4
fi

if test -f $2
then
    echo "Your target file already exists."
    echo "Do you want to overwrite it? [y/n]: \c"
    read ans
    if [ "$ans" != "y" -o "$ans" != "Y" ]
    then
        echo "No copy performed!" >&2
        exit 3
    fi
fi

if test -f $1
then
    cp $1 $2
    echo "Copy complete"
else
    echo "Source file does not exist"
    echo "No copy performed"
    exit 4
fi
```

11. Write a shell program called **options** which 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**.

Answer:

```
if
    [ "$#" -lt 1 -o "$# "-gt 2 ]
then
    echo "usage: $0 -d" >&2
    echo " $0 -l directory" >&2
    echo " $0 -w" >&2
    exit 10
fi
case $1 in
     -d) date
         ;;
     -w) who
         ;;
     -l) if test -d $2
         then
             echo the contents of directory $2 are:
             ls -F $2
         else
             echo directory $2 does not exist
         fi
         ;;
      *) echo "bad option" >&2
         echo "usage: $0 -d" >&2
         echo " $0 -l directory" >&2
         echo " $0 -w" >&2
         exit 20
         ;;
esac
```

18-12. LAB: Shell Programming — Loops

1. Create a program called **double_it** that will prompt the user for a number and then display two times the number.

Answer:

```
#!/usr/bin/sh
# double_it: Prompt the user for a number and then display 2 times
#            its value.
#
echo "Input an integer value: \c"
read num
echo "Two times the number you entered is \c"
let num=num*2
echo $num
```

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.

Answer:

```
#!/usr/bin/sh
# sum_them: Prompt the user for 10 numbers and add them together
#
sum=0
count=1
echo You will be prompted to enter 10 numbers.
echo Their sum will be displayed after all 10 numbers have been entered.
while
     [ $count -le 10 ]
do
     echo "Please enter a number ($count): \c"
     read num
     let sum=sum+num
     let count=count+1
done
echo The sum of the 10 numbers you entered is: $sum
```

Optional solution supporting a command line argument identifying the number of numbers to enter:

```
#!/usr/bin/sh
# sum_them2: The user will provide the number of numbers to
#            add together as a command line argument
#
if
    [ $# -ne 1 ]
then
    echo Usage: $0 number >&2
```

```
    exit 99
fi

count=1
echo You will be prompted to enter $1 numbers.
echo Their sum will be displayed after all $1 numbers have been entered.
while
    [ $count -le $1 ]
do
    echo "Please enter a number ($count): \c"
    read num
    let sum=sum+num
    let count=count+1
done
echo The sum of the $1 numbers you entered is: $sum
```

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.)

Answer:

```
#!/usr/bin/sh
# words_in: prompt the user to input words until "quit" is entered
#
until
    echo Please enter a word. Enter "quit" when you are done.
    read input
    [ "$input" = quit ]
do
    words="$words\n$input"
done
echo $words
#Print words out in alphabetical order
echo $words | sort
```

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

Answer:

```
for name in Adir Bdir Cdir Ddir Edir
do
    mkdir $name
    cp $HOME/funfile $name
    ls $name
```

```
    echo done with $name
done
```

an alternative method could be:

```
for name in A B C D E
do
    mkdir ${name}dir
    cp $HOME/funfile ${name}dir
    ls ${name}dir
    echo done with $name
done
```

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.

Answer:

```
#!/usr/bin/sh
# new_files: create new files as provided by the command line arguments
# Usage: new_files f1 f2 f3 f4 ...
#
for i in $*
do
    echo creating file $i
    touch $i
done
```

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.)

Answer:

```
$ for i in $(who | cut -f1 -d" ")
> do
>     mail $i < mailtest
> done
```

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.

Answer:

```
#!/usr/bin/sh
# my_menu: A menu interface
```

```
# Usage: my_menu
#
until
   [ $ans -eq 3 ]
   clear
   echo
   echo
   echo
   echo 1) Double a number.
   echo 2) Add together 10 numbers.
   echo 3) Quit
   echo
   echo "Enter your selection number ->\c"
   read ans
do
   case $ans in
     1) double_it
        ;;
     2) sum_them
        ;;
3|quit|q|Q) exit
        ;;
     *) echo You have not entered a legal option.
        echo Please try again.

        ;;                              screen clears before displaying menu
   esac
   sleep 3
done
```

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.

Answer:

```
#!/usr/bin/sh
# msg_me: display a message to your terminal every 5 seconds
#
term=$(who am i | cut -c12-18)
count=1
while
    [ $count -lt 12 ]
do
    cat msg.file > /dev/$term
    sleep 5
    let count=count+1
done
```

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.

Answer:

```
#!/usr/bin/sh
# ison: Check for a user to log into the system
#       Usage: ison username
#
if [ "$#" -ne 1 ]
then
    echo "usage: $0 user_id" >&2
    exit 99
fi

until who | grep $1 > /dev/null
do
    sleep 60
done

# When you reach this point, the user has logged in

echo $1 has logged on
who | grep $1
```

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

Answer:

```
#!/usr/bin/sh
# myrm: WasteBasket
#
if [ "$1" = "" ]
   then
   echo "Usage: $0 file [file ...]" >&2
   echo " or: $0 [-l] | [-d]" >&2
   exit 5
fi
opt=$(echo $1 | cut -c1)
if [ "$opt" = "-" ]
then
   case $1 in
     -l) echo;echo "The WasteBasket includes the following files:"
         ls $HOME/.waste;;
     -d) echo;echo "The WasteBasket is being dumped!"
         rm $HOME/.waste/*;;
     -*) echo "$0: $1 invalid argument" >&2
         exit;;
   esac
else
   echo "Are you sure you want to remove $*? [y/n]: \c"
   read ans
   if [ "$ans" = "y" -o "$ans" = "Y" ]
   then
      for i in $*
      do
        if test -f $i
        then
          mv "$i" $HOME/.waste
        else
          echo "$i: Does not exist" >&2
        fi
      done
   else
      exit
   fi
fi
```

19-4. LAB: Offline File Storage

1. Using **tar**, create an archive of all files in your *HOME* directory that start with **abc**.

   Answer:

   ```
   $ tar cf /dev/rmt/0m abc*
   ```

2. Obtain a table of contents listing of this tape archive.

Answer:

```
$ tar tf /dev/rmt/0m
```

3. Using **find** and **cpio**, make a backup of your whole directory structure from your *HOME* directory on down.

Answer:

```
$ cd
$ find . | cpio -ocv > /dev/rmt/0m
```

4. Remove the file **backup** from your current directory. Then restore the file from tape using the **cpio** command.

Answer:

```
$ rm backup
$ cpio -iumc "backup" < /dev/rmt/0m
$ ll backup
```

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**.

Answer:

```
$ mkdir $HOME/tree.cp
$ cd $HOME/tree
$ find . | cpio -pcduv $HOME/tree.cp
```

Glossary