HP Computer Systems
Training Course

Fundamentals of the UNIX System for HP Channel Partners

Student Workbook

Version G.00
51434P Student
Printed in USA 01/99

Notice

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

Contents

Module 4 — File Permissions and Access

Module 5 — Shell Basics

Module 6 — Shell Advanced Features

Module 7 — Input and Output Redirection

Module 8 — Pipes

Module 9 — Using Network Services

Module 10 — Process Control

Module 11 — Introduction to Shell Programming

Module 12 — Shell Programming — Branches

Module 13 — Shell Programming — Loops

Appendix A — Commands Quick Reference Guide

Solutions

Figures

Tables

Overview

Student Performance Objectives

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.

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.

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.

File Permissions and Access

- 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.

Shell Basics

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

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.

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**.

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.

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.

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.

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.

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.

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.

Commands Quick Reference Guide

- To provide a list of frequently used commands along with an explanation of proper use.

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, but is designed to run in conjunction with the self study course modules.

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.

Module 1 — 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.

1-1. SLIDE: Logging In and Out

Logging In and Out

```
login: user1 [Return]                  Log in
Password:
Welcome to HP-UX                       Login messages
Erase is Backspace
Kill is Ctrl-U
$ date                                 Do work
Fri Jul 1 11:03:42 EDT 1994
    :
$ other commands

$ exit or [Ctrl] + [d] [Return]        Log out
login:
```

a56610

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.

1-2. SLIDE: Command Line Format

Command Line Format

Syntax:

```
$ command [-options] [arguments] Return
```

Examples:

```
$ date Return                          No argument
Fri Jul  1 11:10:43 EDT 1994
$ banner hi Return                     One argument
 #    #    #
 #    #    #
 ######    #
 #    #    #
 #    #    #
 $ bannerHi  Return                    Incorrect syntax
 sh: bannerHi : not found
 $ ls -F Return                        One option
   dira/  dirb/    f1    f2  prog1* prog2*
```

a56612

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.

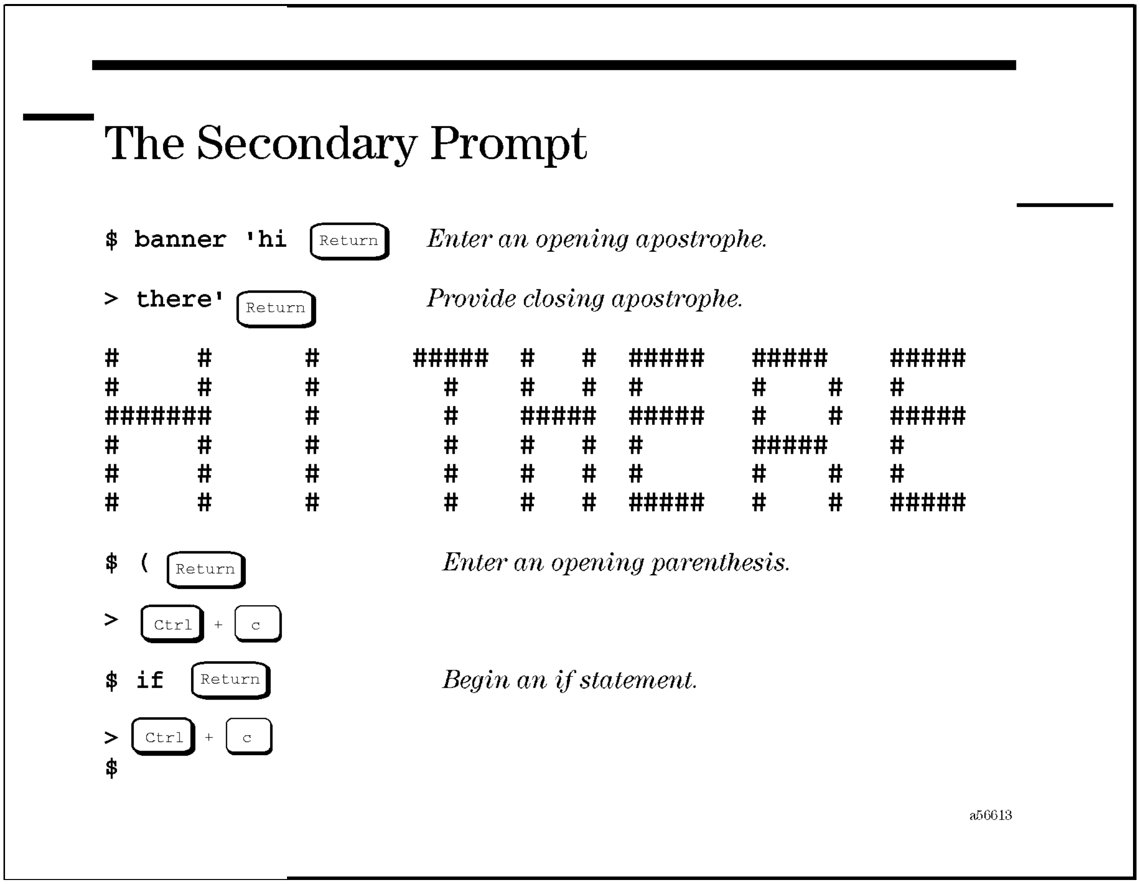
The terminal input/output supports typing ahead. This allows you to enter a command and then enter the next command(s) before the prompt is returned. The command will be buffered and executed when the current command has finished.

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`**

1-3. 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.

1-4. SLIDE: The Manual

The Manual

The *HP-UX Reference Manual* contains:

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

a56614

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.

1-5. SLIDE: Some Beginning Commands

Some Beginning Commands

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

a65030

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.

1-6. 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. 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
```

3. 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?

4. 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)**.

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

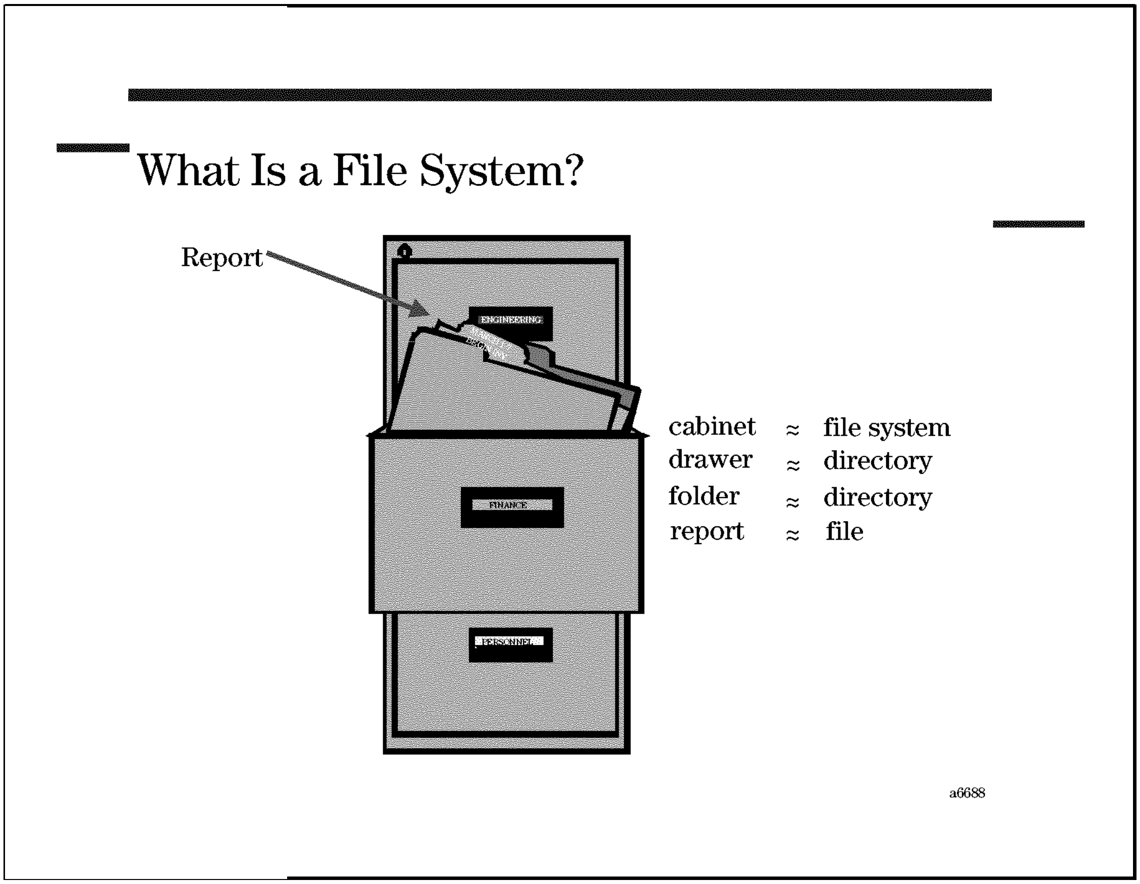
Module 2 — Navigating the File System

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.

2-1. SLIDE: What Is a File System?

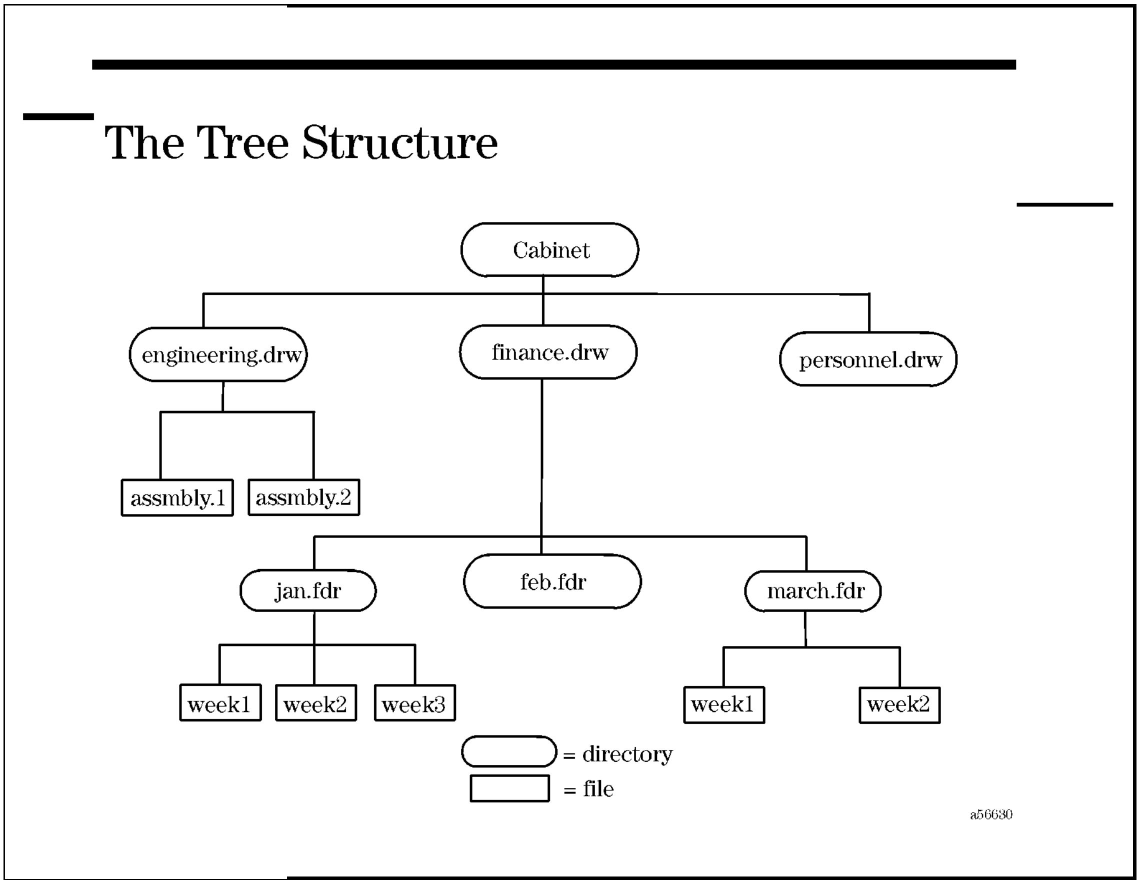


Student Notes

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.

2-2. SLIDE: The Tree Structure

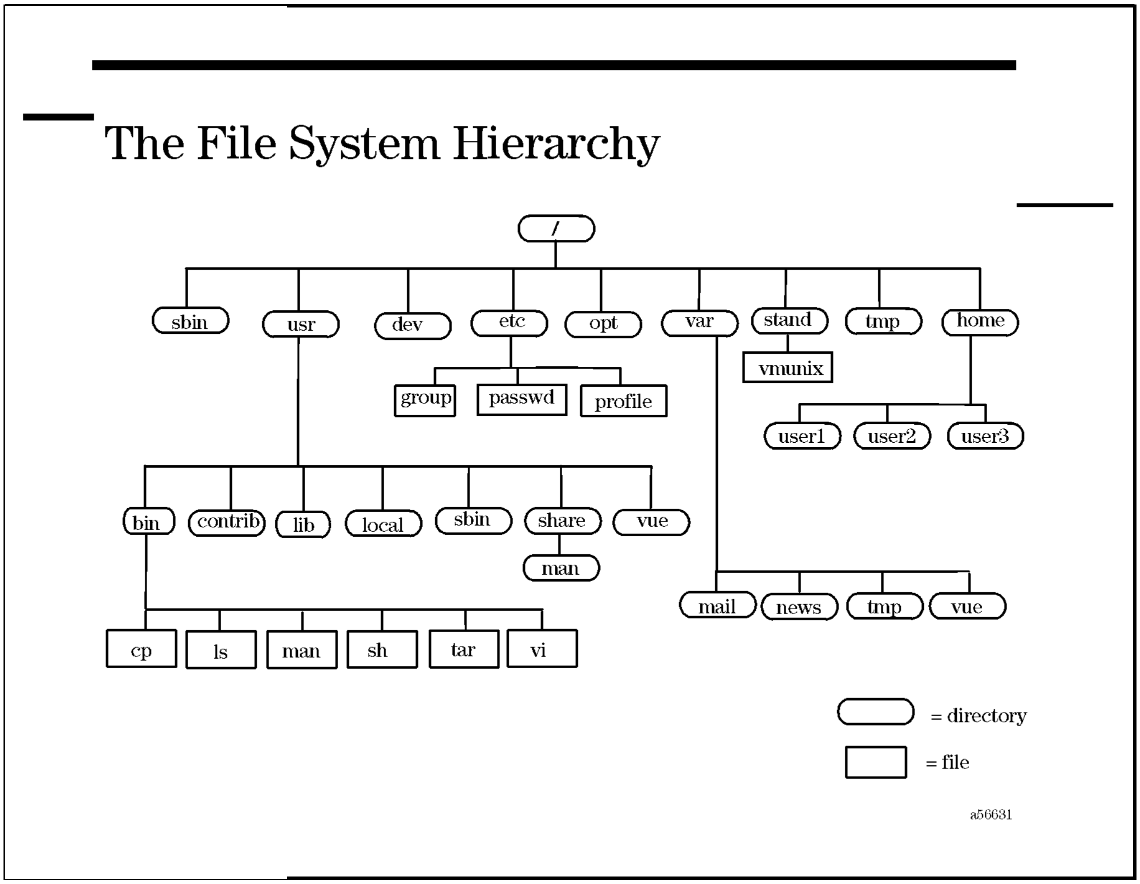


Student Notes

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.

2-3. 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. This directory is cleared during reboot. 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**.

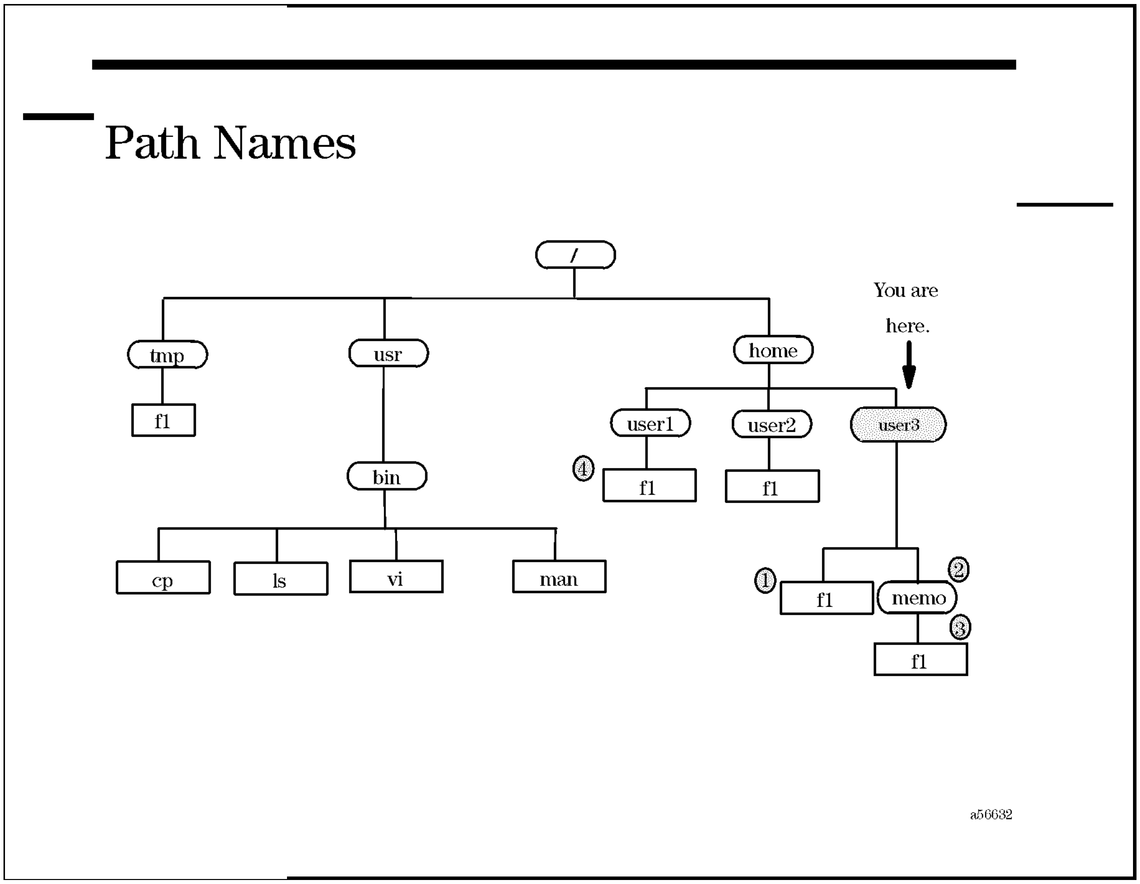
/mnt — This directory will be used to mount other devices (laserROM for instance).

/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.

2-4. 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:

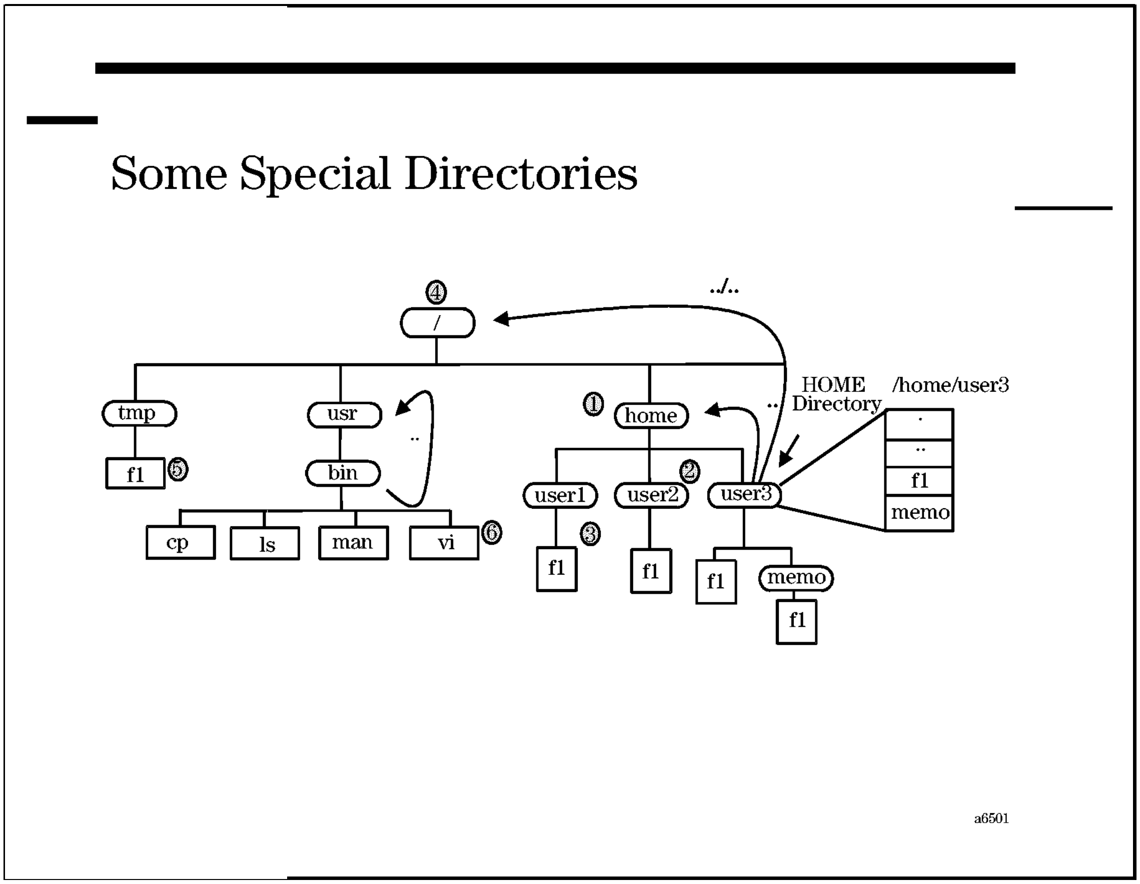
Absolute path name	**lp /home/user3/memo/f1**
Relative path name	**lp f1**

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.

2-5. 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/f1	represents **/home/user1/f1**
../../tmp/f1	represents **/tmp/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.

2-6. SLIDE: Basic File System Commands

Basic File System Commands

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

a56631

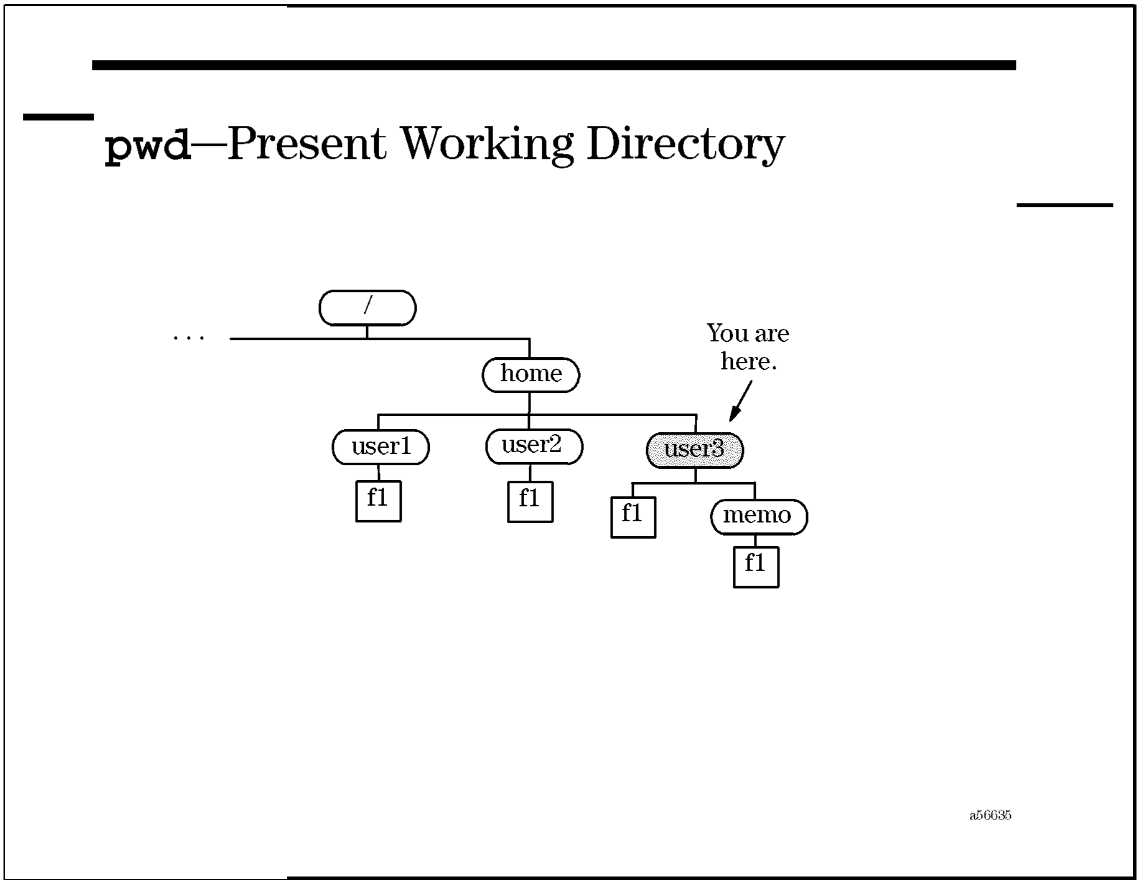
Student Notes

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.

2-7. 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.

2-8. SLIDE: **ls** — List Contents of a Directory

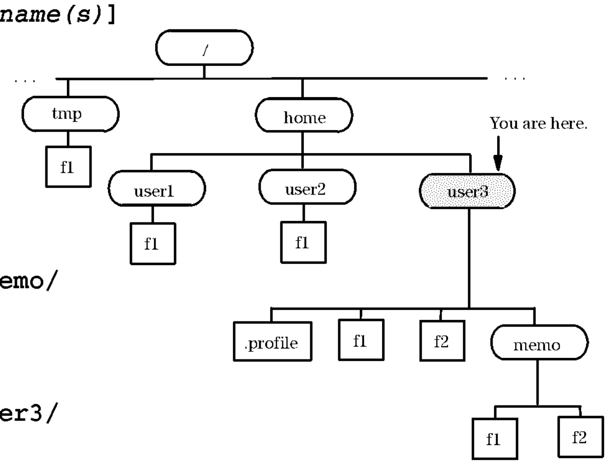
ls—List Contents of a Directory

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
```



a56636

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
```

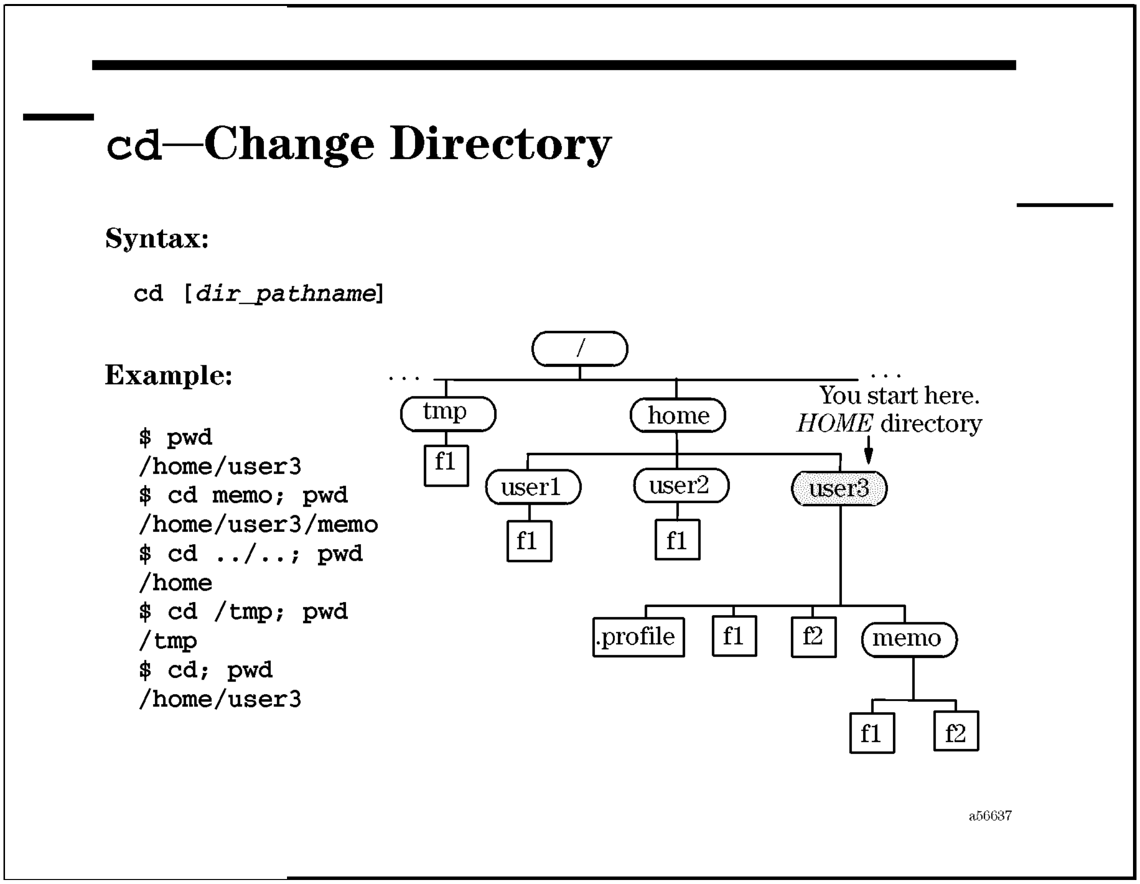
HP-UX Shorthand Commands

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`

2-9. SLIDE: cd — Change Directory



Student Notes

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

The syntax is

```
cd path_name
```

in which *path_name* is the relative or absolute path name of the directory to which you would like to go. When executed with no arguments, the **cd** command will return you to your login or *HOME* directory. 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.

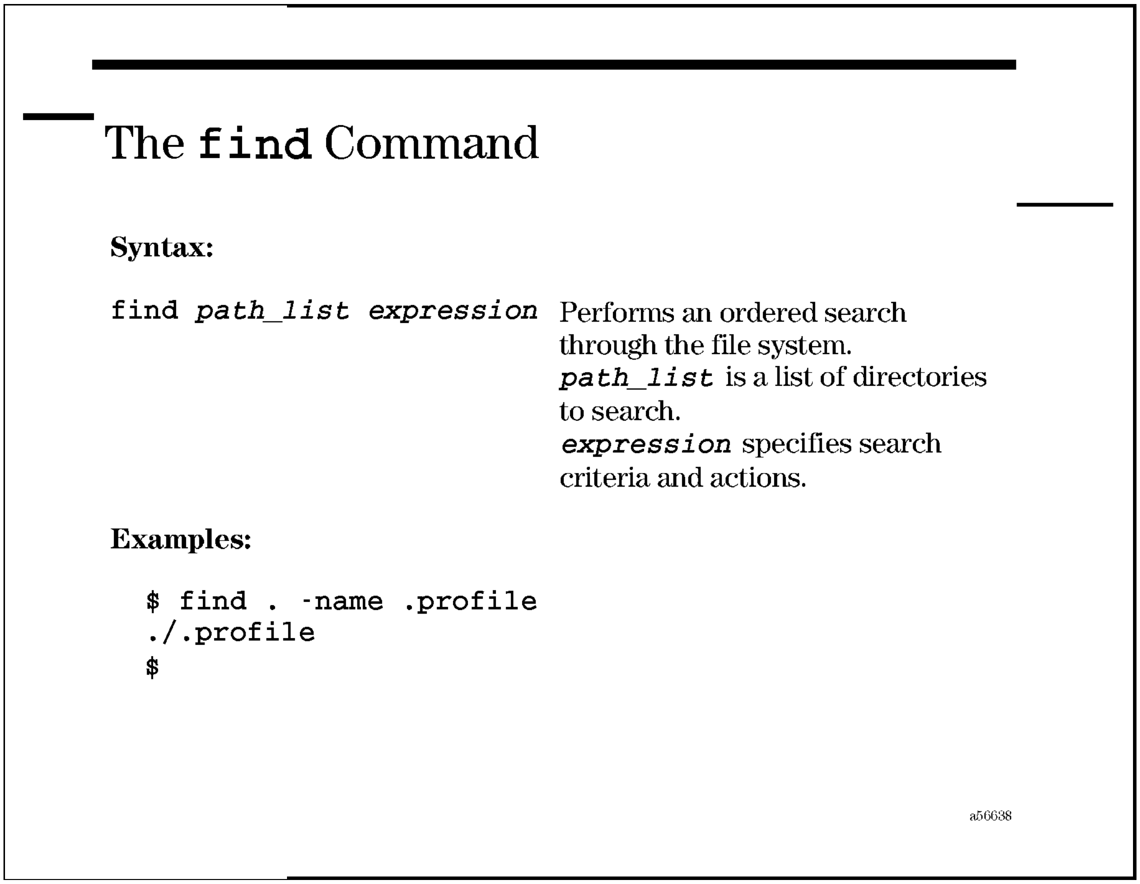
POSIX Shell Enhanced cd

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
```

2-10. 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.

2-11. SLIDE: mkdir and rmdir — Create and Remove Directories

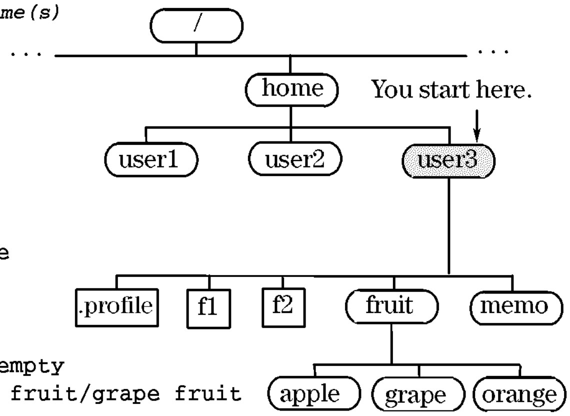
mkdir and rmdir—Create and Remove Directories

Syntax:

```
mkdir [-p] dir_pathname(s)
rmdir dir_pathname(s)
```

Example:

```
$ pwd
/home/user3
$ mkdir fruit
$ mkdir fruit/apple
$ cd fruit
$ mkdir grape orange
$ rmdir orange
$ cd ..
$ rmdir fruit
  rmdir: fruit not empty
$ rmdir fruit/apple fruit/grape fruit
```



a56639

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*.

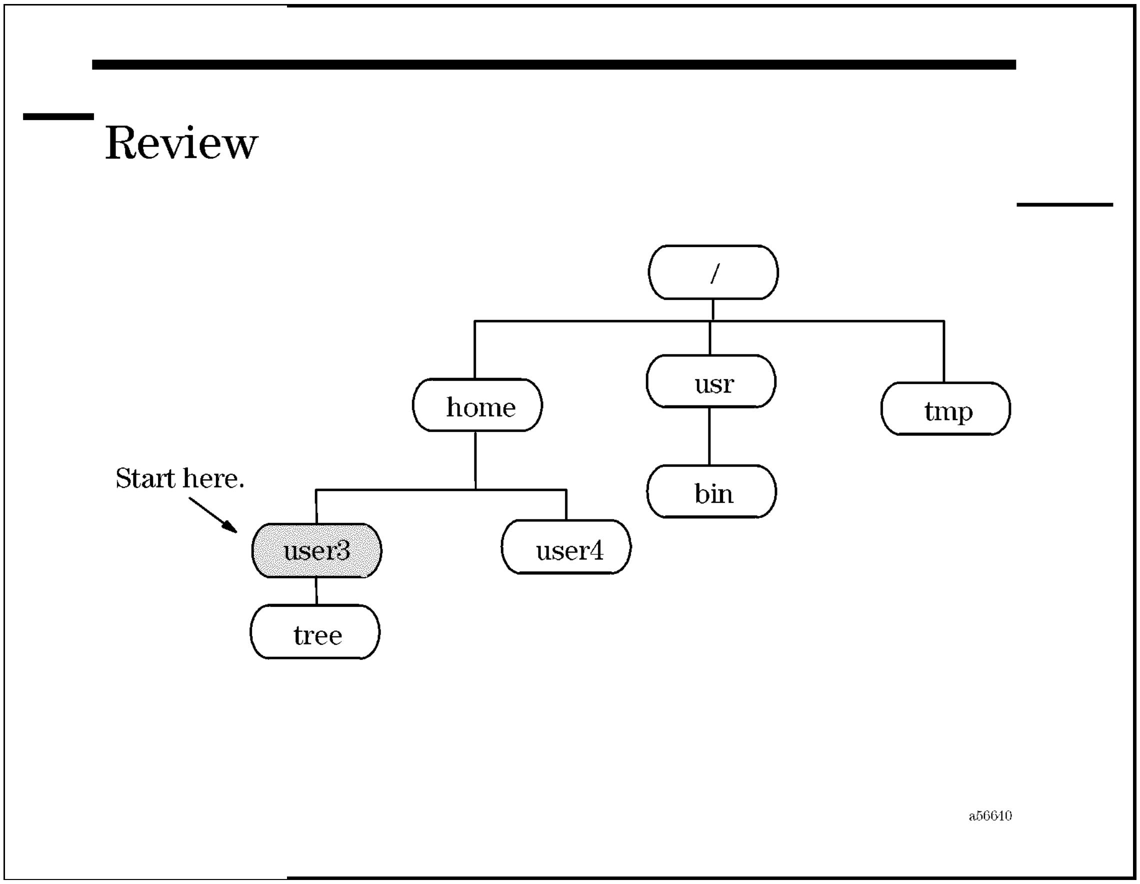
The following command would make the **fruit** directory if it does not already exist:

```
$ 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.

2-12. 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
```
____

2-13. SLIDE: The File System — Summary

The File System–Summary

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

a56641

Student Notes

2-14. LAB: The File System

Directions

Complete the following exercises and answer the associated questions.

1. 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.

2. 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?

3. 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?

Module 3 — 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.

3-1. SLIDE: What Is a File?

What Is a File?

A container for data or a link to a device.

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

a56643

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.

3-2. SLIDE: What Can We Do with Files?

What Can We Do with Files?

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

a56641

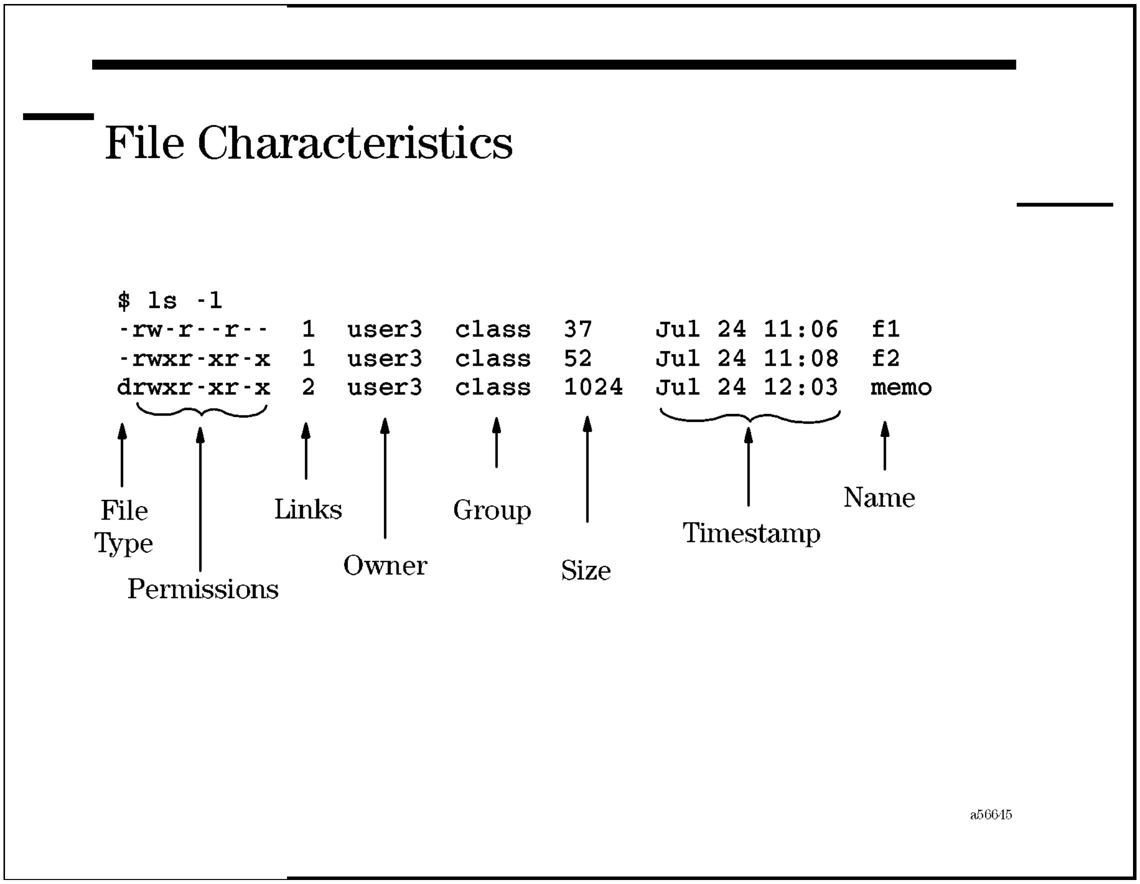
Student Notes

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`**.

3-3. 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*

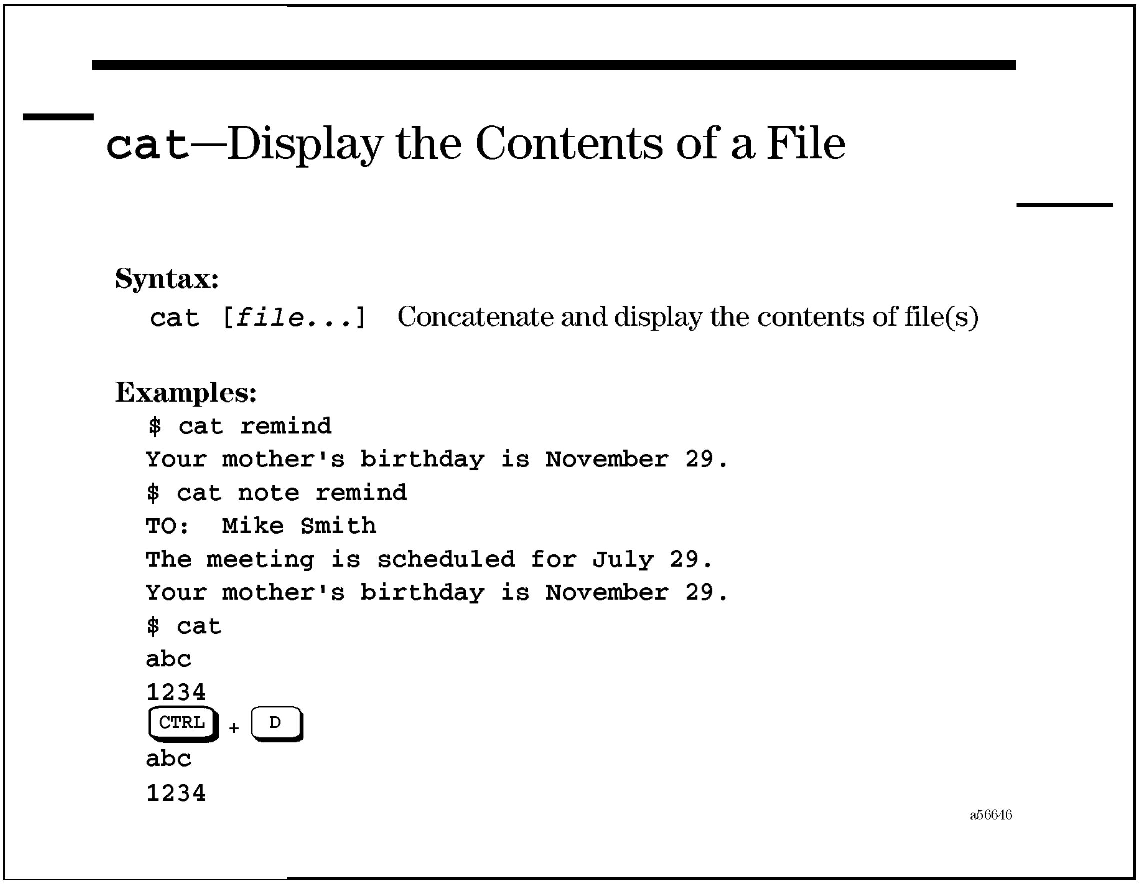
In the UNIX system the dot (.) is just a regular character, and, therefore, can appear anywhere (and multiple times) in a file name, making file names **`a.bcdefg`**, **`a.b.c.d`** and **`a...b`** legal. Dot is only somewhat special when it appears as the *first* character of a file name, in which case 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)

3-4. 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:

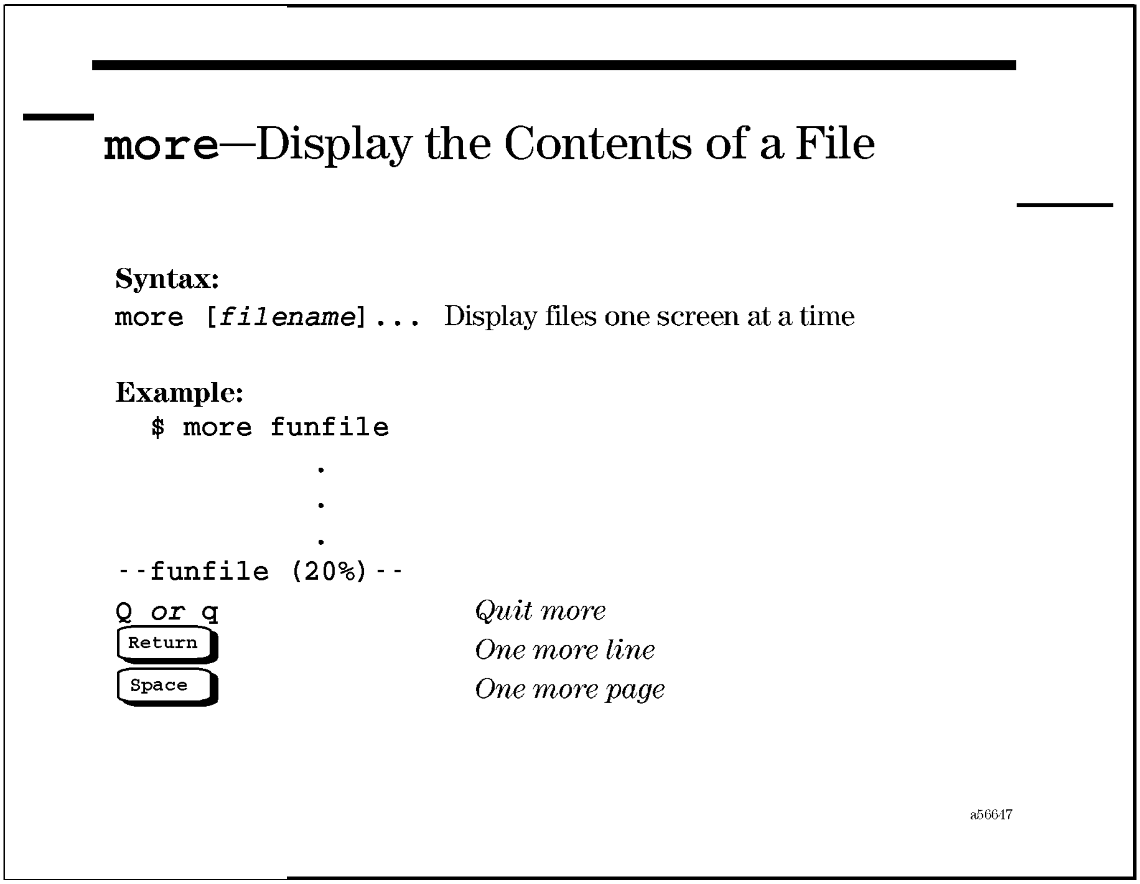
1. Try to log out—press Return and then issue the **exit** command.
2. Power cycle your terminal—turn it off, and then turn it on.
3. Log back in—you should be able to log in and continue normally.

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.

3-5. SLIDE: more — Display the Contents of a File

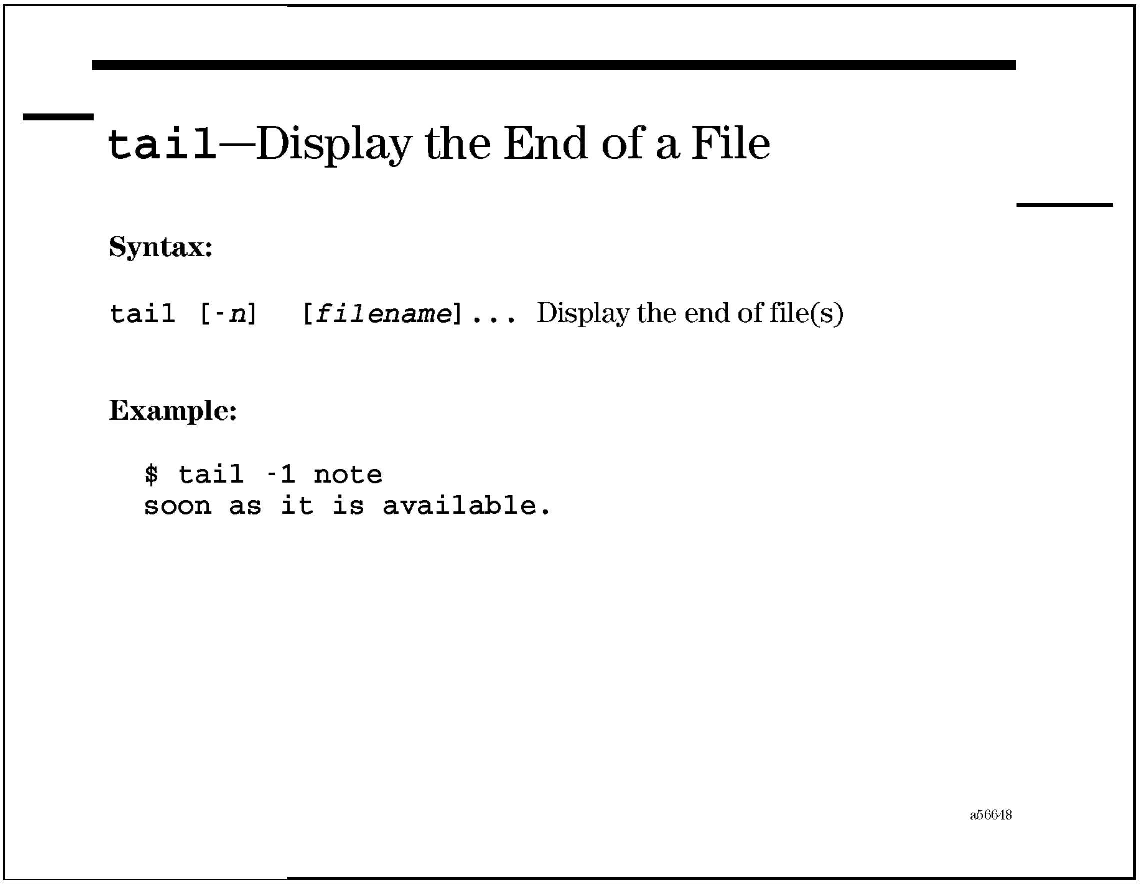


Student Notes

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.

3-6. SLIDE: **tail** — Display the End of a File



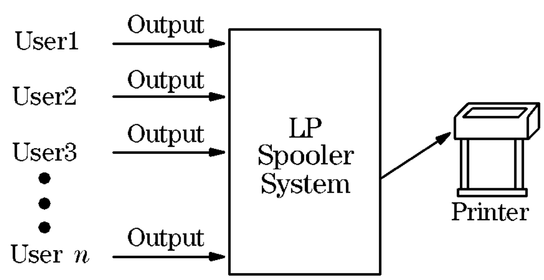
Student Notes

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

3-7. SLIDE: The Line Printer Spooler System

The Line Printer Spooler System

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



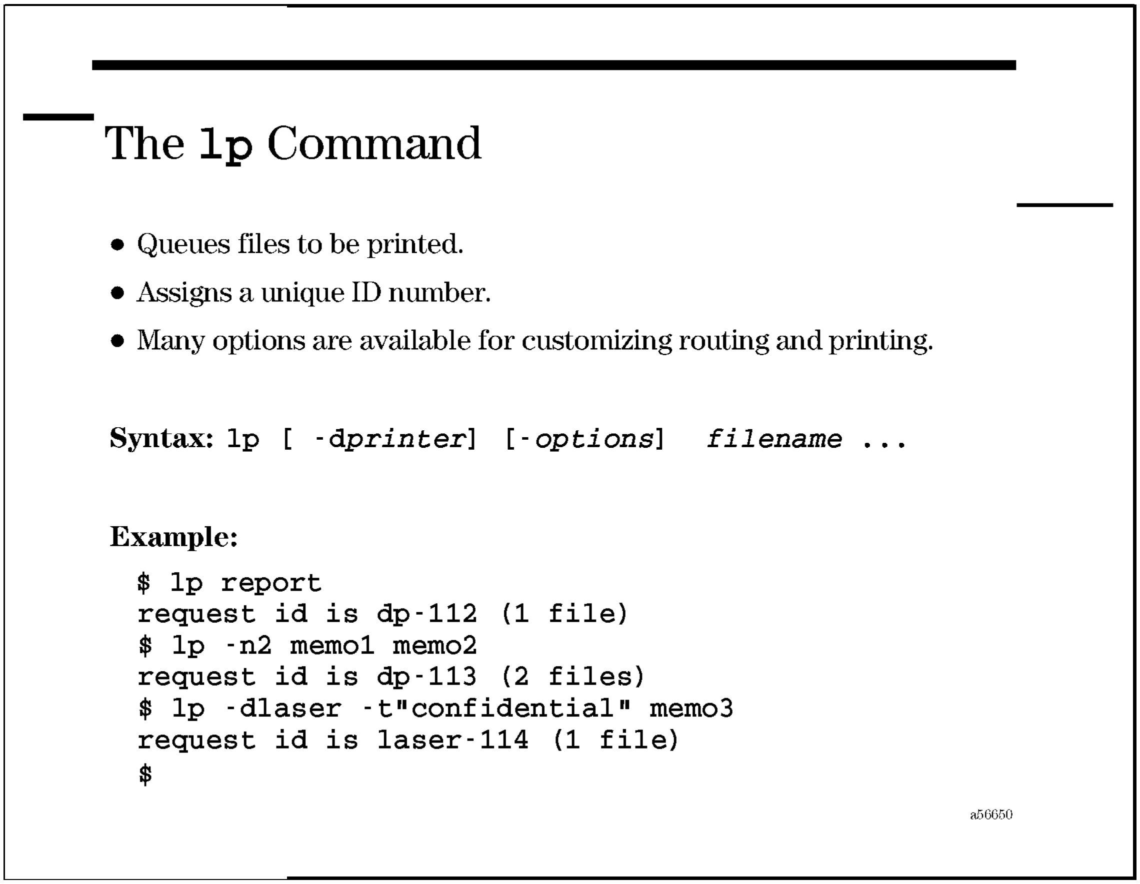
a56649

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.

3-8. SLIDE: The **lp** Command



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] [-ttitle] [-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."

3-9. SLIDE: The lpstat Command

The lpstat Command

Syntax:

```
lpstat [-t]
```

- `lpstat` reports the requests that you have queued to be printed.
- `lpstat -t` reports the status of the scheduler, default printer name, device, printer status, and all queued print requests.

a56651

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.

3-10. 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.

3-11. SLIDE: cp — Copy Files

cp—Copy Files

Syntax:

`cp [-i] file1 new_file`	Copy a file
`cp [-i] file [file...] dest_dir`	Copy files to a directory
`cp -r [-i] dir [dir...] dest_dir`	Copy directories

Example:

```
$ ls -F
f1  f2* memo/  note  remind
$ cp f1 f1.copy
$ ls -F
f1  f1.copy f2* memo/ note remind
$ cp note remind memo
$ ls -F memo
note  remind
```

a56653

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.
```

3-12. SLIDE: mv — Move or Rename Files

mv—Move or Rename Files

Syntax:

`mv [-i] file new_file`	Rename a file
`mv [-i] file [file...] dest_dir`	Move files to a directory
`mv [-i] dir [dir...] dest_dir`	Rename or move directories

Example:

```
$ ls -F
f1  f2* memo/ note remind
$ mv f1 file1
$ ls -F
file1  f2* memo/ note remind
$ mv f2 memo/file2
$ ls -F
file1 memo/ note remind
$ ls -F memo
file2*
```

```
$ mv note remind memo
$ ls -F
file1    memo/
$ls -F memo
file2* note remind
$ mv memo letters
$ ls -F
file1    letters/
```

a56651

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.
```

3-13. SLIDE: `ln` — Link Files

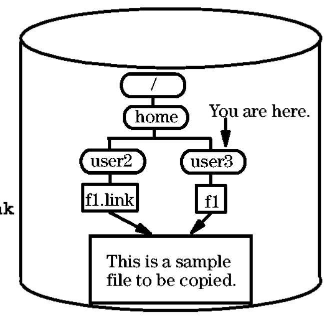
`ln`—Link Files

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
```

3-14. 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**

3-15. SLIDE: File/Directory Manipulation Commands — Summary

File/Directory Manipulation Commands—Summary

`ls -l`	Display file characteristics
`cat`	Concatenate and display contents of files to screen
`more`	Format and display contents of files to screen
`tail`	Display the end of files to screen
`cp`	Copy files or directories
`mv`	Move or rename files or directories
`ln`	Link file names together
`rm`	Remove files or directories
`lp`	Send requests to a line printer
`lpstat`	Print spooler status information
`cancel`	Cancel requests in the line printer queue

a56657

Student Notes

3-16. LAB: File and Directory Manipulation

Directions

Complete the following exercises and answer the associated questions.

File Manipulation

1. 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?

2. 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.

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

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

5. 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**?

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

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.

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.

Module 4 — 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.

4-1. SLIDE: File Permissions and Access

File Permissions and Access

Access to files is dependent on a user's identification and the permissions associated with a file. This module will show how to

Permissions	Understand the read, write, and execute access to a file
`ls (11, ls -l)`	Determine what access is granted on a file
`chmod`	Change the file access
`umask`	Change default file access
`chown`	Change the owner of a file
`chgrp`	Change the group of a file
`su`	Switch your user identifier
`newgrp`	Switch your group identifier

a56659

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.

4-2. SLIDE: Who Has Access to a File?

Who Has Access to a File?

- The UNIX system incorporates a three-tier structure to define who has access to each file and directory:

  `user` The owner of the file
  `group` A group that may have access to the file
  `other` Everyone else

- The `ls -l` command displays the owner and group who has access to the file.

```
$ ls -l
-rw-r--r-- 1  user3  class   37  Jul 24 11:06 f1
-rwxr-xr-x 1  user3  class   37  Jul 24 11:08 f2
drwxr-xr-x 2  user3  class 1024  Jul 24 12:03 memo
                |       |
              owner   group
```

a56660

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

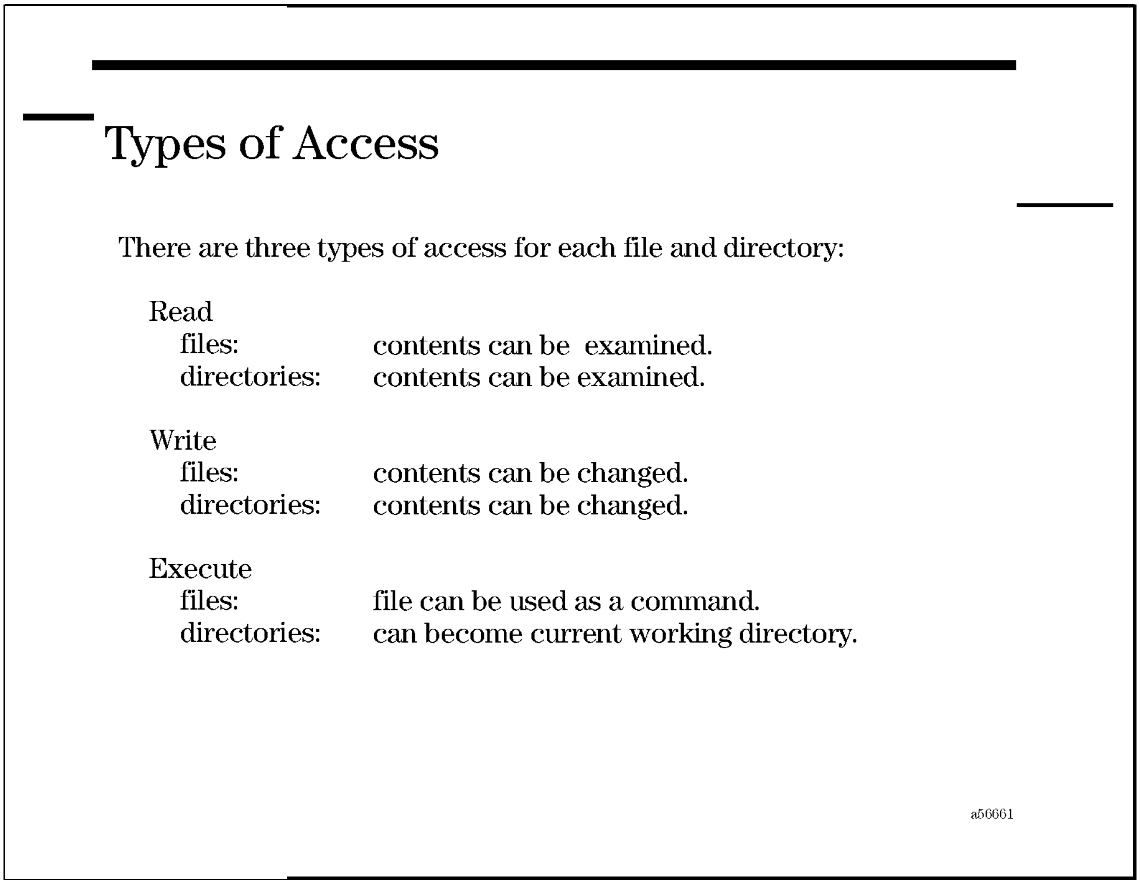
other represents all other users on the system

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.

4-3. SLIDE: Types of Access



Student Notes

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

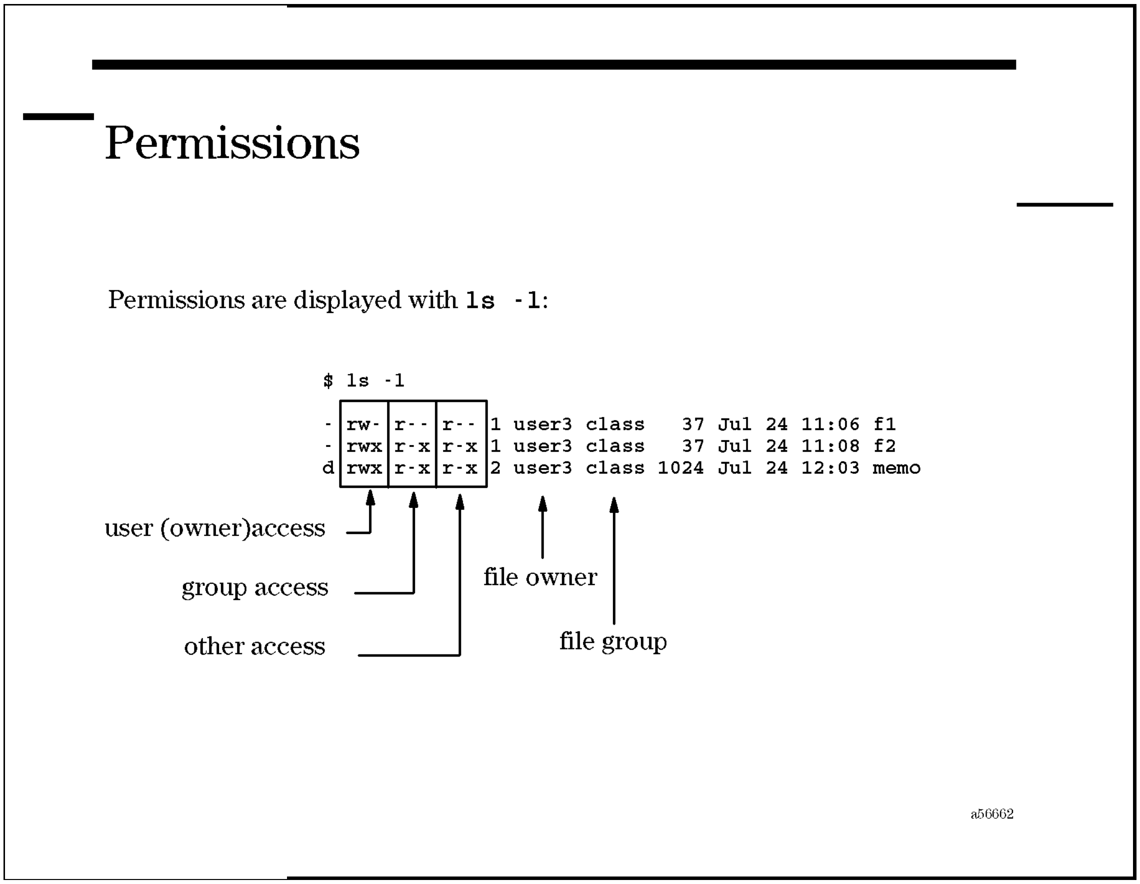
- read
- write
- execute

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.

4-4. 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

4-5. 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:

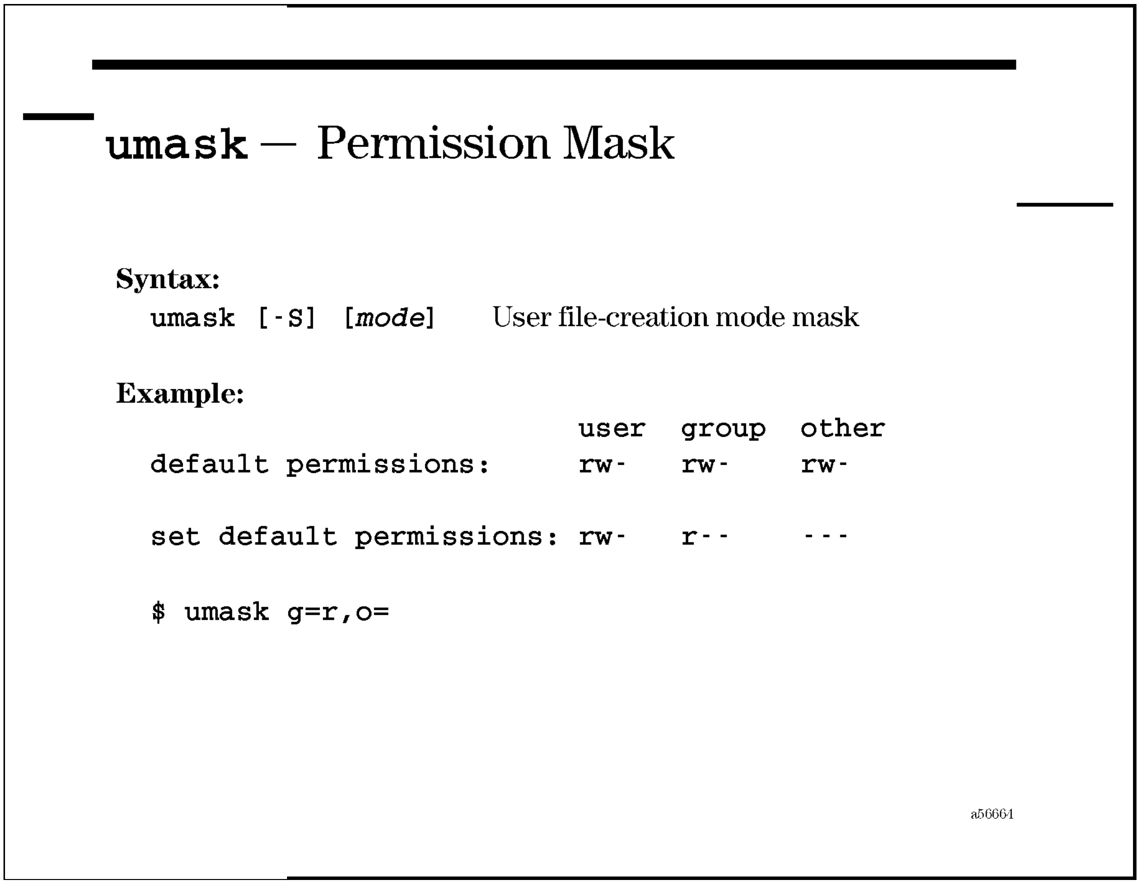
```
user    group   others          convert to numeric values
rw-     r--     ---
4+2+0   4+0+0   0+0+0
6       4       0
```

Thus the **chmod** command would be:

```
chmod 640 filename
```

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

4-6. SLIDE: umask — Permission Mask



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.

4-7. SLIDE: touch — Update Timestamp on File

touch — Update Timestamp on File

Syntax:

`touch [-amc] file...` update access and/or modification times of file

Examples:

```
$ ll
-rw-r--r--  1  karenk  users  25936 Aug 24 09:53 firstfile
-rw-r--r--  1  karenk  users  10245 Aug 24 09:53 secondfile
$ touch newfile
$ ll
-rw-r--r--  1  karenk  users  25936 Aug 24 09:53 firstfile
-rw-r--r--  1  karenk  users      0 Aug 25 10:02 newfile
-rw-r--r--  1  karenk  users  10245 Aug 24 09:53 secondfile
$ touch secondfile
$ ll
-rw-r--r--  1  karenk  users  25936 Aug 24 09:53 firstfile
-rw-r--r--  1  karenk  users      0 Aug 25 10:02 newfile
-rw-r--r--  1  karenk  users  10245 Aug 25 10:05 secondfile
$
```

a6502

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
```

4-8. SLIDE: chown — Change File Ownership

chown — Change File Ownership

Syntax:

```
chown owner [:group] filename ...
```

Changes owner of a file(s) and, optionally, the group ID

Example:

```
$ id
uid=303 (user3),  gid=300 (class)
$ cp f1 /tmp/user2/f1
$ ls -l /tmp/user2/f1
-rw-r----- 1 user3 class 3967 Jan 24 13:13 f1
$ chown user2 /tmp/user2/f1
$ ls -l /tmp/user2/f1
-rw-r----- 1 user2 class 3967 Jan 24 13:13 f1
```

Only the owner of a file (or root) can change the ownership of the file.

a6503

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*.

4-9. SLIDE: The chgrp Command

The chgrp Command

Syntax:

`chgrp newgroup filename ...` Changes group access to a file

Only the owner of a file (or root) can change the group of the file.

Example:

```
$ id
uid=303 (user3),  gid=300 (class)
$ ls -l f3
-rw-r----- 1 user3 class 3967 Jan 24 13:13 f3
$ chgrp class2 f3
$ ls -l f3
-rw-r----- 1 user3 class2 3967 Jan 24 13:13 f3
$ chown user2 f3
$ ls -l f3
-rw-r----- 1 user2 class2 3967 Jan 24 13:13 f3
$
```

a56667

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*.

4-10. SLIDE: su — Switch User Id

su — Switch User Id

Syntax:

su [*user_name*] Change your user ID and group ID designation

Example:

```
$ ls -l /usr/local/bin/class_setup
-rwxr-x--- 1 class_admin teacher 3967 Jan 24 13:13 class_setup
$ id
uid=303 (user3), gid=300 (class)
$ su class_admin
Password:
$ id
uid=400 (class_admin), gid=300 (class)
$ /usr/local/bin/class_setup
$
log out of su session
$ CTRL + D
```

a6281

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**.

4-11. SLIDE: File Permissions and Access — Summary

File Permissions and Access — Summary

Permissions	Define who has what access to a file user, group, others read, write, execute
`chmod`	Change the permissions of a file
`umask`	Define the default permissions for new files
`chown`	Change the owner of a file
`chgrp`	Change the group of a file
`su`	Switch user ID
`newgrp`	Switch group ID

a56671

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.

4-12. LAB: File Permissions and Access

Directions

There are three 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. 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?

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.

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.

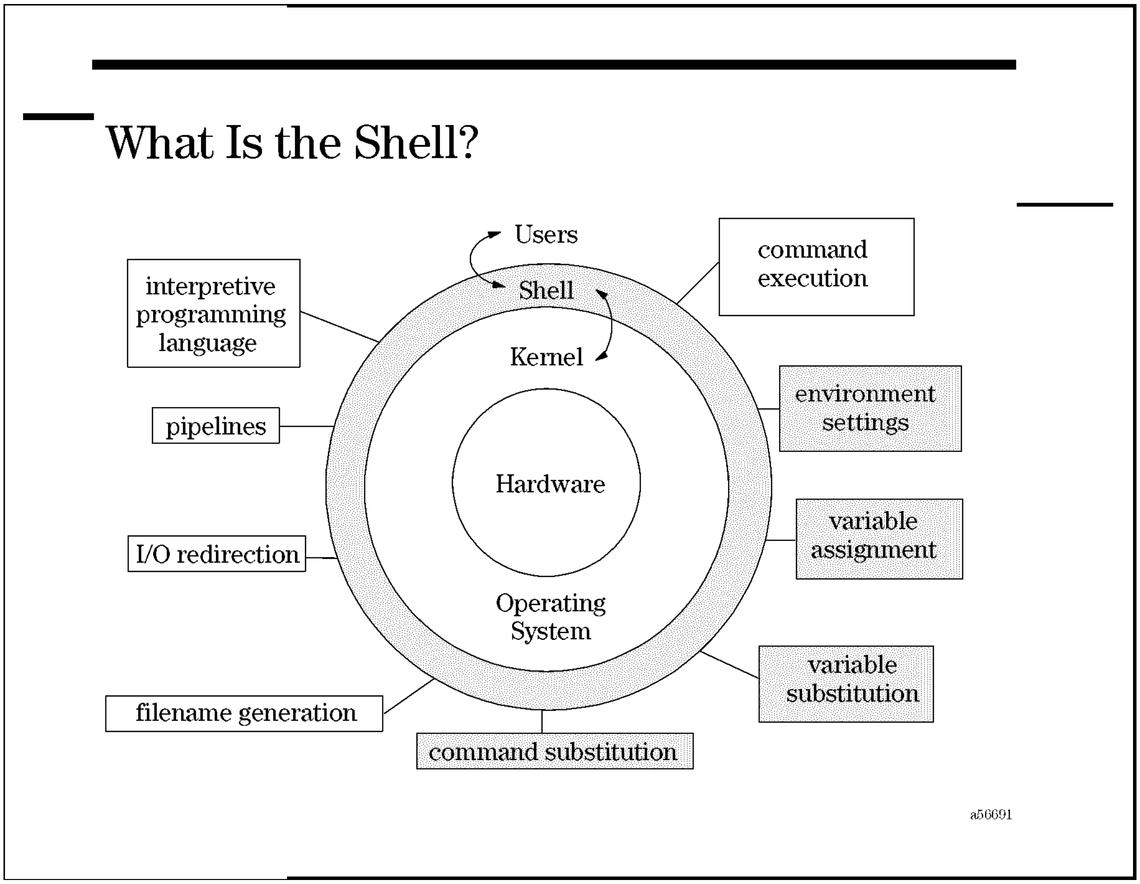
Module 5 — Shell Basics

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.
- Understand and change specific environment variables such as *PATH* and *TERM*.
- Customize the user environment to fit a particular application.

5-1. 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 (%).

5-2. SLIDE: Commonly Used Shells

Commonly Used Shells

`/usr/bin/sh`	POSIX shell
`/usr/bin/ksh`	Korn shell
`/usr/old/bin/sh`	Bourne shell
`/usr/bin/csh`	C Shell
`/usr/bin/keysh`	A context-sensitive softkey shell
`/usr/bin/rksh`	Restricted Korn shell
`/usr/bin/rsh`	Restricted Bourne shell

a56692

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 5-1. 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

5-3. SLIDE: POSIX Shell Features

POSIX Shell Features

- A shell user interface with some advanced features:
  - Command aliasing
  - File name completion
  - Command history mechanism
  - Command line recall and editing
  - Job control
  - Enhanced cd capabilities
  - Advanced programming capabilities

a56693

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

5-4. SLIDE: Aliasing

Aliasing

Syntax:

```
alias [name[=string]]
```

Examples:

```
$ alias dir=ls
$ alias mroe=more
$ alias mstat=/home/tricia/projects/micron/status
$ alias laser="lp -dlaser"
$ laser fileX
request id is laser-234 (1 file)
$ alias          displays aliases currently defined
$ alias mroe     displays value of alias mroe
mroe=more
```

a6507

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.

5-5. SLIDE: File Name Completion

File Name Completion

```
$ more fra ESC ESC
$ more frankenstein Return
   .
   .
   .
$ more abc ESC ESC
$ more abcdef ESC  =

1)abcdefXlmnop
2)abcdefYlmnop

$ more abcdef
Then type X or Y, then ESC ESC.
Associated file name will be completed
```

a6508

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 **vi** 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
```

5-6. SLIDE: Command Line Editing

Command Line Editing

- Provides the ability to modify text entered on current or previous command lines.
- Press Esc to enter *command mode*.
- Recall desired command by either
  - Pressing K until it appears
- Typing the *command number*, then **G**

a6683

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.

5-7. SLIDE: Command Line Editing (continued)

Command Line Editing (continued)

- To position the cursor
  - Use *l*, or [space] key to move right
  - Use *h*, or [backspace] to move left

- *Do Not Use the Arrow Keys!*

- To modify text
  - Use *x* to delete a character
  - Use *i or* a to insert or append characters
  - Press [Esc] to stop adding characters

- Press [return] to execute the modified command

a6686

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.

5-8. SLIDE: The User Environment

The User Environment

- Your environment describes your session to the programs you run.

Syntax:

```
env
```

Example:

```
$ env
HOME=/home/gerry
PWD=/home/gerry/develop/basics
EDITOR=vi
TERM=70092
 ...
PATH=/usr/bin:/usr/contrib/bin:/usr/local/bin:\
/home/gerry/bin
```

a566100

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**.

5-9. 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=$HOME/.kshrc	execute **$HOME/.kshrc** 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

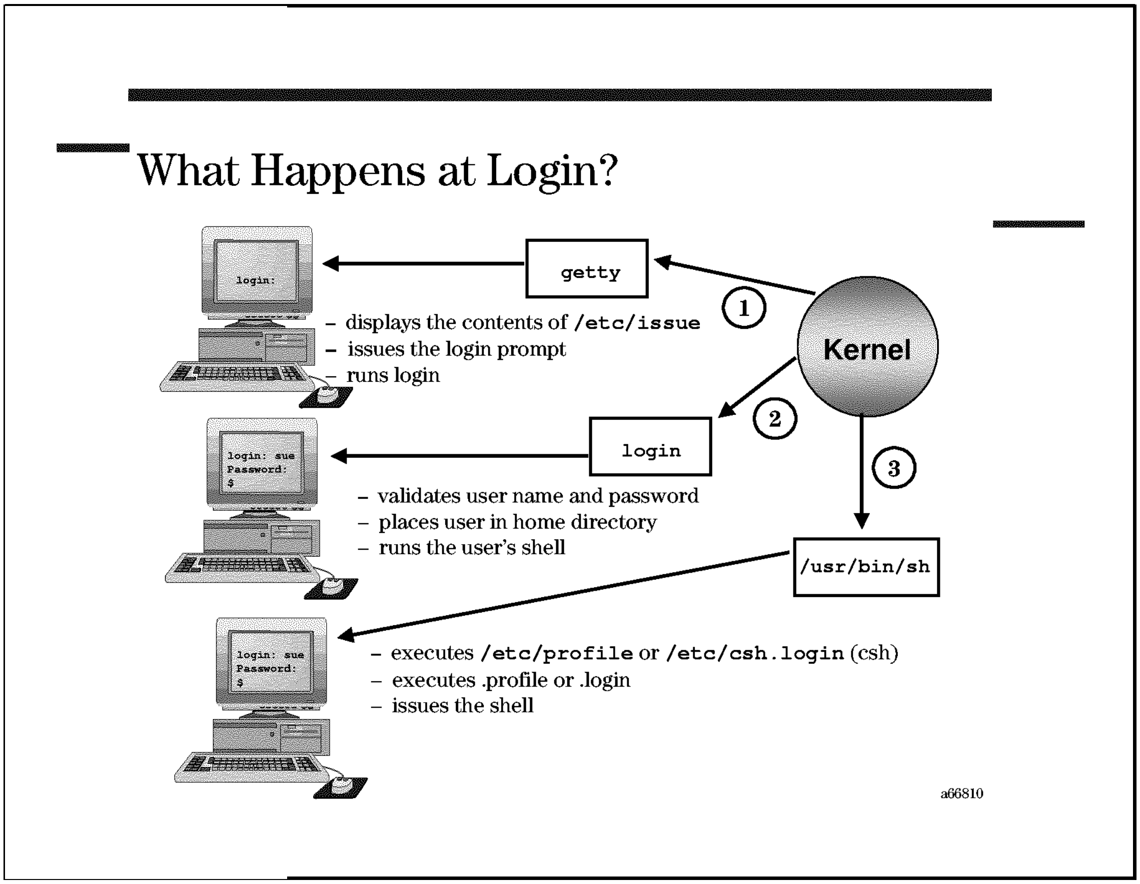
The *TERM* variable must be properly defined so that the UNIX system knows the characteristics of your terminal. Many commands need to know what kind of terminal you are on so that they can properly display their output. For example, **more** and **vi** must know how many lines and columns are on your 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

5-10. 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.

5-11. LAB: Exercises

Directions

Complete the following exercises and answer the associated questions.

1. Set up an alias called **go** to change your working directory to **tree** and do an **ls -F**. Now type 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.)

2. 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**?

3. 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.

Module 6 — Shell Advanced Features

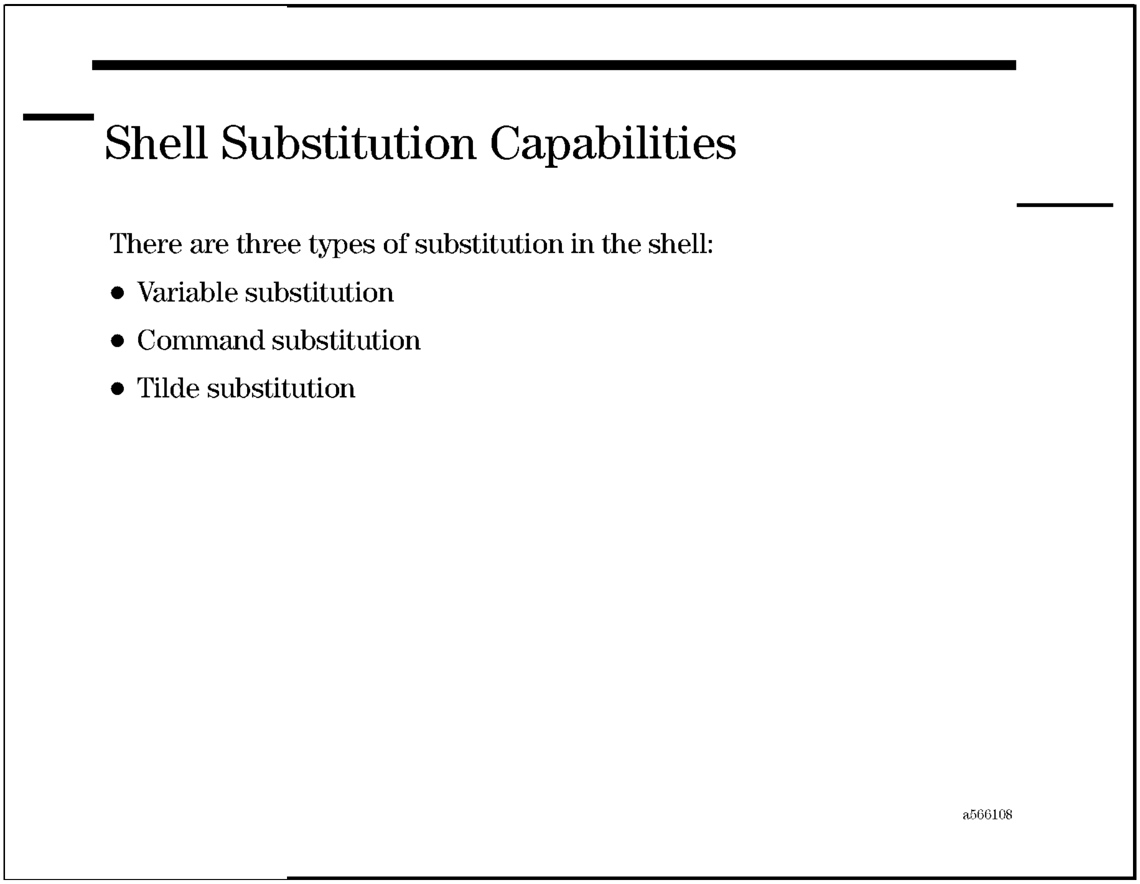
Objectives

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.

51434P G.00

6-1. SLIDE: Shell Substitution Capabilities



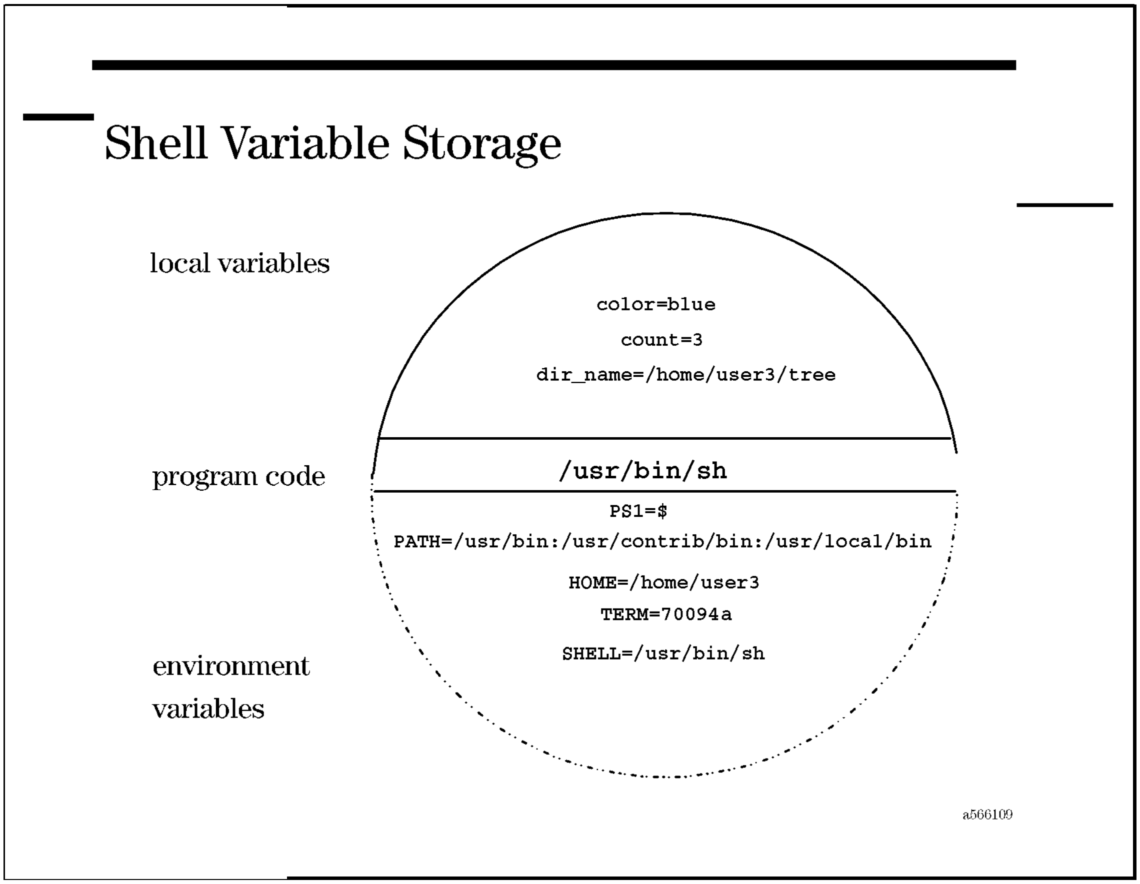
Student Notes

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.

6-2. 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.

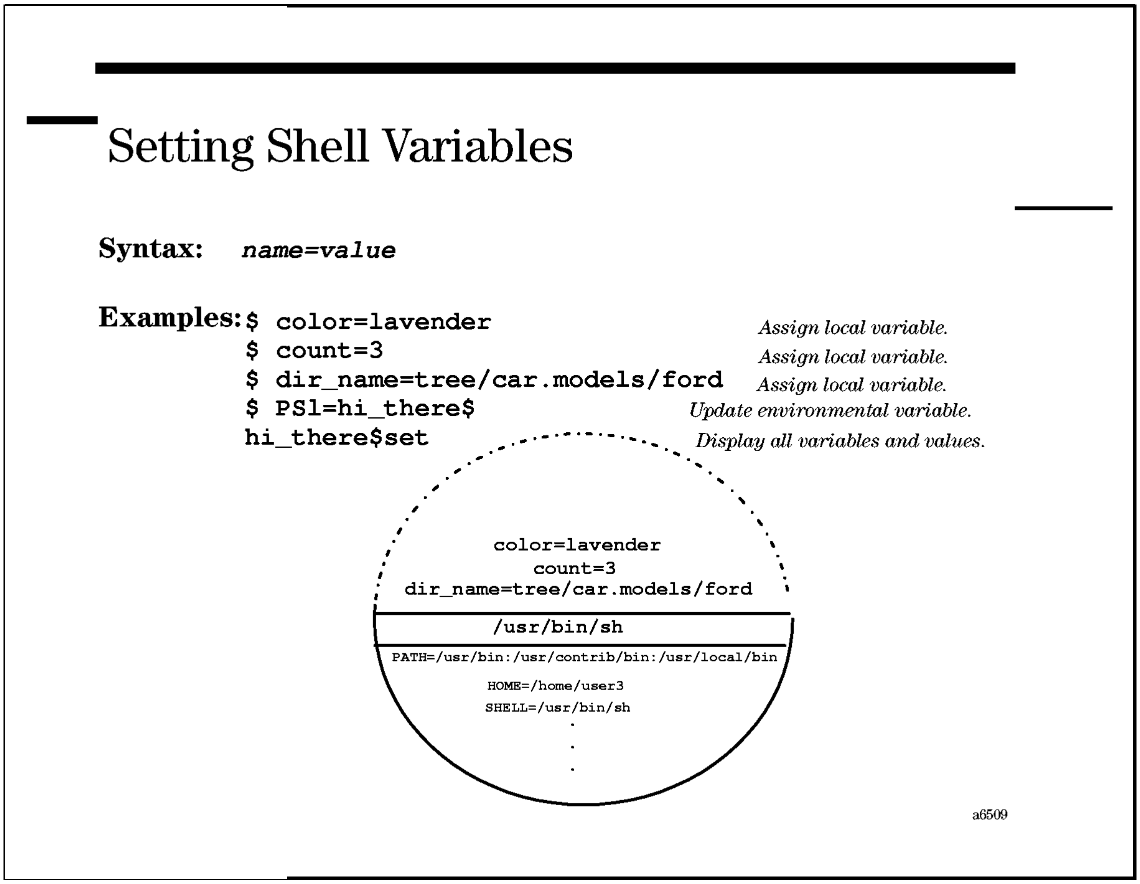
There are several special shell variables that are defined for you through your login process. Many of these variables are stored in the environment; some, such as *PS1* and *PS2*, are usually stored in the local data area. The values of these variables can be changed to customize characteristics of 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
```

6-3. 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.

6-4. SLIDE: Variable Substitution

Variable Substitution

Syntax:

$*name* Directs the ***shell*** to perform variable substitution

Example:

```
$ 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
$ more $file_name
<contents of /home/user3/file1>
```

a566111

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.

6-4. 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`	

6-5. SLIDE: Command Substitution

Command Substitution

Syntax:

$(*command*)

Example:

```
$ pwd
/home/user2
$ curdir=banner $(ls)
$ echo $curdir
/home/user2
$ cd /tmp
$ pwd
/tmp
$ cd $curdir
$ pwd
/home/user2
```

a65010

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**.

6-6. SLIDE: Tilde Substitution

Tilde Substitution

```
$ echo $HOME
HOME=/home/user3
$ echo ~
/home/user3

$ echo $PWD
PWD=/home/user3/tree
$ ls ~+/poodle
/home/user3/tree/dog.breeds

$ echo $OLDPWD
/home/user3/mail
$ ls ~-
/home/user3/mail/from.mike /home/user3/mail/from.jim

$ echo ~tricia/file1
/home/tricia/file1
```

a65011

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
```

6-7. SLIDE: Displaying Variable Values

Displaying Variable Values

```
$ echo $HOME
/home/user3

$ env
HOME=/home/user3
PATH=/usr/bin:/usr/contrib/bin:/usr/local/bin
SHELL=/usr/bin/sh

$ set
HOME=/home/user3
PATH=/usr/bin:/usr/contrib/bin:/usr/local/bin
SHELL=/usr/bin/sh
color=lavender
count=3
dir_name=/home/user3/tree

$ unset dir_name
```

a566115

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.

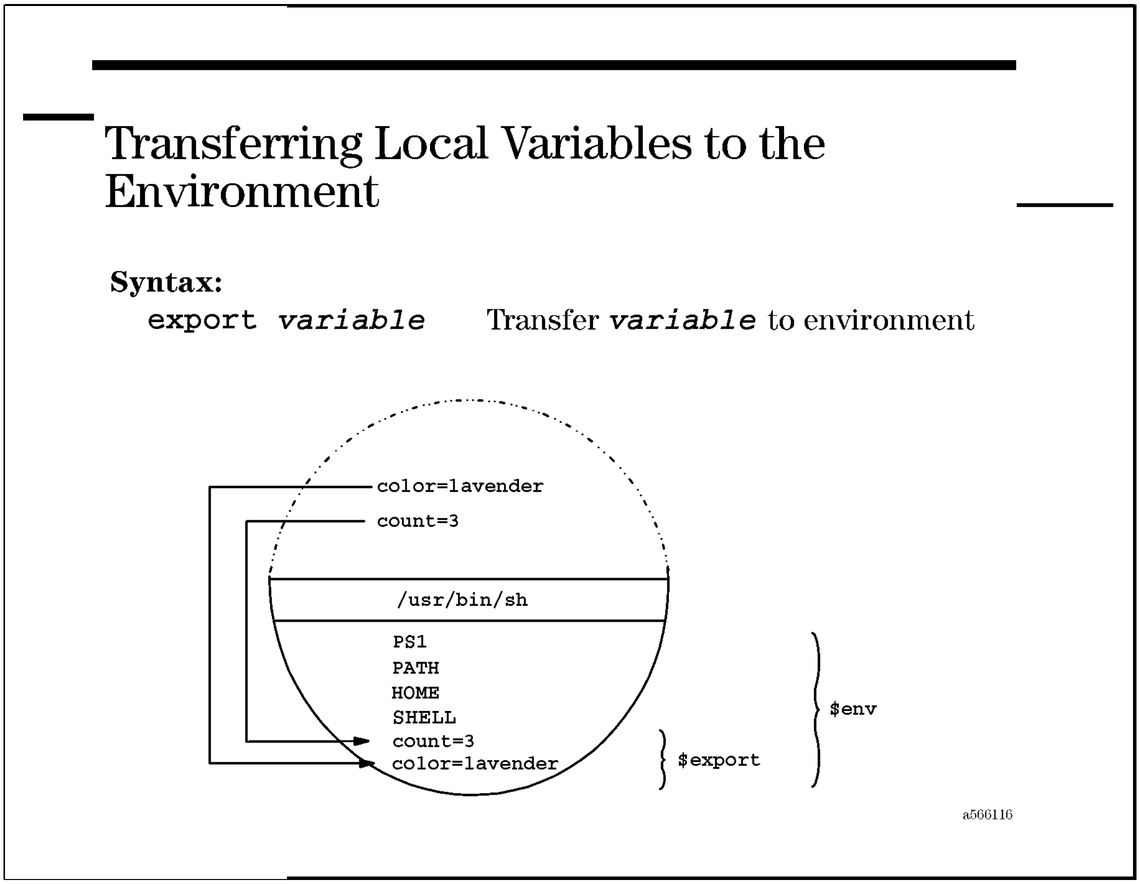
The **env** command can be used to display *all* of the variables that are currently held in the environment.

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**.

6-8. SLIDE: Transferring Local Variables to the Environment



Student Notes

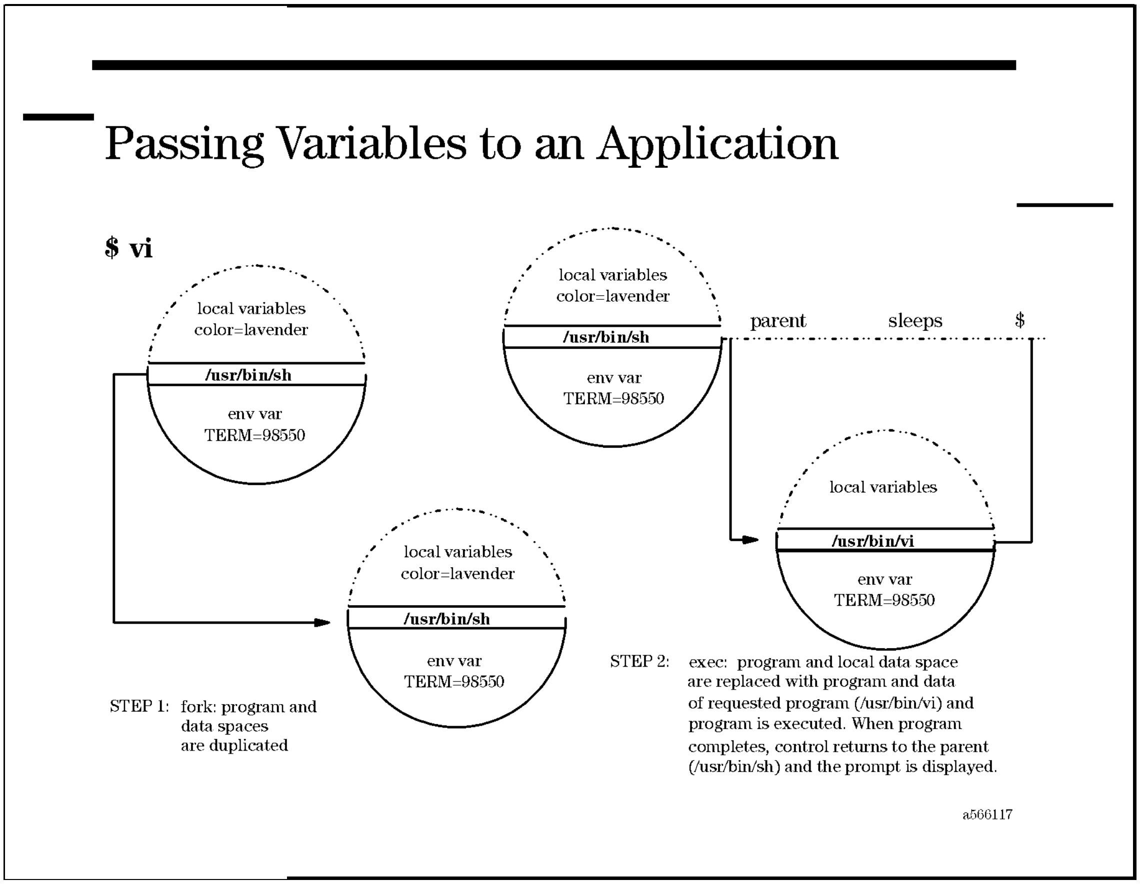
The diagram on the slide illustrates transferring the variables *color* and *count* into the environment by executing the following commands:

```
$ color=lavender
$ export color
$ export count=3
$ export
export PATH=/usr/bin:/usr/ccs/bin:/usr/contrib/bin:/usr/local/bin
export color=lavender
export count=3
```

In order for a variable to be available to other processes, it must exist in the environment. When a variable is defined, it is stored in the local data space and must be **exported** to the environment.

The **`export`** *variable* command will transfer the specified variable from the local data space to the environment data space. **`export`** *variable=value* will assign (possibly update) the value of a variable, and place it in the environment. With no arguments, the **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.

6-9. 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.

6-10. SLIDE: Monitoring Processes

Monitoring Processes

```
$ ps -f
   UID  PID  PPID    C    STIME   TTY     TIME COMMAND
 user3 4702     1    1   08:46:40 ttyp4   0:00 -sh
 user3 4895  4702   18   09:55:10 ttyp4   0:00 ps -f

$ ksh
$ ps -f
   UID  PID  PPID    C    STIME   TTY     TIME COMMAND
 user3 4702     1    0   08:46:40 ttyp4   0:00 -sh
 user3 4896  4702    1   09:57:20 ttyp4   0:00 ksh
 user3 4898  4896   18   09:57:26 ttyp4   0:00 ps -f

$ exec ps -f
   UID  PID  PPID    C    STIME   TTY     TIME COMMAND
 user3 4702     1    0   08:46:40 ttyp4   0:00 -sh
 user3 4896  4702   18   09:57:26 ttyp4   0:00 ps -f
$
```

a566118

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:

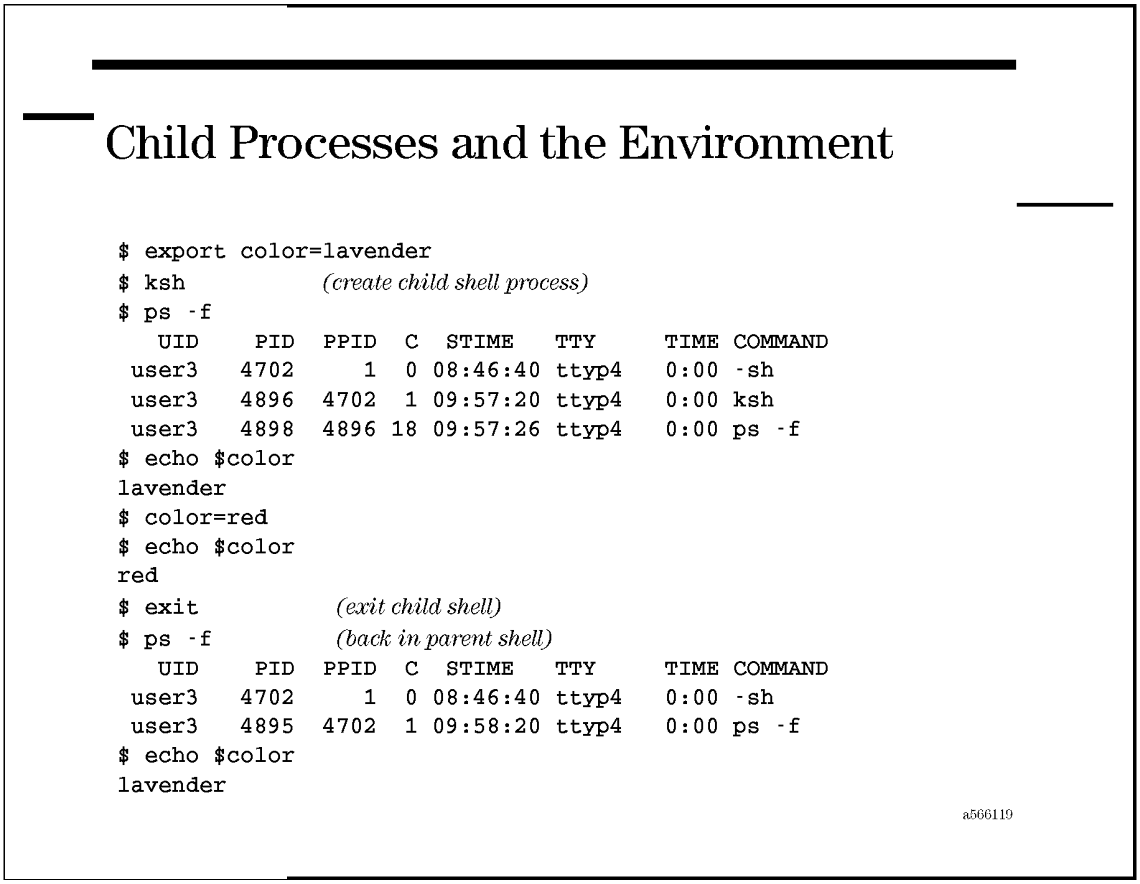
```
$ ps
PID TTY TIME COMMAND
4702 ttyp4 0:00 sh
4894 ttyp4 0:00 ps
```

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.

6-11. 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*.

6-12. 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. Set a shell variable named *MYNAME* equal to your first name. How do you see the contents of that variable?

3. Now start a child shell by typing **sh**. Look at the contents of *MYNAME* now. What happened? Exit the child shell (use Ctrl+c Return or **exit**). Does the parent still know about the variable *MYNAME*?

4. What command can be typed 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?

5. 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?

6. 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?

7. 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?

8. Modify your shell prompt so that it displays: *good_day$*. What happens to your prompt when you log out and log back in?

9. 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?)

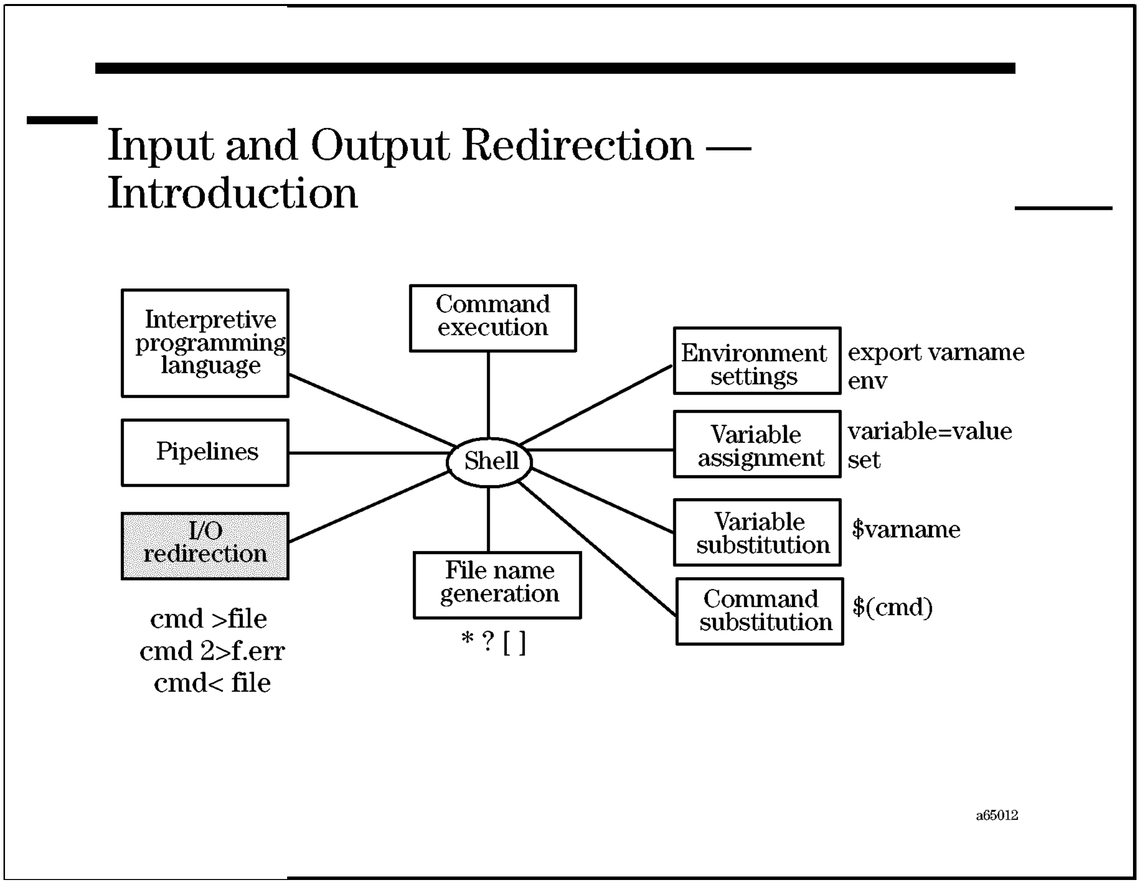
Module 7 — Input and Output Redirection

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**.

7-1. SLIDE: Input and Output Redirection — Introduction



Student Notes

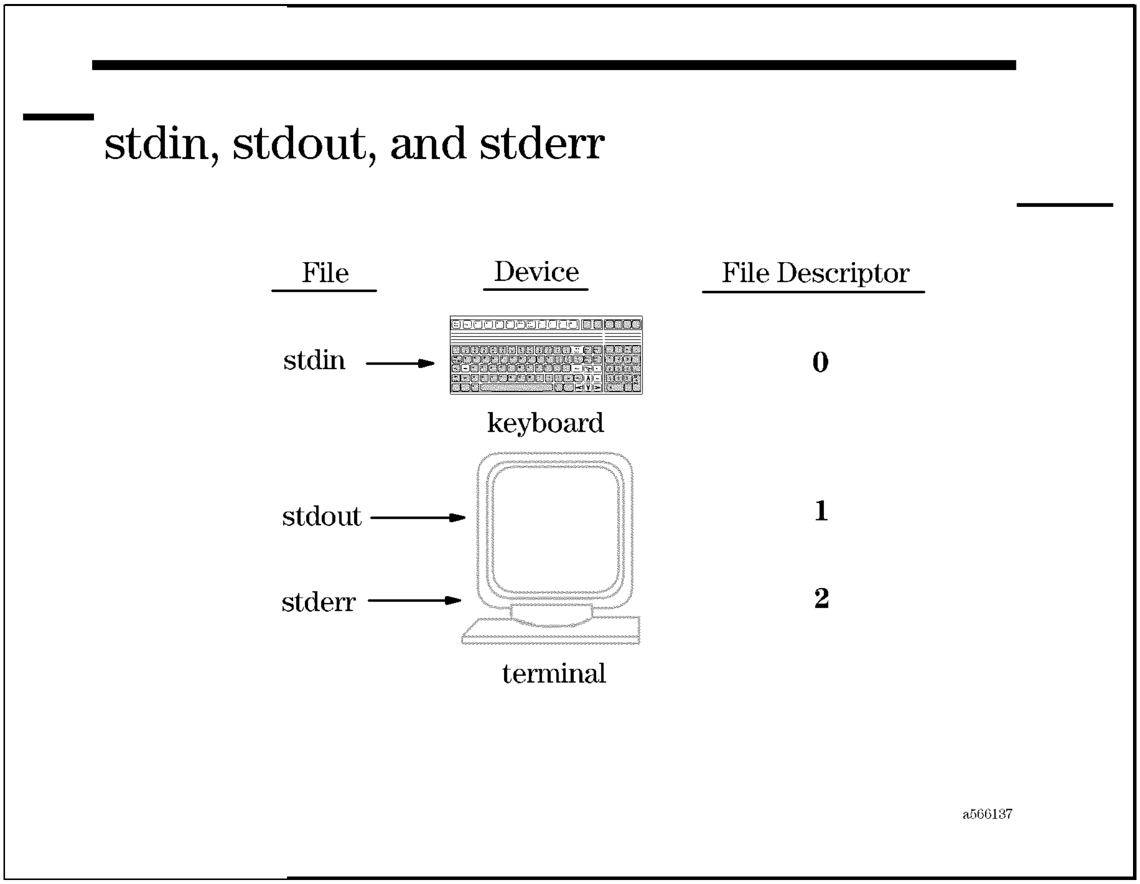
Another feature that the shell provides is the capability to redirect the input or output of a command. Most commands send their output to your terminal; examples include **`date`**, **`banner`**, **`ls`**, **`who`**, etc. Other commands get input from your keyboard; examples include **`mail`**, **`write`**, **`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.

7-2. 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.

7-3. SLIDE: Input Redirection — <

Input Redirection — <

Any command that reads its input from stdin can have its input redirected to come from another file.

Example:

```
$ cat remind
Your mother's birthday is November 29
$ mail user3 < remind
$ mail
From user3 Mon July 15 11:30 EDT 1993
Your mother's birthday is November 29
?d
$
```

a566138

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.

7-4. SLIDE: Output Redirection — > and >>

Output Redirection — > and >>

Any command that produces output to stdout can have its output redirected to another file.

Examples:

Create/Overwrite	**Create/Append**
`$ date > date.out`	`$ ls >> ls.out`
`$ date > who.log`	`$ who >> who.log`
`$ cat > cat.out`	`$ ls >> who.log`
input text here	
Ctrl + d	

a566139

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.

7-5. SLIDE: Error Redirection — 2> and 2>>

Error Redirection — 2> and 2>>

Any command that produces error messages to stderr can have those messages redirected to another file.

Examples:

```
$ cp 2> cp.err       Create/Overwrite
$ cp 2>> cp.err      Create/Append
$

$ more cp.err
Usage: cp [-f|-i] [-p] source_file target_file
       cp [-f|-i] [-p] source_file ...target_directory
       cp [-f|-i] [-p] -R|-rsource_directory...target_directory
Usage: cp [-f|-i] [-p] source_file target_file
       cp [-f|-i] [-p] source_file ... target_directory
       cp [-f|-i] [-p] -R|-r source_directory...target_directory
```

a566140

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.

7-6. SLIDE: What Is a Filter?

What Is a Filter?

- Reads standard input *and* produces standard output.
- Filters the contents of the input stream or a file.
- Sends results to screen, never modifies the input stream or file.
- Processes the output of other commands when they are used in conjunction with output redirection.

Examples: `cat`, `grep`, `sort`, `wc`

a566141

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.

7-7. SLIDE: `wc` — Word Count

`wc` — Word Count

Syntax:

`wc [-lwc] [file...]` Counts lines, words, and characters in a file

Examples:

```
$ wc funfile     funfile provided as a command line argument
116 529 3134 funfile
$
$ wc -l funfile
116 funfile
$
$ ls > ls.out
$
$ wc -w ls.out   count the number of entries in your directory
72 ls.out
```

a566142

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.

7-8. SLIDE: sort — Alphabetical or Numerical Sort

sort — Alphabetical or Numerical Sort

Syntax:

sort [-ndutX] [-k *field_no*] [*file*...] Sorts lines

Examples:

```
$ sort funfile              funfile provided as a command line argument

$ tail -1 /etc/passwd
user3:xyzbkd:303:30:studentuser3:/home/user3:/usr/bin/sh
 1     2      3  4      5            6           7
$ sort -nt: -k 3 < /etc/passwd

$ who > whoson
$ sort whoson                       sort logged in users alphabetically
$ sort -u -k 1,1 whoson             sort and suppress duplicate lines
```

a65013

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**.

7-9. SLIDE: grep — Pattern Matching

grep — Pattern Matching

Syntax:

```
grep [-cinv] [-e] pattern [-e pattern] [file...]
grep [-cinv] -f  patterns_list_file [file...]
```

Examples:

```
$ grep user /etc/passwd
$ grep -v user /etc/passwd
$ grep -in -e like -e love funfile

$ who > whoson
$ vi whoson
:1,$s/ .*//g
ZZ
$ grep -f whoson /etc/passwd
```

A6286

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.

7-10. SLIDE: Input and Output Redirection — Summary

Input and Output Redirection — Summary

`cmd < file`	Redirects input to `cmd` from *file*
`cmd > file`	Redirects standard output from `cmd` to *file*
`cmd >> file`	Redirects standard output from `cmd` and append to *file*
`cmd 2> file.err`	Redirects errors from `cmd` to *file*.err
A filter	A command that accepts stdin and generates stdout
`wc`	Line, word, and character count
`sort`	Sorts lines alphabetically or numerically
`grep`	Searches for lines that contain a pattern

a566145

Student Notes

7-11. LAB: Input and Output Redirection

Directions

Complete the following exercises and answer the associated questions.

1. Create two very short files called **f1** and **f2** using **cat** and output redirection.

2. 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?

3. 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?

4. 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.

5. 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.

6. 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?

7. Sort the file **/etc/passwd** on the third field. What happens? Now do a numeric sort on the third field. Any difference?

8. 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*.

9. Using redirection and filters, how many users are logged in on the system?

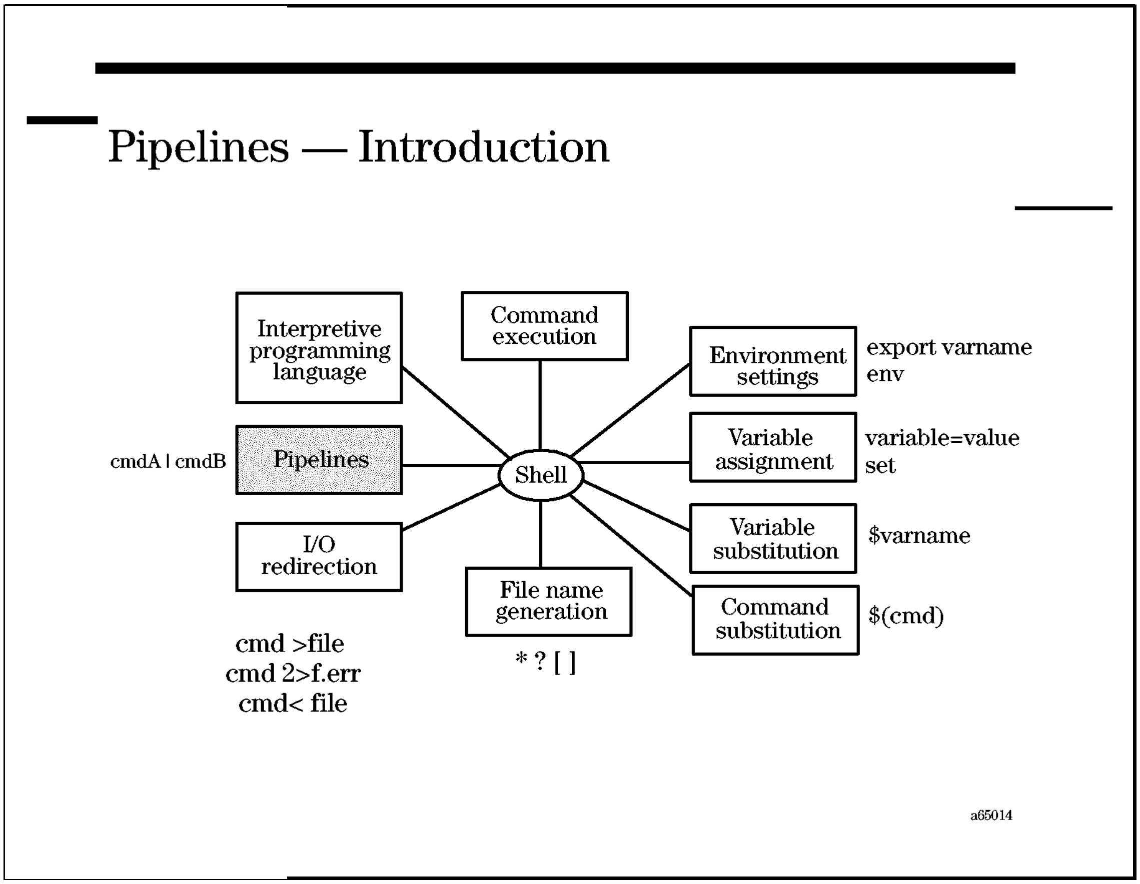
Module 8 — Pipes

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.

8-1. SLIDE: Pipelines — Introduction



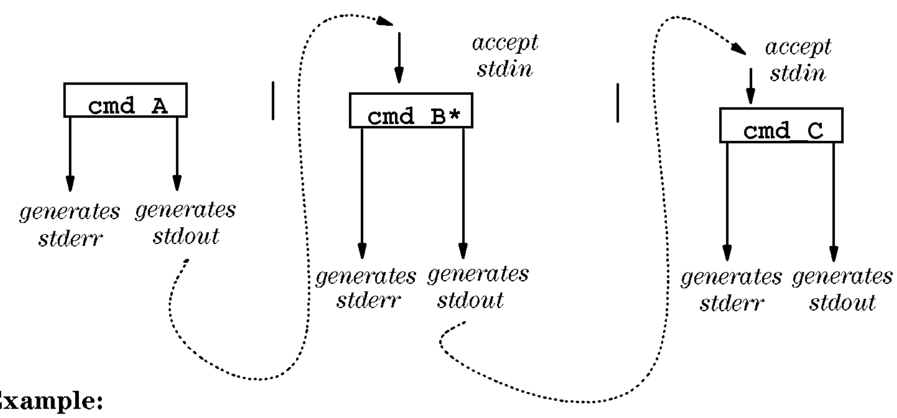
Student Notes

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.

8-2. SLIDE: The | Symbol

The | Symbol



Example:

```
$ ps -ef | more
$ ls | more
$ ls | sort -r | more
```

*`cmd_B` must be a filter.

a566149

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.

8-3. SLIDE: Pipelines versus Input and Output Redirection

Pipelines versus Input and Output Redirection

Input and Output Redirection	Pipelines
Syntax:	
cmd_out > *file*	**cmd_out | cmd_in**
or	
cmd_in < *file*	
Example:	
who > who.out	**who | sort**
sort < who.out	

a566150

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.

8-4. SLIDE: Some Filters

Some Filters

`cut`	Cuts out specified columns or fields and display to `stdout`
`tr`	Translates characters
`tee`	Passes output to a file *and* to `stdout`
`pr`	Prints and format output to `stdout`

a566152

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.

8-5. SLIDE: The cut Command

The **cut** Command

Syntax:

cut ***-clist*** **[*file*...]**	Cuts columns or fields
cut ***-flist*** **[*-dchar*][-s][*file*...]**	from files or **stdin**

Examples:

```
$ date | cut -c1-3
$ tail -1 /etc/passwd
user3:mdhbmkdj:303:30:student user3:/home/user3:/usr/bin/sh
  1      2      3  4        5          6           7
$ cut -f1,6 -d: /etc/passwd
$ cut -f1,6 -d: /etc/passwd | sort -r
$ ps -ef | cut -c49- | sort -d
```

a65015

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
```

8-6. SLIDE: The `tr` Command

The **tr** Command

Syntax:

tr [-s] [*string1* [*string2*]] Translates characters

Examples:

```
$ who | tr -s " "
$
$ date | cut -c1-3 | tr "[:lower:]" "[:upper:]"
```

A6287

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.

8-7. SLIDE: The tee Command

The tee Command

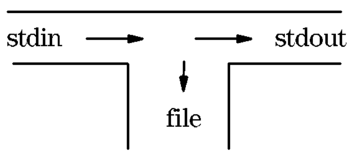
Syntax:

```
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

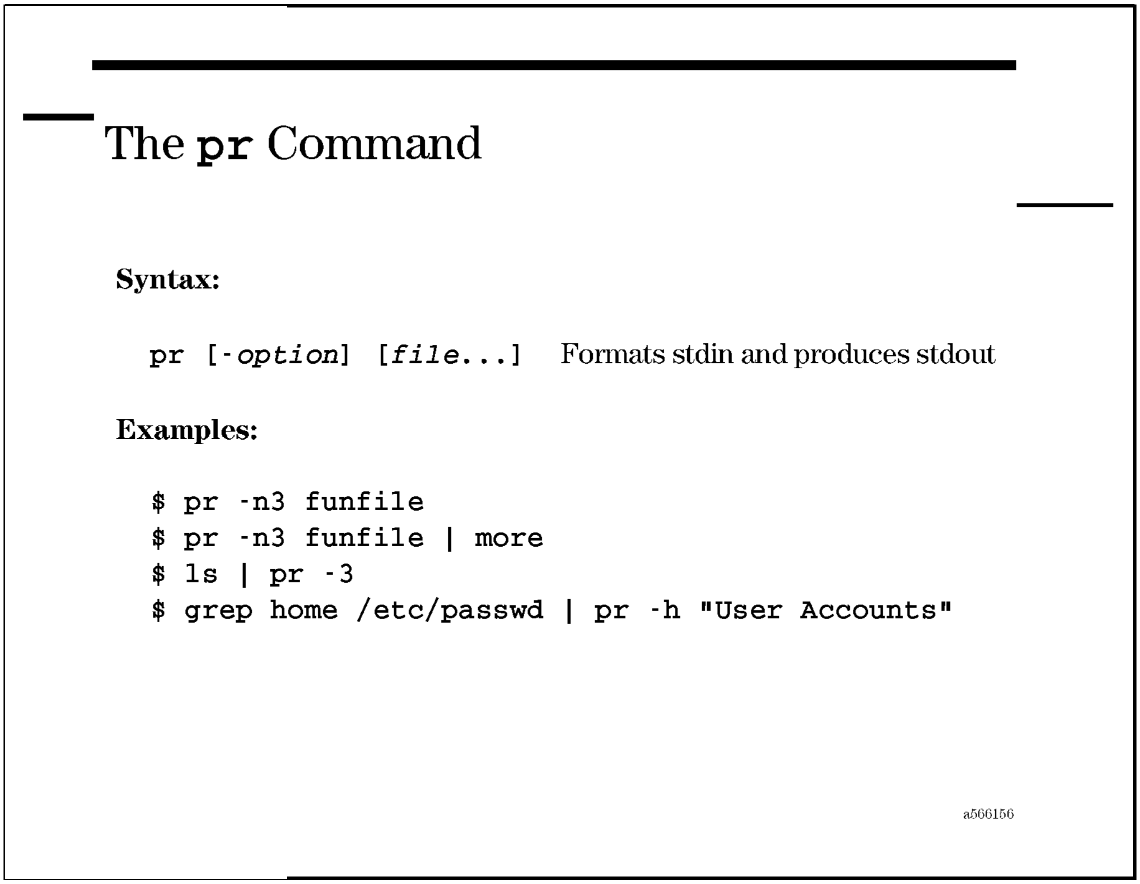
Generally, when you are executing a complex pipeline, the output of the intermediate commands is submitted to the next command in the pipe and you will not be able to view the intermediate output. The **tee** command is used to tap a pipeline. **Tee** reads from standard input and writes its output to 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.

8-8. 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

8-9. SLIDE: Printing from a Pipeline

Printing from a Pipeline

`... | lp` Located at end of pipe; sends output to printer

Examples:

```
$ pr -l58 funfile | lp
Request id is laser-226 (standard input).
$
$ ls -F $HOME | pr -3 | tee homedir | lp
Request id is laser-227 (standard input).
$
$ grep home /etc/passwd | pr -h "user accounts" | lp
Request id is laser-228 (standard input).
```

a566157

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.

8-10. SLIDE: Pipelines — Summary

Pipelines — Summary

Pipeline	`cmd_out	cmd_in` `cmd_out	cmd_in_out	cmd_in`
`cut`	Cuts out columns or fields to standard output
`tee`	Sends input to standard output and a specified file
`pr`	Prints formatter to the screen, commonly used with `lp`
`tr`	Translates characters

a566158

Student Notes

8-11. LAB: Pipelines

Directions

Complete the following exercises and answer the associated questions.

1. 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.

2. 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.

3. Create an alias called **`whoson`** that will display an alphabetical listing of the users currently logged into your system.

4. 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.

5. 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. Second, use the same pipeline but now output the list in numerical order by user ID number. Hint: the information can be found in **`/etc/passwd`**.

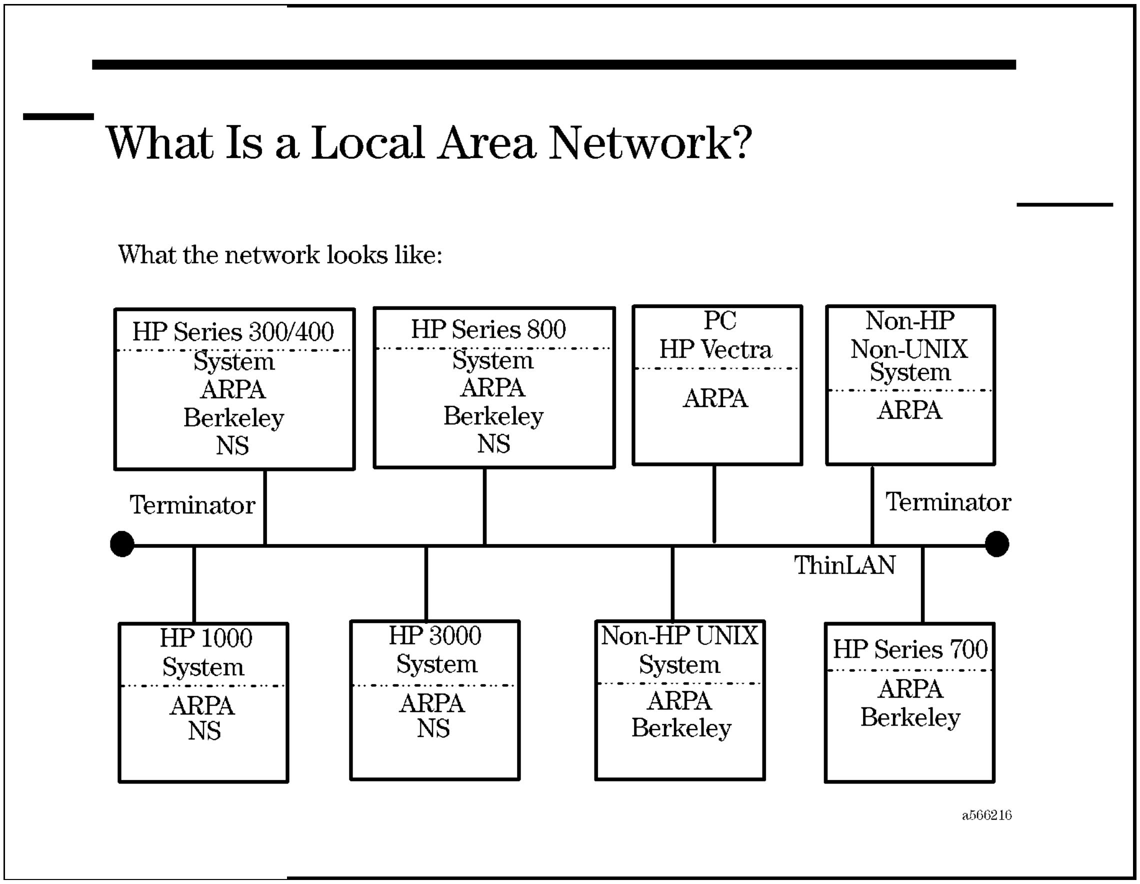
Module 9 — Using Network Services

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.

9-1. SLIDE: What Is a Local Area Network?



Student Notes

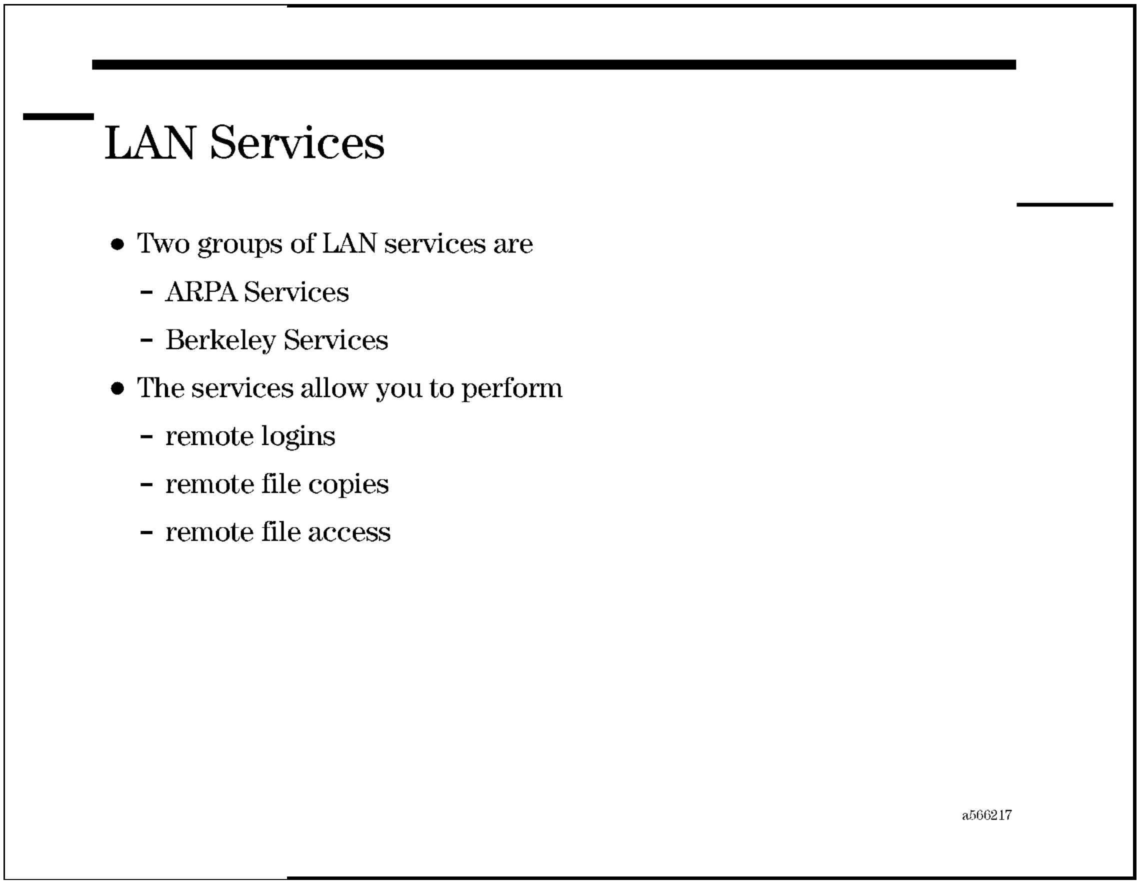
A **Local Area Network** (LAN) is a method of connecting two or more computer systems over a small area. Most installations that have more than one computer will install a LAN to allow the users to work on several different computers without physically picking up all of their work and moving to the computer 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.

9-2. 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.

9-3. SLIDE: The hostname Command

The **hostname** Command

Syntax:

`hostname` Reports your computer's network name

Example:

```
$ hostname
fred
$
$ more /etc/hosts
192.1.2.1    fred
192.1.2.2    barney
192.1.2.3    wilma
192.1.2.4    betty
```

a566218

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.

9-4. SLIDE: The `telnet` Command

The `telnet` Command

Syntax:

`telnet` *`hostname`* ARPA Service to remotely log in to another computer

Example:

```
$ telnet fred
Trying ...
Connected to fred.
Escape character is '^]'.

HP-UX fred 10.0  9000/715
login:
```

a566219

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] [Return] or **`exit`**.

9-5. SLIDE: The `ftp` Command

The `ftp` Command

Syntax:

`ftp` *`hostname`*	ARPA Service to copy files to and from a remote computer

`ftp` **Commands:**

`get`	Gets a file from the remote computer
`put`	Sends a local file to the remote computer
`ls`	Lists files on the remote computer
`?`	Lists all `ftp` commands
`quit`	Leaves `ftp`

a566220

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 /tmp
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls /tmp.
-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
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!)

9-6. SLIDE: The `rlogin` Command

The **rlogin** Command

Syntax:

`rlogin` *hostname*	Berkeley Service to remotely log in to another computer; `rlogin` attempts to log you in using local user name

Example:

```
$ hostname
barney
$ rlogin fred
Password:
$ hostname
fred
$ exit
$ hostname
barney
```

a566221

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.

9-7. SLIDE: The rcp Command

The `rcp` Command

Syntax:

rcp ***source_pathname target_pathname*** — Berkeley Service to copy files to and from a remote computer; works just like the **cp** command

Remote file names are specified as ***hostname*:*pathname***

Example:

```
$ rcp funfile fred:/tmp/funfile
$
```

a65019

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.

9-8. SLIDE: The remsh Command

The **remsh** Command

Syntax:

`remsh` *`hostname command`* — Berkeley Service to run a command on a remote computer

Example:

```
$ hostname
barney
$ remsh fred ls /tmp
backuplist
croutOqD00076
fred.log
Update.log
$
EX000662    tmpfile   Update.log
$
```

a6687

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
```


51434P G.00

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.

9-9. SLIDE: Berkeley — The **rwho** Command

The **rwho** Command

- **rwho** produces output similar to **who**.
- **rwho** displays users on machines in LAN running **rwho** daemon.

Example:

```
$ rwho
user1    barney:tty0p1 Jul 18   8:23   :10
user2    wilma:tty0p1  Jul 18  10:13   :03
user3    fred:tty0p1   Jul 18  11:32   :06
```

a566221

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.

9-10. SLIDE: Berkeley — The `ruptime` Command

Berkeley—The `ruptime` Command

- `ruptime` displays the status of each machine in the LAN.
- Each system must be running the `rwho` daemon.

Example:

```
$ ruptime
barney  up      3:10  1 users load 1.32, 0.80, 0.30
fred    up    1+5:15  4 users load 1.47, 1.16, 0.80
wilma   down    0:00
```

a566225

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.

9-11. 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.

Module 10 — 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.

10-1. SLIDE: The ps Command

Student Notes

Every process that is initiated on the system is assigned a unique identification number, known as a process ID (**PID**). The **ps** command displays information about processes currently running (or sleeping) on your system, including the PID of each process and the PID of each process's parent (**PPID**). Through the PID and PPID numbers, you can trace the lineage of any process that is running on your system. The **ps** command will also report who owns each process and which terminal each process is executing through.

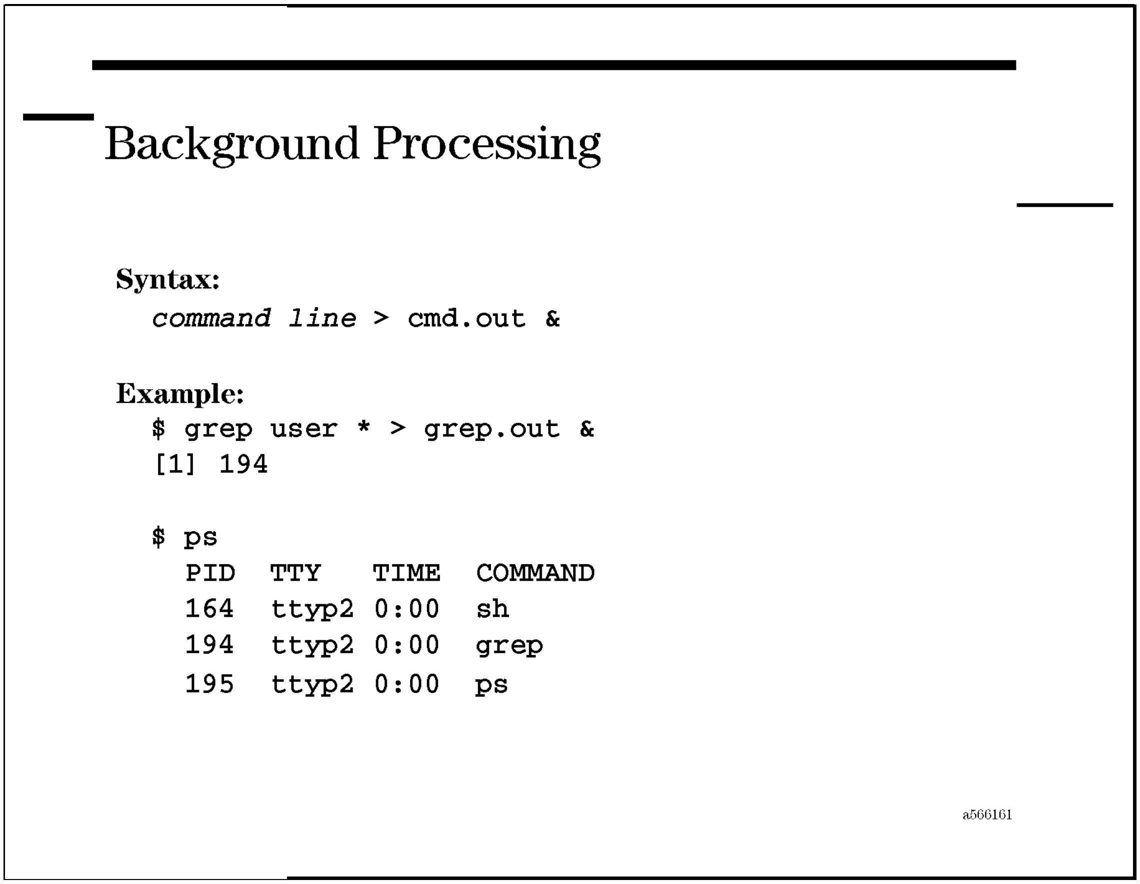
The **ps** command is commonly invoked with no options, which gives a short report about processes associated only with your terminal session. The **-e** option reports about **e**very process running on the system, not just your own. The **-f** and **-l** options report **f**ull and **l**ong listings 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.

10-2. SLIDE: Background Processing



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.

10-3. SLIDE: Putting Jobs in Background/Foreground

Putting Jobs in Background/Foreground

`jobs`	Displays jobs currently running
Ctrl + z	Suspends a job running in the foreground `stty susp ^Z`
`fg [%`*`number`*`]`	Brings job number to the foreground or
`fg [%`*`string`*`]`	any job whose command line begins with *`string`*.
`bg [%`*`number`*`]`	Transfers job number to the background or
`bg [%`*`string`*`]`	any job whose command line begins with *`string`*.

a566162

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.

10-4. 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  Return
login: user3
Password:
  .
  .
  .
$ ps -ef | grep cat
UID     PID   PPID          COMMAND
user3   972     1    ....   cat * > bigfile &
```

a566163

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**.

10-5. 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.

10-6. 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 of 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
```

10-7. 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.

Module 11 — 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.

11-1. SLIDE: Shell Programming Overview

Shell Programming Overview

- A shell program is a regular file containing UNIX system commands.
- The file's permissions must be at least "read" and "execute."
- To execute, type the name of the file at the shell prompt.
- Data can be passed into a shell program through
  - environment variables
  - command line arguments
  - user input

a566167

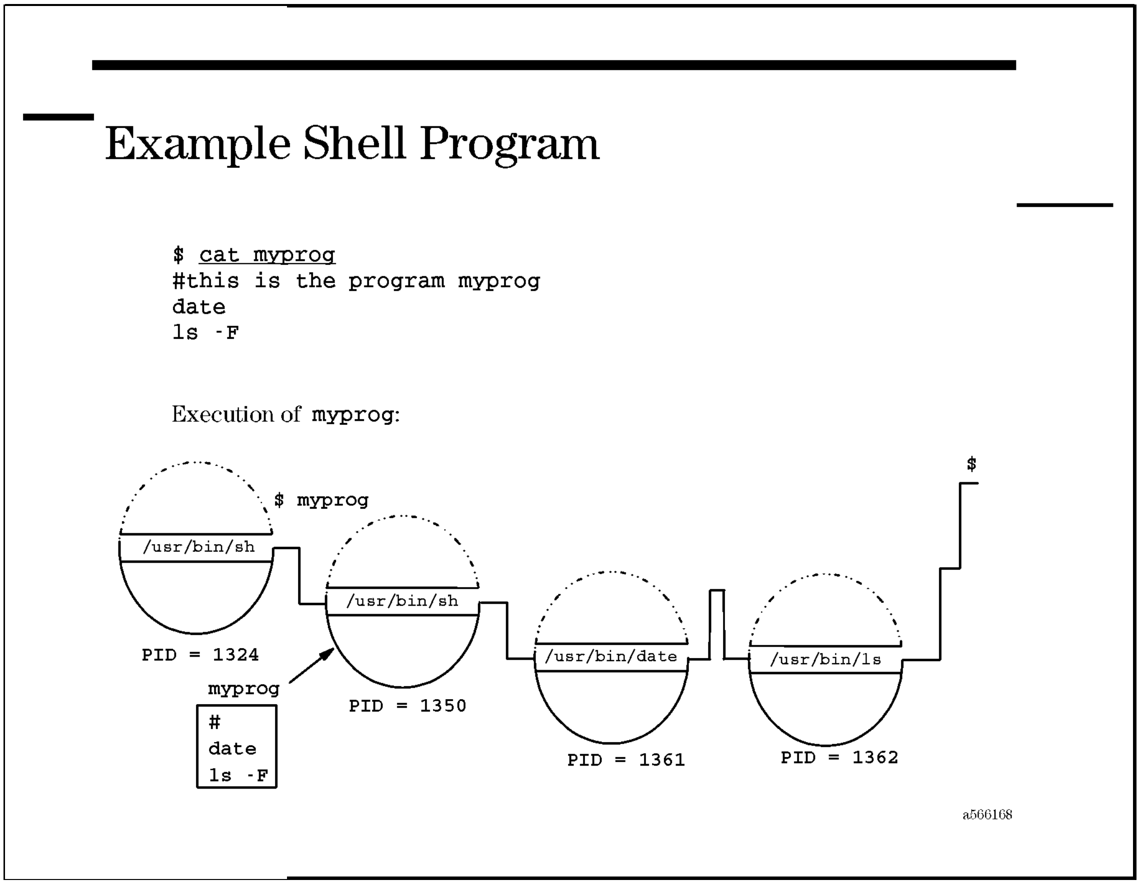
Student Notes

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.

11-2. 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.

11-3. SLIDE: Passing Data to a Shell Program

Passing Data to a Shell Program

```
$ color=lavender

$ cat color1
echo You are now running program: color1
echo the value of the variable color is: $color

$ chmod +x color1

$ color1
You are now running program: color1
the value of the variable color is:

$ export color
$ color1
You are now running program: color1
the value of the variable color is: lavender
```

a566169

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
```

11-4. SLIDE: Arguments to Shell Programms

Arguments to Shell Programs

Command line:

```
  $ sh_program arg1 arg2 . . . argX
        $0       $1   $2  . . .  $X
```

Example:

```
$ cat color3
echo You are now running program: $0
echo The value of command line argument \#1 is: $1
echo The value of command line argument \#2 is: $2

$ chmod +x color3

$ color3 red green
You are now running program: color3
The value of command line argument #1 is: red
The value of command line argument #2 is: green
```

a566170

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.

11-4. SLIDE: Arguments to Shell Programs (Continued)

Arguments to Shell Programs (Continued)

This shell program will install a program, specified as a command-line argument to your bin directory:

```
$ cat my_install
echo $0 will install $1 to your bin directory
chmod +x $1
mv $1 $HOME/bin
echo Installation of $1 is complete

$ chmod +x my_install

$ my_install color3
my_install will install color3 to your bin directory
Installation of color3 is complete
$
```

a566171

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.

11-5. 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.

11-5. 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

$ 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
```

a566173

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.

11-6. SLIDE: The shift Command

The shift Command

- Shifts all strings in * left *n* positions
- Decrements # by *n* (default value of *n* is 1)

Syntax: `shift [n]`

Example:

```
$ cat color5
orig_args=$*
echo There are $# command line arguments
echo They are $*
echo Shifting two arguments
shift 2
echo There are $# command line arguments
echo They are $*
echo Shifting two arguments
shift 2; final_args=$*
echo Original arguments are: $orig_args
echo Final arguments are: $final_args
```

a566174

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
$
```

11-7. SLIDE: The read Command

The **read** Command

Syntax:

```
read variable [ variable ... ]
```

Example:

```
$ cat color6
echo This program prompts for user input
echo "Please enter your favorite two colors -> \c"
read color_a color_b
echo The colors you entered are: $color_b $color_a
$ chmod +x color6
$ color6
This program prompts for user input
Please enter your favorite two colors ->red blue
The colors you entered are: blue red
$ color6
This program prompts for user input
Please enter your favorite two colors ->red blue tan
The colors you entered are: blue tan red
```

a6289

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.

11-7. 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

$ 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
```

a65017

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*.

11-8. LAB: Introduction to Shell Programming

Directions

Complete the following exercises and answer the associated questions.

1. 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.

2. 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.

3. 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.

4. Write a shell program called **alpha** that will display the first and last command line arguments. Hint: use the **cut** command.

5. 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.

Module 12 — 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.

12-1. 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.

12-2. SLIDE: The test Command

The `test` Command

Syntax:
`test` *`expression`* **or** `[` *`expression`* `]`

The `test` command evaluates the expression, and sets the return code.

Expression Value	**Return Code**
true	0
false	non-zero (commonly 1)

The `test` command can evaluate the condition of

- Integers
- Strings
- Files

a566181

Student Notes

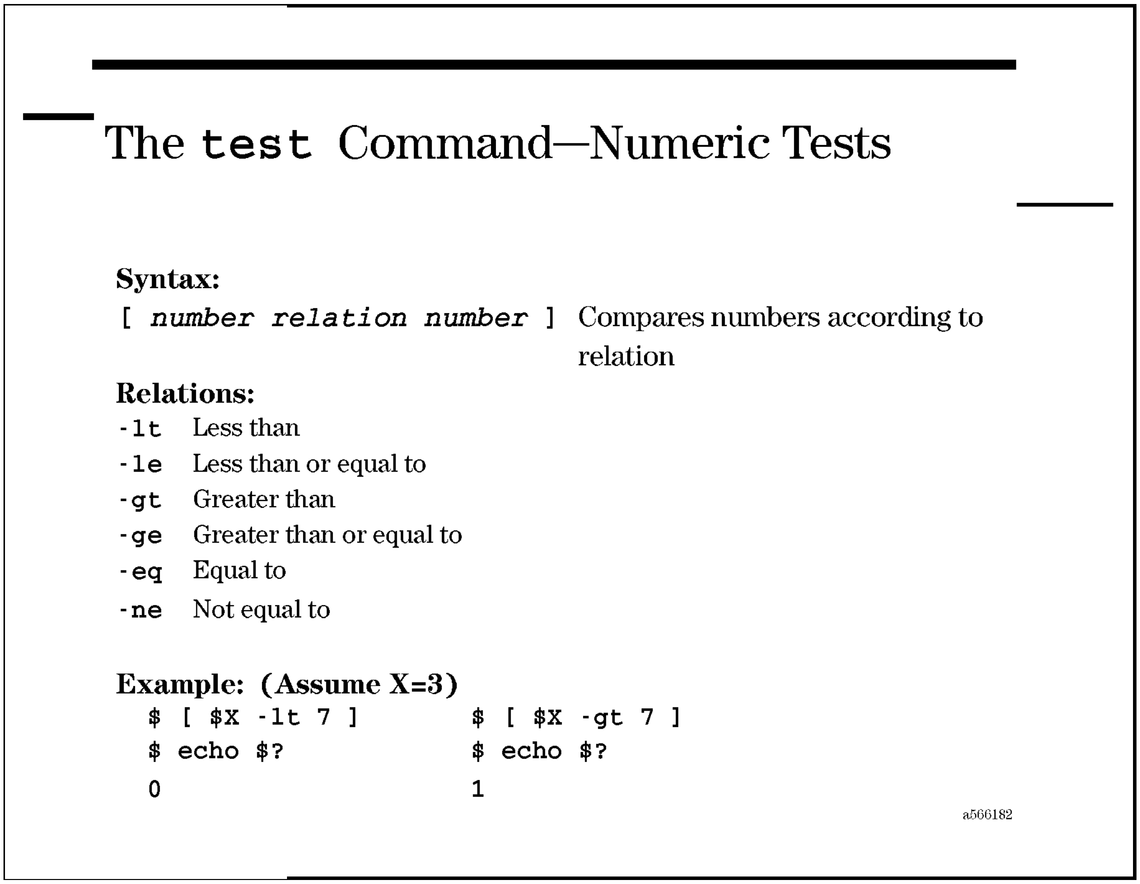
The **`test`** command is used to evaluate expressions and generate a return code. It takes arguments that form logical expressions and evaluates the expressions. *The* **`test`** *command writes nothing to standard output*. You must display the value of the return code to determine the result of the **`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 **`]`**.

12-3. 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.

12-4. SLIDE: The `test` Command — String Tests

The `test` Command—String Tests

Syntax:

`[` *`string1`* `=` *`string2`* `]`	Determines string equivalence
`[` *`string1`* `!=` *`string2`* `]`	Determines string nonequivalence

Example:

```
$ X=abc                          $ X=abc
$ [ "$X" = "abc" ]               $ [ "$X" != "abc" ]
$ echo $?                        $ echo $?
0                                1
```

a566183

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
```

12-5. SLIDE: The `test` Command — File Tests

The `test` Command—File Tests

Syntax:

`test -`*`option filename`* Evaluates filename according to option

Example:

```
$ test -f funfile
$ echo $?
0
$ test -d funfile
$ echo $?
1
```

a566181

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.

12-6. 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 12-1.

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 \( -d "$2" -o -s "$2" \) ]
```

12-7. SLIDE: The exit Command

The **exit** Command

Syntax:

```
  exit [arg]
```

Example:

```
$ cat exit_test
echo exiting program now
exit 99

$ exit_test
exiting program now

$ echo $?
99
```

a6685

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.

12-8. 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**.

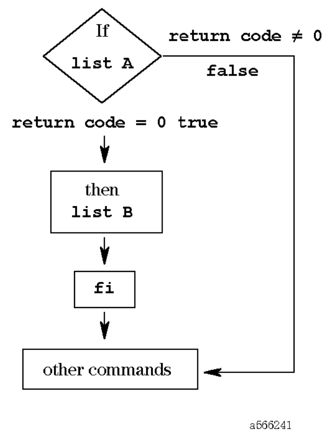


Figure 12-1. The if Construct Flowchart

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
```

12-9. 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
```

a566188

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`**.

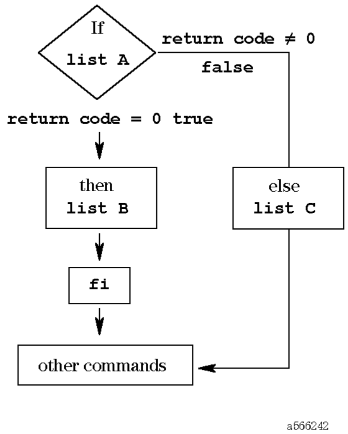


Figure 12-2. The if-else Construct Flowchart

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**.

12-10. 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                  case $OPT in
     yes)  echo O.K.               1)  echo option 1  ;;
           ;;                      2)  echo option 2  ;;
      no)  echo no go              3)  echo option 3  ;;
           ;;                      *)  echo no option ;;
 esac                          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.

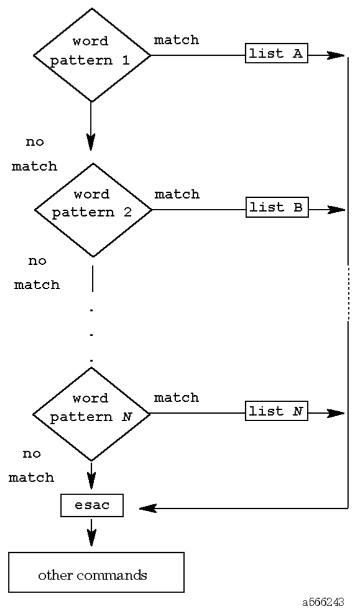


Figure 12-3. The case Construct Flowchart

12-11. SLIDE: The case Construct — Pattern Examples

The `case` Construct—Pattern Examples

The `case` construct patterns use the same special characters that are used to generate file names.

```
$ cat menu_with_case
 echo                  COMMAND MENU
 echo       d to display time and date
 echo       w to display logged-in users
 echo       l to list contents of current directory
 echo         Please enter your choice :
 read choice
 case $choice in
     [dD]*)    date ;;
     [wW]*)    who  ;;
     l*|L*)    ls   ;;
         *)    echo Invalid selection ;;
 esac
$
```

a566190

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.

12-12. 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

a566191

Student Notes

12-13. LAB: Shell Programming — Branches

Directions

Complete the following exercises and answer the associated questions.

1. 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.

2. 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.)

3. 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?)

4. 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.

5. 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.

6. 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.)

7. 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.

Module 13 — 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.

13-1. SLIDE: Loops — an Introduction

Loops—an Introduction

Purpose: Repeat execution of a list of commands.

Control: Based on the *return code* of a key command.

Three forms:

```
while ... do ... done
until ... do ... done
for ... do ... done
```

a566193

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**.

13-2. SLIDE: Arithmetic Evaluation Using `let`

Arithmetic Evaluaton 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

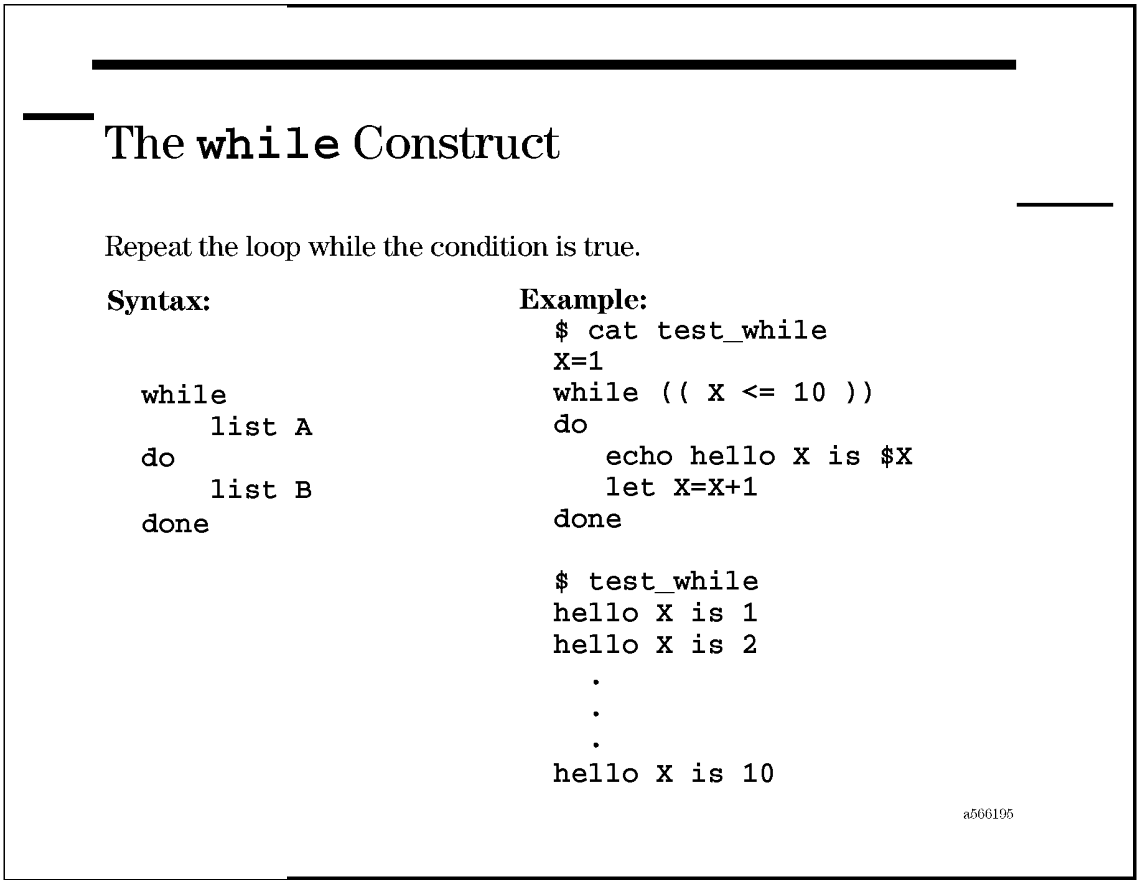
```
let "x=x/(y+1)"
```

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.

13-3. SLIDE: The while Construct



Student Notes

The **while** construct is a looping mechanism provided by the shell that will continue looping through the body of commands (*list B*) **while** a condition is true. The condition will be determined by the return code of the last command in *list A*. Often a **test** or **let** command is used to control the continuance of the loop, but any command can be used that generates a return code.

The example on the slide could have been written using a **test** command instead of the **let** command, as follows:

```
$ X=1
$ while [ $X -le 10 ]
> do
>      echo hello X is $X
>      let X=X+1
> done
```

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.

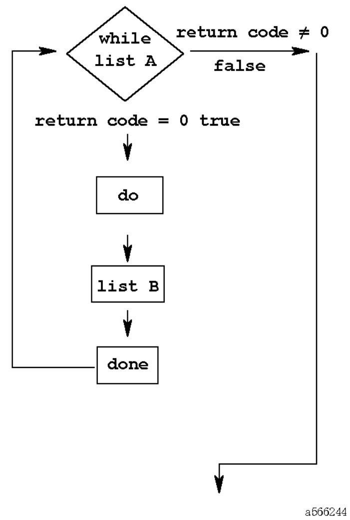


Figure 13-4. The while Construct Flowchart

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
```

13-4. 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.

13-5. 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.

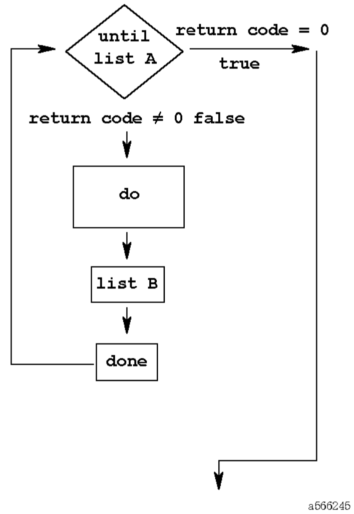


Figure 13-5. The until Construct Flowchart

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
```

13-6. SLIDE: The `until` Construct — Examples

The `until` Construct—Examples

Example A:
Repeat until ans is no.

```
ans=yes
until
 [ "$ans" = no ]
do
 echo Enter a name
 read name
 echo $name >> file.names
 echo "Continue?"
 echo Enter yes or no
 read ans
done
```

Example B:
Repeat until there are no cmd line arg.

```
until (( $# == 0 ))
do
 if test -d $1
 then
  echo contents of $1:
  ls -F $1
 fi
 shift
 echo There are $# items
 echo left on the cmd line.
done
```

a566198

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.

13-7. 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.

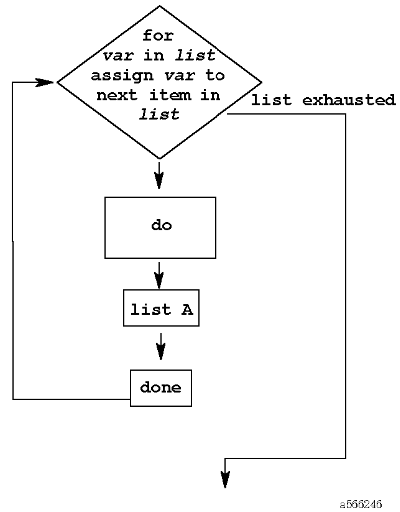


Figure 13-6. The for Construct Flowchart

13-8. SLIDE: The `for` Construct — Examples

The `for` Construct—Examples

Example A:

```
$ cat example_A
for NAME in $(grep home /etc/passwd | cut -f1 -d:)
do
  mail $NAME < mtg.minutes
  echo mailed mtg.minutes to $NAME
done
```

Example B:

```
$ cat example_B
for FILE in *
do
  if
     test -d $FILE
  then
     ls -F $FILE
  fi
done
```

a566200

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 $*                 or            for i
do                                        do
    cp $i $HOME/backups                       cp $i $HOME/backups
done                                      done
```

13-9. SLIDE: The `break`, `continue` and `exit` Commands

The `break`, `continue` and `exit` Commands

`break [n]`	Terminates the iteration of the loop and skips to the next command after [the *n*th] done.
`continue [n]`	Stops the current iteration of the loop and skips to the *beginning* of the next iteration [of the *n*th] enclosing loop.
`exit [n]`	Stops the execution of the shell program, and sets the return code to *n*.

a566201

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**?

13-10. 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.

13-11. SLIDE: Shell Programming — Loops — Summary

Shell Programming—Loops—Summary

`let` *`expression`*	evaluate an arithmetic expression
`((`*`expression`*`))`	evaluate an arithmetic expression
`while` *`condition is true`* `do ... done`	while
`until` *`condition is true`* `do ... done`	until
`for` *`var`* `in` *`list`* `do ... done`	for
`break [`*`n`*`]`	break out of loop
`continue[`*`n`*`]`	terminate current iteration of loop
`exit [`*`n`*`]`	terminate the program

a566203

Student Notes

13-12. 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. 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

4. 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.

5. 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.

Appendix A — Commands Quick Reference Guide

Objectives

- To provide 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

```
                                > stdout (1)
                              /
                             /
stdin (0) <-------- cmd
                             \
                              \
                                2 >stderr (2)
```

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

```
cmd            |        cmd(FILTER)     |       cmd
   \                   /   \                   /   \
    \                 /     \                 /     \
     stdout ----> stdin      stdout ----> stdin      stdout
```

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
     cmd(s)
then
     cmdtrue(s)
else
     cmdfalse(s)
fi
```

if RETURN VALUE of LAST cmd is true do cmds following then
if RETURN VALUE of LAST cmd is false do cmds following else

```
case $vara in

     pat1) cmdsa
```

compare value of vara to patterns
execute commands that follow matching pattern

```
            ;;
      pat2) cmdsb
            ;;
         *) cmds default
            ;;
esac
```

Looping

`while`
`     cmd(s)`
`do`
`     cmdtrue(s)`
`done`
	while RETURN VALUE of LAST cmd is true do cmds following do

`until`
`     cmd(s)`
`do`
`     cmdfalse(s)`
`done`
	until RETURN VALUE of LAST cmd is true do cmds following do

`for vara in a b c d e`
`do`
`     cmd(s)`
`done`
	assign vara to each item in list, do cmds

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**.

Solutions

1-6. 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. 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**.

3. 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
```

4. 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
```

5. 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.

2-14. LAB: The File System

1. 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

```
                              tree/
                                |
  -----------------------------------------------------------------
  |          |            |       |        |           |        |
cherry   car.models/    collie   probe   dog.breeds/   poodle   taurus
             |                             |
     --------------               -------------------
```

```
       |      |      |                |                 |
   chrysler/  gm/   ford/         retriever/         shepherd/
                     |                |
                ----------       -----------------------------
                |        |       |           |               |
              sedan/   sports/   golden     labrador         mixed
                         |
                       mustang
```

2. 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**

3. 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
```

3-16. LAB: File and Directory Manipulation

1. 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
```

2. 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
```

3. 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
```

4. 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
```

5. 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
```

6. 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
```

---

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**.

---

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
$
```

4-12. 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. 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".

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
```

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
```

5-11. LAB: Exercises

1. Set up an alias called **go** to change your working directory to **tree** and do an **ls -F**. Now type 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
```

2. 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.

3. 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
```

6-12. 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. 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
```

3. Now start a child shell by typing **sh**. Look at the contents of *MYNAME* now. What happened? Exit the child shell (use [Ctrl]+[c] [Return] 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
```

4. What command can be typed 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
```

5. 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.

6. 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.

7. 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
```

8. 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.

9. 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
```

7-11. LAB: Input and Output Redirection

1. 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]
```

2. 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**.

3. 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.

4. 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
```

5. 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.

6. 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.

7. 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.)

8. 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.)

9. Using redirection and filters, how many users are logged in on the system?

Answer:

```
$ who > whoson
$ wc -l whoson
```

8-11. LAB: Pipelines

1. 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***

2. 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
```

3. Create an alias called **whoson** that will display an alphabetical listing of the users currently logged into your system.

Answer:

```
$ alias whoson="who | sort"
```

4. 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
```

5. 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. Second, use the same pipeline but now output the list in numerical order by user ID number. 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
```

9-11. 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*.

10-7. 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
```

11-8. LAB: Introduction to Shell Programming

1. 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
```

2. 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"
```

3. 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"
```

4. 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."
```

5. 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"
```

12-13. LAB: Shell Programming — Branches

1. 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
```

2. 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
```

3. 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
```

4. 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
```

5. 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
```

6. 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
```

7. 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
```

13-12. 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. 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
```

4. 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
```

5. 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
```