

*Fundamentals of Solaris™ 8
Operating Environment
for System Administrators*

SA-118

Student Guide



Sun Microsystems, Inc.
MS BRM01-209
500 Eldorado Boulevard
Broomfield, Colorado 80021
U.S.A.

Revision A, June 2000

Copyright 2000 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303, U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, Solaris Operating Environment, Common Desktop Environment (CDE), admintool, SPARC, and Sun-OS are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

U.S. Government approval required when exporting the product.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g) (2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015 (b)(6/95) and DFAR 227.7202-3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS, AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.



Please
Recycle



Adobe PostScript

Contents

About This Course	xv
Course Goal	xv
Course Overview	xvi
Course Map	xvii
Module-by-Module Overview	xviii
Course Objectives	xxi
Skills Gained by Module	xxii
Guidelines for Module Pacing	xxiii
Topics Not Covered	xxiv
How Prepared Are You?	xxv
Introductions	xxvi
How to Use the Course Materials	xxvii
Course Icons and Typographical Conventions	xxviii
Icons	xxviii
Typographical Conventions	xxix
Introduction to the Solaris 8 Operating Environment	1-1
Objectives	1-1
Additional Resources	1-1
Introduction to the Solaris Operating Environment	1-2
Main Components of a Computer	1-3
Hardware Overview	1-3
Random Access Memory	1-3
Central Processing Unit	1-4
Input/Output	1-4
Hard Disk	1-4
The Solaris Operating Environment	1-6
The SunOS Operating System	1-7
The kernel	1-7
The Shell	1-8
The Bourne Shell	1-9
The C Shell	1-10
The Korn Shell	1-10
Tasks	1-11

Exercise Summary.....	1-13
Task Solutions.....	1-14
Check Your Progress	1-15
Accessing the System	2-1
Objectives	2-1
Additional Resources	2-2
User Accounts.....	2-3
The root Account	2-3
The /etc/passwd Entry	2-3
Logging In	2-5
Logging In Using the Login Screen	2-7
Logging In Using the Command Line	2-8
Password Requirements	2-9
Changing Your Password	2-10
Changing Your Password in CDE	2-10
Changing Your Password from the Command Line	2-10
Securing the CDE Session	2-11
Locking the Screen	2-11
Exiting the Session	2-11
Basic UNIX Commands	2-13
Using the uname Command.....	2-13
Using the date Command.....	2-13
Using the cal Command	2-14
Command-Line Syntax	2-15
Control Characters	2-16
Viewing Online Documentation	2-17
Command Format.....	2-17
Using the man Command without Options.....	2-17
Scrolling in Man Pages	2-18
Searching Man Pages by Section.....	2-18
Using the man Command with the -k Option	2-19
Searching Man Pages by Keyword.....	2-19
Determining Current Users	2-20
Command Format.....	2-20
Displaying Users on the System	2-20
Identifying the User Name	2-21
Command Format.....	2-21
Example	2-21
Identifying User Group Details	2-22
Command Format.....	2-22
Identifying a User	2-22
Entering Multiple Commands from a Single Command Line .	2-23
Exercise: Accessing the System	2-24
Tasks	2-24
Exercise Summary.....	2-29

Task Solutions.....	2-30
Check Your Progress	2-32
Accessing Files and Directories.....	3-1
Objectives	3-1
Additional Resources	3-1
The Directory Tree	3-2
Pathnames	3-3
Absolute Pathname.....	3-4
Relative Pathname	3-4
File and Directory Naming Conventions	3-6
Changing Directories.....	3-7
Command Format.....	3-7
Moving Around the Directory Tree	3-7
Displaying the Current Directory.....	3-8
Command Format.....	3-8
Determining the Current Working Directory	3-8
Changing Directories with Pathname Abbreviations.....	3-9
Displaying the Contents of a Directory	3-10
Command Format.....	3-10
Listing the Contents of a Directory	3-10
Displaying Hidden Files	3-10
Displaying File Types.....	3-11
Displaying a Long Listing.....	3-12
Listing Individual Directories	3-13
Metacharacters.....	3-15
Using the Tilde (~) Character	3-15
Using the Dash	3-16
Using the Asterisk.....	3-16
Using the Question Mark.....	3-17
Using the Square Brackets	3-18
Exercise: Accessing Files and Directories	3-20
Tasks	3-20
Exercise Summary.....	3-23
Task Solutions.....	3-24
Check Your Progress	3-27
Directory and File Commands.....	4-1
Objectives	4-1
Additional Resources	4-1
Determining File Type	4-2
Command Format.....	4-2
Example Text File.....	4-2
Example Data File	4-3
Example Executable File	4-3
Displaying the Contents of a Text File	4-4
Command Format.....	4-4

Using the <code>cat</code> Command to Display a Short Text File	4-4
Using the <code>cat</code> Command to Create a Short Text File	4-5
Joining Multiple Files	4-5
Using the <code>strings</code> Command	4-6
Browsing the Contents of a File	4-8
Command Format.....	4-8
Scrolling Keys	4-9
Command Format.....	4-10
Scrolling Keys	4-10
Command Format.....	4-11
Displaying a Specific Number of Lines at the Beginning of a File	4-11
Command Format.....	4-12
Displaying a Specified Number of Lines at the End of a File	4-12
Displaying Lines From a Specific Point in the File	4-12
Command Format.....	4-13
Using the <code>wc</code> Command with Options.....	4-13
Using the <code>wc</code> Command without Options.....	4-13
Determining the Number of Lines in a File	4-14
Creating Empty Files	4-15
Command Format.....	4-15
Creating Empty Files	4-15
Creating and Appending Files Using the <code>tee</code> Command	4-16
Command Format.....	4-16
Creating a Text File.....	4-16
Appending Text to a File	4-17
Capturing and Displaying Output	4-17
Creating Directories.....	4-18
Command Format.....	4-18
Creating a New Directory	4-18
Creating Multiple Levels of Directories.....	4-19
Copying Files and Directories.....	4-20
Copying Files	4-20
Copying Directories.....	4-21
Moving and Renaming Files and Directories	4-23
Command Format.....	4-23
Renaming Files in the Current Directory	4-23
Moving Files to Another Directory	4-24
Renaming Directories.....	4-24
Moving a Directory and its Contents.....	4-25
Renaming Files in Another Directory	4-25
Removing Files and Directories	4-26
Removing Files	4-26
Removing Directories.....	4-27

Command-line Printing	4-29
Command Format.....	4-29
Options	4-29
Sending Files to a Printer	4-30
Displaying Printer Status and Queues.....	4-31
Command Format.....	4-31
Options	4-31
Displaying the Status of All Print Requests	4-31
Displaying Requests on a Specific Printer's Queue	4-32
Determining the Status of All Configured Printers	4-32
Determining Which Printers Are Configured on the System.....	4-32
Displaying Which Printers Are Accepting Requests	4-33
Removing a Print Request	4-34
Command Format.....	4-34
Canceling a Print Request.....	4-34
Format and Print a File.....	4-35
Command Format.....	4-35
Options	4-35
Format and Print Files to the Screen	4-36
Exercise: Using Directory and File Commands.....	4-37
Tasks	4-37
Exercise Summary.....	4-41
Task Solutions.....	4-42
Check Your Progress	4-46
Searching for Files and Text	5-1
Objectives	5-1
Additional Resources	5-1
Locating Files Using the <code>find</code> Command	5-2
Command Format.....	5-2
Finding Differences Between Files	5-5
Locating Text Differences Using the <code>cmp</code> Command.....	5-5
Locating Text Differences Using the <code>diff</code> Command.....	5-5
Sorting Data	5-8
Command Format.....	5-8
Options	5-8
Using <code>sort</code> with Different Options.....	5-9
Using <code>sort</code> on Different Fields Within a File	5-10
Using the <code>grep</code> Command.....	5-12
Using the <code>egrep</code> Command.....	5-17
Using the <code>fgrep</code> Command.....	5-19
Exercise: Locating Files and Text.....	5-20
Tasks	5-20
Exercise Summary.....	5-22
Task Solutions.....	5-23
Check Your Progress	5-25

File Security	6-1
Objectives	6-1
Additional Resources	6-1
Security Overview	6-2
File Type	6-5
User (Owner) Permissions	6-5
Group	6-5
Others (Public).....	6-6
Determining Access to a File or Directory	6-7
Process for Determining Permissions	6-7
Types of Permissions	6-8
Changing Permissions With Symbolic Mode	6-11
Octal (Absolute) Mode	6-12
Changing Permission With Octal Mode	6-14
Default Permissions	6-15
The umask Filter	6-15
Understanding the umask Filter	6-16
Changing the umask Value	6-17
Exercise: Changing File Permissions.....	6-19
Tasks	6-19
Exercise Summary.....	6-22
Task Solutions.....	6-23
Check Your Progress	6-26
Visual (vi) Editor	7-1
Objectives	7-1
Additional Resources	7-1
Introducing vi	7-2
vi Modes	7-3
Command Mode	7-3
Edit Mode.....	7-3
Last Line Mode.....	7-3
Switching Modes.....	7-4
Invoking vi	7-5
Command Format.....	7-5
Input Commands	7-5
Positioning Commands.....	7-6
Editing Commands.....	7-7
Deleting Text.....	7-7
Undoing, Repeating, and Changing Text Commands	7-7
Copying and Pasting Text	7-9
Saving and Quitting Files	7-9
Customizing Your vi Session	7-11
Exercise: Using the vi Editor	7-13
Tasks	7-13
Exercise Summary.....	7-15
Check Your Progress	7-16

Archiving User Data	8-1
Objectives	8-1
Overview of Archive Commands.....	8-2
Archiving Files Using the tar Command.....	8-3
Command Format.....	8-3
Functions	8-3
Creating, Viewing, and Retrieving a Directory from a Tape	8-3
Creating, Viewing, and Retrieving Files from an Archive File.....	8-6
Compressing Files Using the compress Command	8-7
Command Format.....	8-7
Compressing a File	8-7
Command Format.....	8-8
Uncompressing a File	8-8
Viewing the Contents of a Compressed File.....	8-8
Command Format.....	8-9
Viewing the Contents of a Compressed File.....	8-9
Compressing Files Using the jar Command	8-10
Command Format.....	8-10
Options	8-10
Adding All the Files in a Directory to an Archive	8-11
Other Tools.....	8-12
Using the zip Command	8-12
Using the gzip Command.....	8-13
Using the cpio Command.....	8-15
Command Format.....	8-15
Options	8-15
Creating File Archives.....	8-16
Copying All Files in a Directory to a Tape.....	8-17
Listing the Files on a Tape	8-17
Retrieving All Files from a Tape	8-17
Detecting Removable Media Devices	8-19
Command Format.....	8-19
Accessing Removable Media Devices.....	8-20
Ejecting Removable Media Devices	8-22
Ejecting a CD-ROM.....	8-22
Device Busy Message	8-23
Exercise: Saving and Restoring Files.....	8-24
Tasks	8-24
Exercise Summary.....	8-27
Task Solutions.....	8-28
Check Your Progress	8-31
Remote Connections	9-1
Objectives	9-1

Additional Resources	9-1
Example Networking Environment	9-2
Network.....	9-2
Host.....	9-2
Using the telnet Command	9-3
Command Format.....	9-3
Using the rlogin Command	9-4
Command Format.....	9-4
Example.....	9-4
Specifying a Different Login ID	9-5
Command Format.....	9-5
Logging in Remotely as Another User.....	9-5
Executing a Program on a Remote Machine	9-6
Command Format.....	9-6
Example.....	9-6
Copying to and from Another Machine	9-7
Command Format.....	9-7
Copying Files Across the Network.....	9-7
Using the ftp Command.....	9-8
Command Format.....	9-8
Examples	9-9
Exercise: Performing Network Basics	9-11
Tasks	9-11
Exercise Summary.....	9-12
Task Solutions.....	9-13
Check Your Progress	9-14
System Processes	10-1
Objectives	10-1
Additional Resources	10-1
Process Overview.....	10-2
Process UID and GID	10-2
Parent Process.....	10-2
Viewing Processes and PIDs	10-3
Command Format.....	10-3
Options	10-3
Displaying a Full Listing of all Processes	10-3
Searching for a Specific Process	10-4
The pgrep Command.....	10-5
Command Format.....	10-5
Options	10-5
Sending Signals to Processes	10-7
Signals.....	10-7
The kill Command	10-9
Killing Processes Remotely.....	10-11
Managing Jobs	10-12

Exercise: Manipulating System Processes	10-15
Tasks	10-15
Exercise Summary.....	10-18
Task Solutions.....	10-19
Check Your Progress	10-22
The Korn Shell	11-1
Objectives	11-1
The Shell as a Command Interpreter	11-2
Responsibilities of the Shell as a Command Interpreter ...	11-3
Input and Output Redirection and Piping	11-4
Redirecting Data I/O.....	11-4
File Descriptors.....	11-5
Standard Command Input and Output.....	11-5
Redirecting stdin, stdout, and stderr	11-6
The Pipe Feature.....	11-8
Command Format.....	11-8
Some Basic Examples Using a Pipe	11-8
Building a Pipeline.....	11-9
Options of the Korn Shell.....	11-10
Protecting File Content During I/O Redirection	11-10
File Name Completion in the Korn Shell.....	11-12
Using File Name Completion.....	11-12
Variables of the Korn Shell	11-14
Display of Variables.....	11-15
Setting Shell Variables.....	11-16
Unsetting Shell Variables.....	11-17
Variables Set by the Shell on Login	11-17
Customizing Korn Shell Variables	11-19
The Prompt Variable PS1.....	11-19
The PATH Variable	11-20
Extending the PATH Variable	11-21
Metacharacters of the Korn Shell.....	11-22
Quoting with the Backslash.....	11-22
Quoting with Single Quotes and Double Quotes.....	11-22
Command Substitution	11-23
The History Mechanism.....	11-24
The history command.....	11-24
The r command.....	11-25
Using vi Commands to Edit a	
Previously Executed Command	11-26
The Korn Shell Alias Utility.....	11-28
Command Format.....	11-28
Pre-defined Korn Shell Aliases	11-28
User-defined Aliases.....	11-29
Command Sequences	11-30

Removing Aliases	11-31
Defining a Function	11-32
Some Function Examples	11-32
Configuring of the Korn Shell Environment.....	11-34
The ~/.profile File	11-34
The ~/.kshrc File.....	11-34
Reading Initialization Files	11-35
The .dtprofile File.....	11-36
Exercise: Modifying the Korn Shell.....	11-37
Tasks	11-37
Exercise Summary.....	11-41
Task Solutions.....	11-42
Check Your Progress	11-46
Introduction to sed and awk.....	12-1
Objectives	12-1
The Stream Editor	12-2
Command Format.....	12-2
Options	12-2
Using the Stream Editor.....	12-4
Deleting Lines with the d Command	12-4
Printing Lines with the p Command.....	12-5
Placing Characters at the End of Each Line	12-5
Changing Spaces to Colons in Data	12-5
Multiple Edits with sed	12-6
Text Processing Using the awk Command	12-7
Command Format.....	12-7
Basic awk Command Format.....	12-7
Using awk to Display Specific Data.....	12-8
Using awk to Change the Format of Data	12-11
Exercise: Using sed and awk	12-14
Tasks	12-14
Exercise Summary.....	12-17
Task Solutions.....	12-18
Check Your Progress	12-20
Reading Shell Scripts	13-1
Objectives	13-1
Additional Resources	13-1
The Basics of Shell Scripts.....	13-2
Determining the Type of Shell Script.....	13-2
Creating a Basic Shell Script	13-3
Bourne Shell Programming	13-4
Bourne Shell Scripts	13-5
Special Shell Built-in Variables	13-6
Positional Parameters	13-6

Conditional Constructs and Flow Control	13-10
Conditional Commands	13-10
Exit Status	13-10
The test Command	13-11
An Alternate Format for test	13-12
The case Command	13-14
The exit Command	13-16
The /etc/init.d/audit Script	13-19
Interpreting the audit Administration Script	13-20
Exercise: Introduction to Scripting	13-23
Tasks	13-23
Exercise Summary	13-26
Task Solutions	13-27
Check Your Progress	13-29

About This Course

Course Goal

The Fundamentals of Solaris™ 8 Operating Environment for System Administrators course teaches you how to use UNIX® operating system commands and basic Solaris Operating Environment commands.

Course Overview

The class is for new users of the Solaris Operating Environment. You will learn fundamental command-line features of the Solaris Operating Environment, including:

- File system navigation
- File permissions
- The `vi` text editor
- Command shells
- Basic network use

Course Map

The following course map enables you to see what you have accomplished and where you are going in reference to the course goal.

Getting Started

“Introduction
to the Solaris 8
Operating
Environment”

“Accessing the
System”

File Operations

“Accessing
Files and
Directories”

“Directory and
File
Commands”

“Searching for
Files and Text”

“File Security”

Text Editing

“Visual (vi)
Editor”

Saving Data

“Archiving
User Data”

Connecting to Other Hosts

“Remote
Connections”

Manipulating Processes

“System
Processes”

Shell Operations

“The Korn
Shell”

“Introduction to
sed and awk”

“Reading Shell
Scripts”

Module-by-Module Overview

This course contains the following modules:

- Module 1 – “Introduction to the Solaris 8 Operating Environment”

This module provides a brief introduction to the components of the Solaris Operating Environment.

Lab exercise – The lab provides a review of the main components of a computer and the Solaris 8 Operating Environment.

- Module 2 – “Accessing the System”

This module provides instructions on logging in to the system, executing simple commands, and acquiring simple command syntax from online documentation.

Lab exercise – The lab practices logging in and logging out of the system, as well as using the online documentation.

- Module 3 – “Accessing Files and Directories”

This module introduces the commands needed to traverse the Solaris Operating Environment directory tree. The concept of absolute and relative pathnames is introduced as is the use of metacharacters to list files with similar naming characteristics.

Lab exercise – The lab centers around commands needed to list the contents of specified directories and commands needed to move between directories.

- Module 4 – “Directory and File Commands”

This module focuses on commands that read the contents of existing files or create and remove new files or directories. Commands to copy or rename files, and commands to perform basic print functions are also addressed.

Lab exercise – Copying or renaming files, creating and deleting directories, and manipulating print queues are addressed in this lab.

- Module 5 – “Searching for Files and Text”

This module addresses commands used to sort text, compare files, and search through files for regular expressions.

Lab exercise – This lab deals with the `sort`, `cmp`, `diff`, `find`, and `grep` commands.

- Module 6 – “File Security”

The focus of this module is file and directory permissions.

Lab exercise – This lab practices use of the commands needed to change file and directory permissions, as well as setting of default permissions through the use of `umask`.

- Module 7 – “Visual (vi) Editor”

This module provides instruction on the use of the `vi` editor.

Lab exercise – The lab is a guided practice in the use of `vi` commands.

- Module 8 – “Archiving User Data”

This module introduces various commands and utilities that support archive and compression of user data.

Lab exercise – The lab demonstrates use of the various archive and data compression commands and utilities.

- Module 9 – “Remote Connections”

This module focuses on establishing connections to remote hosts.

Lab exercise – This lab demonstrates the use of the `rlogin`, `telnet`, and `ftp` commands.

- Module 10 – “System Processes”

This module introduces basic job control and process management. It also covers the use of signals for controlling process activity.

Lab exercise – This lab focuses on the `fg` and `bg` commands for job control, the `ps` command for process management, and the use of signals with the `kill` command.

- Module 11 – “The Korn Shell”

This module introduces the use of the Korn shell as a command interpreter.

Lab exercise – This lab demonstrates basic shell operations using the Korn shell.

- Module 12 – “Introduction to sed and awk”

This module is an introduction to using the streams editor (`sed`) and using the `awk` command as a text processor used for manipulating data and generating reports.

Lab exercise – This lab guides you through some simple examples using `sed` and `awk`.

- Module 13 – “Reading Shell Scripts”

This module guides you through interpretation of basic shell script programming examples, such as the `if`, `test`, and `case` commands, and the use of shell variables.

Lab exercise – The lab leads you through interpretation of basic system administration scripts using the previously introduced shell commands.

Course Objectives

Upon completion of this course, you should be able to:

- Log in and log out of the Solaris Operating Environment and CDE systems
- Compose command-line strings to perform Solaris Operating Environment functions
- Navigate the Solaris Operating Environment directory tree
- Manipulate text files
- Create files and directories
- Back up and restore user files and directories
- Change permissions of files and directories
- Use the `vi` text editor
- Identify and modify initialization files
- Employ shell features to streamline command execution
- Use basic network commands
- Use commands to search directories and files
- List active user processes and selectively kill user processes

Skills Gained by Module

The skills for *Fundamentals of Solaris™ 8 Operating Environment for System Administrators* are shown in the first column of the following matrix. The black boxes indicate the main coverage for a topic; the gray boxes indicate the topic is briefly discussed.

Skills Gained	Module												
	1	2	3	4	5	6	7	8	9	10	11	12	13
Log in and log out of Solaris Operating Environment and CDE systems													
Compose command-line strings to perform Solaris Operating Environment functions													
Navigate the Solaris Operating Environment directory tree													
Create files and directories													
Manipulate text files													
Use commands to search directories and files													
Change permissions of files and directories													
Use the vi text editor													
Back up and restore user files and directories													
Use basic network commands													
List active user processes and selectively kill user processes													
Employ shell features to streamline command execution													
Identify initialization files and read basic scripts													

Guidelines for Module Pacing

The following table provides a rough estimate of pacing for this course:

Module	Day 1	Day 2	Day 3	Day 4
"About This Course"	A.M.			
"Introduction to the Solaris 8 Operating Environment"	A.M.			
"Accessing the System"	A.M. /P.M.			
"Accessing Files and Directories"	P.M.			
"Directory and File Commands"		A.M.		
"Searching for Files and Text"		A.M.		
"File Security"		P.M.		
"Visual (vi) Editor"		P.M.		
"Archiving User Data"			A.M.	
"Remote Connections"			A.M. /P.M.	
"System Processes"			P.M.	
"The Korn Shell"				A.M.
"Introduction to sed and awk"				A.M. /P.M.
"Reading Shell Scripts"				P.M.

Topics Not Covered

This course does not cover the topics shown below. Many of the topics mentioned here are covered in other courses offered by Sun Educational Services:

- Detailed Shell Programming – Covered in SA-225: *Advanced UNIX® Utilities and Shell Programming for Administrators*
- System Administration concepts – Covered in SA-238: *Solaris™ 8 Operating Environment System Administration I*

Refer to the Sun Educational Services catalog for specific information on course content and registration.

How Prepared Are You?

To be sure you are prepared to take this course, can you answer yes to the question below?

- Can you input commands and control characters from a keyboard?

Introductions

Now that you have been introduced to the course, introduce yourself to each other and the instructor, addressing the items shown on the overhead.

How to Use the Course Materials

To enable you to succeed in this course, these course materials employ a learning model including the following components:

- **Course Map** — An overview of the course content appears in the “About This Course” module so you can see how each module fits into the overall course goal.
- **Objectives** — At the beginning of each module is a list of what you should be able to accomplish after completing the module.
- **Lecture** — The instructor presents information specific to the topic of the module. This information helps you learn the knowledge and skills necessary to succeed with the exercises.
- **Exercise** — Lab exercises give you the opportunity to practice your skills and apply the concepts presented in the lecture.
- **Check Your Progress** — Module objectives are restated, sometimes in question format, so you are sure you can accomplish the objectives of the current module before moving on in the course.

Course Icons and Typographical Conventions

The following icons and typographical conventions are used in this course to represent various training elements and alternative learning resources.

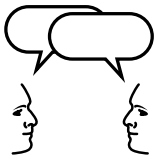
Icons



Additional resources – Indicates additional reference materials are available.



Demonstration – Indicates a demonstration of the current topic is recommended at this time.



Discussion – Indicates a small-group or class discussion on the current topic is recommended at this time.



Exercise objective – Indicates the objective for the lab exercises that follow. The exercises are appropriate for the material being discussed.

Note – This will contain additional important, reinforcing, interesting, or special information.



Caution – This points out a potential hazard to data or machinery.



Warning – This warns of anything that poses personal danger or irreversible damage to data or the operating system.

Typographical Conventions

Courier is used for the names of commands, files, and directories, as well as on-screen computer output. For example:

```
Use ls -al to list all files.  
system% You have mail.
```

Courier bold is used for characters and numbers you type. For example:

```
system% su  
Password:
```

Courier italic is used for variables and command-line placeholders that are replaced with a real name or value. For example:

To delete a file, type `rm filename`.

Palatino italics is used for book titles, new words or terms, or words that are emphasized. For example:

Read Chapter 6 in *User's Guide*.
These are called *class* options.
You *must* be *root* to do this.

Introduction to the Solaris 8 Operating Environment

1 

Objectives

Upon completion of this module, you should be able to:

- List the four main hardware components of a computer
- Describe the four main components of the Solaris Operating Environment
- State the three main components of the SunOS™ operating system
- Identify the default shells available in the Solaris Operating Environment
- Understand the basics of the Common Desktop Environment (CDE)

Additional Resources



Additional resources – The following reference can provide additional details on the topics discussed in this module:

- *System Administration Guide*, Part Number 805-7228

Introduction to the Solaris Operating Environment

The UNIX® operating system was originally developed at AT&T Bell Laboratories in 1969. It was created as a tool set by programmers for programmers. The early source code was made available to universities all over the country.

Programmers at the University of California at Berkeley made significant modifications to the original source code and called it BSD (Berkeley Software Distribution) UNIX. This version of the UNIX environment was sent to other programmers around the country, who then added tools and code to further enhance BSD UNIX.

Possibly the most important advance made to the operating system by the programmers at Berkeley was the addition of networking software. This allowed the operating system to function in a local area network (LAN).

Sun's original version of the UNIX operating system was known as SunOS, based on BSD UNIX Version 4.2. At that time, AT&T's version of the UNIX environment was known as System V.

In 1988, SunOS/BSD, AT&T UNIX, and XENIX were combined into what became System V Release 4 (SVR4) UNIX.

This new generation of the operating system was an effort to combine the best features of both BSD and AT&T UNIX, creating an industry standard for the operating system.

The new SVR4 became the basis for not only Sun and AT&T versions of the UNIX environment, but also IBM's AIX and Hewlett-Packard's HP-UX.

Main Components of a Computer

The core of all computer systems is the hardware that works with the system software to perform various tasks.

The computer hardware is made up of a number of different components, such as central processing unit (CPU), memory, and disks, each of which has a specific purpose. For these components to work as a team, they require management by the operating system.

The operating system is a collection of programs and files whose primary functions are to instruct the computer on what to do with the hardware.

The Solaris Operating Environment runs on two types of hardware, or platforms: SPARC™ platform edition and IA (Intel 32-bit processor architecture).

Hardware Overview

The four main hardware components of a computer are the random access memory (RAM), the CPU, the input/output (I/O), and the hard disk or other mass storage device.

Random Access Memory

RAM is the main computer memory, often referred to as *primary memory*. It is that part of memory where system processing occurs. The statement, "This system has 64 Mbytes of memory," refers to the amount of RAM or primary memory currently installed.

A software program resides on the *hard disk*, and when activated, an image or copy of that program is loaded into RAM.

This remains in RAM as long as needed. When images are no longer required, each can be overwritten by other images. If the system experiences a power loss or is rebooted, all images in RAM disappear.

Central Processing Unit

The CPU is the computer logic chip that executes instructions received from primary memory (RAM). These instructions are stored in binary language.

Input/Output

The I/O component reads input from a device, such as a keyboard, into memory, and it writes output from memory to a device, such as a terminal window.

Your input devices include the keyboard and mouse. The monitor, printer, and tape drive would be examples of primary output devices.

Hard Disk

The hard disk is a magnetic storage device where data is stored permanently. All files, directories, and applications are stored on a hard disk.

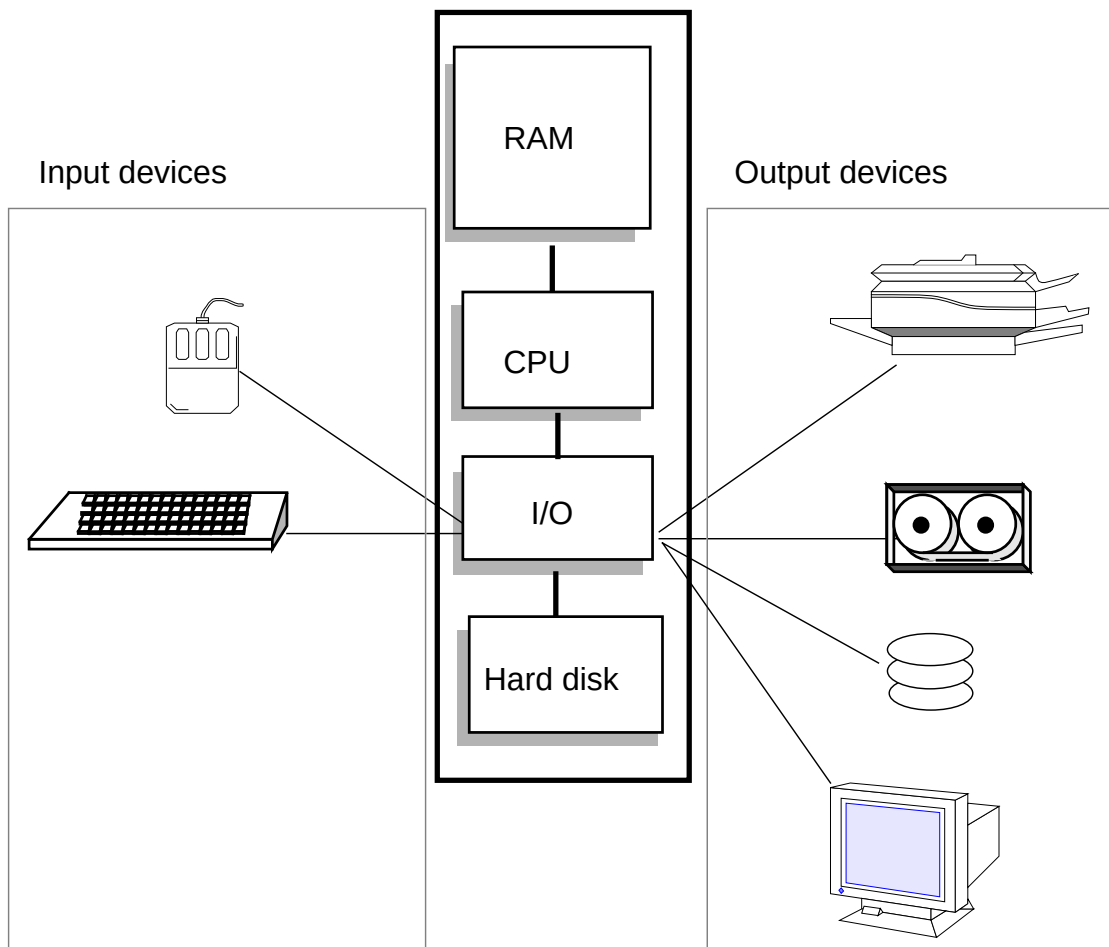


Figure 1-1 Main Hardware Components of a Computer

The Solaris Operating Environment

The Solaris Operating Environment contains the following four components.

- SunOS 5.x operating system (based on SVR4 UNIX OS). This is the heart of the Solaris Operating Environment.
- ONC+™ Technologies (Open Network Computing), consists of a set of core services for enterprise-wide distributed computing. ONC+ services include TI-RPC and XDR, which provide the basis for applications, such as NIS+. These services are completely compatible and interoperate with the installed base of ONC services, including NIS, NFS, and RPC.
- Solaris Common Desktop Environment (CDE) graphical user interface (GUI).
- OpenWindows™ graphical environment.

Like all operating systems, SunOS is a collection of software that manages system resources and schedules system operations.

The operating system interprets instructions from the user, or an application, and tells the computer what to do. It handles input and output, keeps track of data stored on disks, and communicates with peripherals, such as monitors, hard disks, floppy disk drives, printers, or modems.

The SunOS Operating System

There are three main components of the SunOS operating system:

- The kernel
- The shell
- The directory tree

Note – The directory tree is discussed in Module 3, “Accessing Files and Directories.”

The kernel

The kernel is the core of the SunOS operating system. It is the master program that manages all the physical resources of the computer, including:

- File systems
- Device management
- Process management
- Memory management

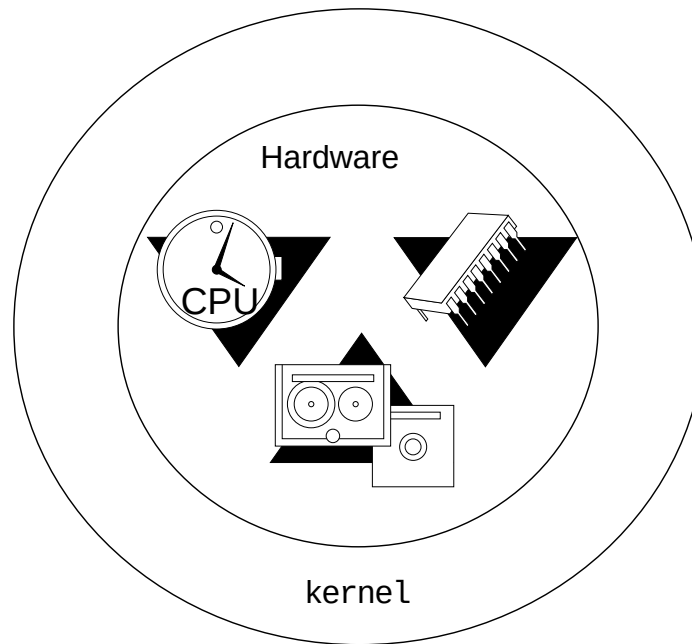


Figure 1-2 The Role of the kernel

The Shell

The *shell* is an interface between the user and the kernel. The primary function of a shell is to be a command interpreter. That is, the shell accepts commands you enter, it interprets these commands, and then it executes them.

The Solaris Operating Environment supports three primary shells:

- Bourne shell
- C shell
- Korn shell

The Solaris 8 Operating Environment also supports the following shells:

- Bash shell — GNU Bourne-Again Shell is a Bourne-compatible shell that incorporates useful features from the Korn and C shells.
- Z shell — The Z shell most closely resembles the Korn shell but includes many other enhancements.
- TC shell — The TC shell is a completely compatible version of the Berkeley UNIX C Shell with additional enhancements.

Note – Examples given in this course are geared toward the user; therefore, they are based on the Korn shell.

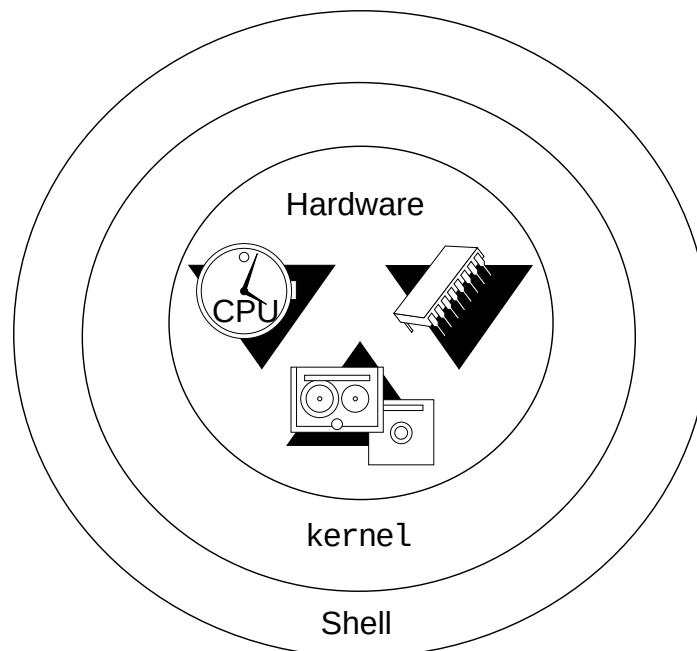


Figure 1-3 The Role of the Shell

The Bourne Shell

The Bourne shell is the original UNIX shell, developed by Steve Bourne at AT&T Bell Laboratories.

It is the default shell for the Solaris Operating Environment and the shell commonly used to administer the system.

The default Bourne shell prompt is a dollar sign (\$).

The C Shell

The C shell was developed by Bill Joy at the University of California at Berkeley.

It is based on the C programming language and has a number of features, such as command line history, aliasing, and job control. This shell has been favored over the Bourne shell by ordinary system users.

The default C shell prompt is a host name and a percent sign (hostname%).

The Korn Shell

The Korn shell is a superset of the Bourne shell, developed by David Korn at AT&T. This shell had a number of features added to it, beyond the enhancements of the C shell.

Additionally, the Bourne shell is almost completely upwardly compatible with the Korn shell, so older Bourne shell scripts can run in this shell.

The Korn shell is considered the most efficient shell and is recommended as the standard shell for ordinary system users.

The default Korn shell prompt is a dollar sign (\$).

Exercise: Using the Solaris 8 Operating Environment



Exercise objective – This exercise helps you review the basics of the computing environment.

Tasks

Answer the following questions:

1. List the four main hardware components of a computer.

2. Name the three main components of the SunOS operating system.

3. List the three primary shells supported by the Solaris Operating Environment.

4. Match the following terms with their definitions:

__ Shell	a. Common Desktop Environment
__ kernel	b. Command interpreter
__ CDE	c. Core of the Solaris Operating Environment

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

1. List the four main components of a computer.
 - ▼ RAM
 - ▼ CPU
 - ▼ I/O
 - ▼ Hard disk
2. Name the three main components of the SunOS operating system.
 - ▼ The kernel
 - ▼ The shell
 - ▼ The directory tree
3. List the three primary shells supported by the Solaris Operating Environment.
 - ▼ Bourne Shell
 - ▼ C Shell
 - ▼ Korn Shell
4. Match the following terms with their definitions:

b	Shell	a. Core of the Solaris Operating Environment
a	kernel	b. Command interpreter
c	CDE	c. Common Desktop Environment

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ List the four main hardware components of a computer
- ☐ Describe the four main components of the Solaris Operating Environment
- ☐ State the three main components of the SunOS operating system
- ☐ Identify the default shells available in the Solaris Operating Environment
- ☐ Understand the basics of the Common Desktop Environment (CDE)

Objectives

Upon completion of this module, you should be able to:

- List the characteristics of an effective password
- Log in and log out of a system from the command line
- Log in and log out of a CDE session
- Execute basic commands
- Change your password
- Identify and describe the components of a command line
- Use control characters to erase a command line, stop the execution of a command, and stop and start screen output
- Display online manual pages
- Search the online manual pages by keyword
- Identify users logged on to the system using the commands `who`, `who am i`, and `id`
- Enter multiple commands on a single command line

Additional Resources



Additional resources – The following references can provide additional details on the topics discussed in this module:

- *Solaris Common Desktop Environment: User's Guide*,
"Starting a Desktop Session," Part Number 806-1360-10
- *System Administration Guide, Volume 1*, Part Number 805-7228-10

User Accounts

Every user must have a *user account* on the system to log in. All user accounts are defined in the `/etc/passwd` file and contain the elements that identify each unique user to the system.

System administrators are responsible for creating and maintaining user accounts.

The root Account

The `root` account and password are set up during the Solaris Operating Environment installation process. This login account is used by the system administrator to perform specific administration tasks on the system.

The `/etc/passwd` Entry

Each user account entry in the `/etc/passwd` file contains seven fields, each separated by a colon.

Placeholder	UID	Comment
user1	x:102:10	User Account 1
1	/export/home/user1	/bin/ksh

user1:x:102:10:User Account 1:/export/home/user1:/bin/ksh

Login ID GID Login directory Default shell

Figure 2-1 Example `/etc/passwd` Entry

- **Login ID** — Contains the identification required by the system for a successful log in. Depending on the system administrator, login IDs are usually some combination of a user's first and last names. For example, user Bob Wood's login ID might be `bobw`, `bwood`, or `woodb`.

The login ID must be:

- ▼ Limited to eight alphanumeric characters
- ▼ Entered in lowercase characters
- ▼ Unique
- Placeholder — Maintains the field for the password, which is kept in the `/etc/shadow` file. The `/etc/shadow` file contains encrypted passwords and password aging information (for example, how long before a user must change a password and how long a password must be in use before it can be changed). This file can be read only by the system administrator.
- UID — Identifies the user's unique numerical ID (UID).
- GID — Identifies the group's unique numerical ID (GID) within the system.
- Comment — Is traditionally the full name of the user.
- Login directory — Is referred to as the home directory and is where users create and store their personal files.
- Default shell — Defines shell the users will be working in once they log in to the system.

Logging In

The log in process identifies a user to the system.

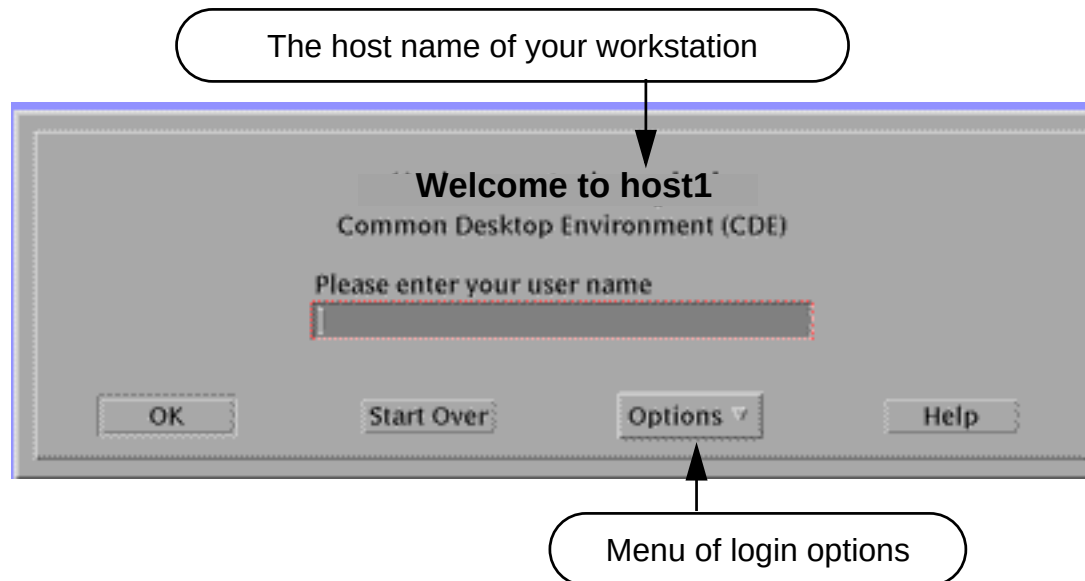


Figure 2-2 The Login Screen

The Login screen, displayed by the CDE Login Manager, is your entry to the desktop. It provides a place for you to enter your login ID and password.

As an alternative to the CDE Login screen, you can log in from a command line by selecting that option from the Options menu.

The Options Button

When you select the options button, on the login screen the Options menu on lists your login options.

- Options
- Language
- Session
 - Common Desktop Environment (CDE)
 - OpenWindows Desktop
 - User's Last Desktop
 - Failsafe Session
- Remote Login
 - Enter Host Name
 - Choose Host From List
- Command Line Login
- Reset Login Screen

Language

The Options menu enables you to select a particular language for your session. The default language for the system is set by the system administrator.

Session

You can also select which desktop environment to work in (for example, CDE or OpenWindows).

The Failsafe session opens a single terminal window on the workstation screen; instead of opening a desktop session. This is useful if you need access to only a single terminal window to quickly execute some commands, without having to login and wait for the desktop session. To log out of a Failsafe session, execute the `exit` command.

You can consider the Failsafe session a backup method for logging in should other options not work.

Remote Login

The Remote Login option enables you to connect to a remote system to start a remote Desktop login. This option allows you to either enter the specific host name of a remote system or to select from a list of available remote systems.

Command-line Login

The command-line login enables you to work in the more traditional non-GUI environment. This mode is not a desktop session. When the system is in command-line login, the desktop is suspended.

When you log out from a command-line prompt, the CDE Login screen is restarted.

Reset Login Screen

The Reset Login Screen option restarts the CDE Login Screen.

Logging In Using the Login Screen

To log in to a Desktop Session from the CDE login screen:

1. Enter your login ID (user name) in the text field, then press the Return key or click on the OK button.
2. You must then enter your password in the password text field and press the Return key or click on the OK button.

If the Login Manager does not recognize your user name or password, click on Start Over and start the log in process again.

Logging In Using the Command Line

To log in to a command-line session:

1. Display the Options pull-down menu and select Command Line Login. The Login screen disappears and is replaced by a console prompt.
2. Press the Return key to get a prompt for a user name entry.

Note – If you select the Command Line Login option, press the Return key. Otherwise, the CDE login screen will automatically restart.

3. Enter your user name (or login ID) at the prompt and press the Return key.
4. Enter your password at the prompt and press the Return key. The password does not appear on the screen when it is entered.

Note – By default, if a user does not have a password, then they are automatically prompted to enter a new password during the initial login.

Password Requirements

Passwords protect user accounts from unauthorized access. In the Solaris Operating Environment, a user's password:

- Must be six to eight characters in length
- Should contain at least two alphabetic characters and must contain at least one numeric or special character
- Must differ from the login ID name
- Must differ from the previous password by at least three characters
- Can contain spaces

These password requirements do not apply to the system administrator's `root` account password.

Changing Your Password

Frequently changing user passwords helps prevent unauthorized access to the system.

Changing Your Password in CDE

To change your password in a CDE session:

1. From the desktop, open a terminal window.
2. Execute the `passwd` command at the shell prompt and press the Return key.
3. When the prompt `Enter login password:` appears, enter the current password and press the Return key.
4. When the prompt `New password:` appears, enter the new password and press the Return key.
5. When prompted, reenter the new password and press the Return key. The system requires this to verify the new password.

Changing Your Password from the Command Line

To change your password from the command line:

1. Execute the `passwd` command at the prompt.
2. Enter the current password.
3. Enter the new password.
4. Re-enter the new password for verification. The shell prompt is redisplayed. For example:

```
$ passwd
passwd: Changing password for user1
Enter login password:
New password:
Re-enter new passwd:
passwd (SYSTEM): passwd successfully changed for user1
$
```


Securing the CDE Session

Securing your CDE session prevents unauthorized users from gaining access to the system. The two ways to secure the system are:

- Locking the screen
- Exiting the session

Locking the Screen

Locking the screen prevents unauthorized users from gaining access to your CDE session, while keeping your session intact.

The padlock icon on the Front Panel is used to secure the screen and apply password protection. To regain access to your CDE session, enter your password and press the Return key.

Exiting the Session

Two ways to exit the CDE session include:

- Using the Exit Button
- Using the log out option from the Workspace Menu

Exiting Using the Exit Button

The Exit Button, on the Front Panel, is used to log out from a CDE session.

By default, a Logout Confirmation window is displayed. To confirm the log out process, click on the OK button, or press the Return key when the OK button is highlighted.

The home CDE session is saved automatically so you can return to the same workspace windows at your next CDE session.



Caution – Any data contained in the current set of open windows will be lost. Be sure to save all data before exiting from a CDE session.

Exiting Using the Workspace Menu

To log out using the Workspace Menu, right click on the desktop area and select the Logout option from the Workspace Menu.

By default, a Logout Confirmation window is displayed. To confirm the log out process, click on the OK button, or press the Return key when the OK button is highlighted.

The home CDE session is saved automatically so the you can return to the same workspace windows at your next CDE session.



Caution – Any data contained in the current set of open windows is be lost. Be sure to save all data before exiting from a CDE session.

Basic UNIX Commands

Once you log in to the system, open a terminal window. A Korn shell prompt appears at the start of the command line, indicating that the shell is ready to receive a command.

Note – The default shell used in this course is the Korn shell.

For both the Bourne and Korn shells, the default shell prompt is a dollar sign (\$). For the C shell, the prompt is a host name and a percent sign (hostname%).

Using the uname Command

The `uname` command lists information about the system. By default, entering this command displays the name of the current operating system.

Displaying the Operating System Name

To display the operating system information, execute the following:

```
$ uname
SunOS
$
```

Using the date Command

The `date` command displays the system's current date and time.

Displaying the Date and Time

To display the date and time, execute the following:

```
$ date
Mon Feb 25 12:55:29 MST 2000
$
```

Using the cal Command

The `cal` command displays a calendar for the current month and year.

Displaying the Calendar

To display the calendar, execute the following:

```
$ cal
      February 2000
S  M Tu  W Th  F  S
      1  2  3  4  5
6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29
```

Command-Line Syntax

You can enhance the capability of commands by the using options and arguments. The basic syntax of a UNIX command includes:

`$ command option(s) argument(s)`

command Executable (specifies *what* the system is to do)

option Modify the command (this specifies *how* the command is run)

argument A file name, directory name, or text

The following are some examples of commands and commands using options and arguments.

```
$ cal 2000                    (Command and argument)
$ uname -rps                (Command and multiple options)
$ cal 12 1999               (Command and two arguments)
$ banner "hi there"        (Command and argument)
$ ls                        (Command)
$ ls -l                    (Command and option)
$ ls directory_name       (Command and argument)
$ ls -l directory_name (Command, option, and argument)
```

Control Characters

Through the use of special control characters, you stop and start screen output, erase an entire command line, or stop the execution of a command from the keyboard.

To enter a sequence of control characters, hold down the Control key and press the appropriate character on the keyboard for the desired action.

Table 2-1 describes the control characters you can use.

Table 2-1 Control Characters

Control Characters	Purpose
Control-c	Terminates a command
Control-d	Indicates end-of-file or exit
Control-q	Resumes screen output if the screen is locked
Control-s	Stops screen output
Control-u	Erases the command line
Control-w	Erases the last word on the command line

Viewing Online Documentation

The online UNIX Reference Manuals (also called *man pages*) provide detailed descriptions about commands and their usage. These online man pages are included in the Solaris Operating Environment.

The `man` command is primarily used to display the online manual page for any given command.

Command Format

```
$ man command_name
```

Using the `man` Command without Options

You can invoke the `man` command without options. For example:

```
$ man uname
Reformatting page.  Please Wait... done

User Commands                                     uname(1)

NAME
  uname - print name of current system

SYNOPSIS
  uname [ -aimnprsvX ]

  uname [ -S system_name ]

DESCRIPTION
  The uname utility prints information about the current sys-
  tem on the standard output. When options are specified, sym-
  bols representing one or more system characteristics will be
  written to the standard output. If no options are specified,
  uname prints the current operating system's name. The
  options print selected information returned by uname(2),
  sysinfo(2), or both.

<output omitted>
```

Scrolling in Man Pages

Table 2-2 shows the keys used to control the scrolling capabilities while using the `man` command:

Table 2-2 Keys to Control Scrolling in Man Pages

Key	Action
Spacebar	Displays the next screen of a man page
Return	Scrolls through a man page one line at a time
b	Moves back one screen
f	Moves forward one screen
q	Quits the <code>man</code> command
<i>/pattern</i>	Searches forward for this <i>pattern</i>
n	Finds the next occurrence of <i>pattern</i>
h	Gives a description of all scrolling capabilities

Searching Man Pages by Section

There are many different sections contained in the man pages. The **SEE ALSO** section at the bottom of a man page lists alternative references that pertain to the topic addressed. When these references are followed by a number in parentheses, it indicates a section of the man pages that you can access using the `-s` option with the `man` command.

For example, executing `$ man passwd` displays information on the `passwd` command and gives instructions on how to change a password. The **SEE ALSO** section of this man page reads in part as follows:

SEE ALSO

```
finger(1), login(1), nispasswd(1), nistbladm(1),
yppasswd(1), domainname(1M), eeprom(1M), id(1M),
passmgmt(1M), pwconv(1M), su(1M), useradd(1M), userdel(1M),
```

Executing `$ man -s4 passwd` displays information on the `/etc/passwd` file and `$ man -s3C crypt` displays information on the password encryption process.

Using the man Command with the -k Option

You can invoke the `man` command with the `-k` option for a keyword lookup to display a list of the commands that might be relevant.

```
$ man -k keyword
```

Searching Man Pages by Keyword

When you are not sure of the name for a command, you can use the `-k` option with the `man` command to specify a keyword as a subject.

```
$ man -k calendar
cal                cal (1)          - display a calendar
calendar           calendar (1)    - reminder service
difftime           difftime (3c)   - computes the difference between two
                                calendar times
mktime             mktime (3c)     - converts a tm structure to a calendar
                                time
$
```

Determining Current Users

The `who` command displays a list of users currently logged in to the local system, with their login name, login TTY port, login date and time, and the elapsed time since their last activity. When a user is logged in remotely, the remote system name displays for that user.

Command Format

`who`

Displaying Users on the System

To display the users on the system, execute:

```
$ who
user1      console      Feb 25 13:50    (:0)
user1      pts/4         Feb 25 14:37    (:0.0)
user1      pts/6         Feb 25 14:54    (:0.0)
user1      pts/7         Feb 25 15:24    (:0.0)
```

Identifying the User Name

Use the `who am i` command to identify the user name.

Command Format

```
who am i
```

Example

To display the user name, execute the following:

```
$ who am i  
user1      pts/7 Feb 25 15:24    (:0.0)  
$
```

Identifying User Group Details

Use the `id` command to identify the user ID, user name, group ID, and group name of a system user.

Command Format

```
id username
```

Identifying a User

To identify a user, execute the following:

```
$ id
uid=11001(user1) gid=10(staff)
$
```

Entering Multiple Commands from a Single Command Line

The semicolon (;) is a special character often referred to as the *command separator*.

The semicolon enables you to enter multiple commands on a single command line. The shell executes each command from left to right when the Return key is pressed.

An example of the semicolon follows:

```
$ cd;ls
dante      dir2      file.1     file1      file4      practice
dante_1    dir3      file.2     file2      fruit      tutor.vi
dir1       dir4      file.3     file3      fruit2
$
```

```
$ date;cal;pwd
Thu Feb 17 16:49:34 MST 2000
    February 2000
S  M Tu  W Th  F  S
      1  2  3  4  5
6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29
/export/home/user1
$
```

Exercise: Accessing the System



Exercise objective – In this exercise, you practice logging in using the command line, changing your password, executing commands, and logging out.

Tasks

Complete the following steps:

1. Obtain a user name and password from your instructor.
2. Log in to the system using the CDE Login Manager login screen.
 - a. Enter your user name and press the Return key.
 - b. Enter your password and press the Return key.
 - c. Select CDE by clicking on OK or pressing the Return key.
3. On the CDE Desktop, right click on the background screen. The Workspace menu is displayed. Select Tools from this menu.
4. The Tools menu displayed. Select Terminal from this menu. A terminal window is displayed.
5. Use the mouse to move the cursor into the terminal window.
6. Using the `passwd` command, change your password to `mypass1`.

\$ passwd

Enter login password: (This is your original password)

New password:

Re-enter new password:

`passwd (SYSTEM) passwd successfully changed for username`

7. Exit the CDE desktop by clicking on the Exit button, located on the Front Panel.
8. A Logout Confirmation window is displayed. Click on OK or press the Return key to continue to log out.

9. At the CDE Login Manager screen, enter the following, incorrect login name and password:

▼ Login Name: **nosuchuser**

▼ Password: **wrong**

The message Login incorrect; please try again is displayed.

10. Click on OK.
11. When the CDE Login Manager screen is displayed, click on Options and select the Reset Login screen.

Logging in Using the Command Line Login Option

1. When the CDE Login Manager screen displays, click on Options and select Command Line Login.
2. When the following message appears, press the Return key to display the console login prompt.

```
*****
```

```
*Suspending Desktop Login ...
```

```
*
```

```
*If currently logged out, press [Enter] for a console *login prompt.
```

```
*
```

```
*Desktop Login will resume shortly after you exit console *session.
```

```
*****
```

```
<hostname> console login:
```

3. Enter your user name and press the Return key.
4. Enter your new password and press the Return key.
5. Press the Control key and the C key simultaneously when the following message appears.

```
Last login <date> <time> on console
Sun Microsystems Inc. SunOS 5.8 Generic February 2000
```

A ^C\$ appears on the console screen. The ^C is the Control+C key sequence echoed to the screen. The dollar sign (\$) is your shell prompt.

6. At the shell prompt, type the exit command.

7. The `<hostname>` console login: prompt reappears. After a few seconds, the CDE Login Manager screen is displayed.

Invoking a Failsafe Session

8. When the CDE Login Manager screen is displayed, click on Options, select Session, then select Failsafe Session.
9. Enter your user name and press the Return key.
10. Enter your password and press the Return key. A single terminal window is displayed.
11. Using the mouse, move the cursor into the terminal window.
12. Execute the `exit` command.

Invoking the User's Last Desktop Session

1. At the CDE login screen, enter your user name and password and start a CDE Desktop session.
2. On the CDE Desktop, using the right mouse button and click on the background screen.
3. Open a terminal window.
4. Display the date and time using the `date` command.
Display the date and time using the `date -u` command.
What is the difference in the information displayed on the screen?

5. Display this month's calendar? How was this command entered on the command line?

6. Display what user name you entered to log in to the system? What command did you use to display this information?

-
7. What command displays information about other users currently logged onto your workstation?
- _____
8. What command can be used to discover the user name, UID, group name, and GID of any user on the workstation?
- _____
- a. Enter the `id` command.
What information displays?
- _____
- b. Enter the `id -a root` command.
What information displays?
- _____
9. To display information about the operating system and the workstation name, what command could be used?
- _____
- a. Enter the `uname -s` command.
What information displays? _____
- b. Enter the `uname -r` command.
What information displays? _____
- c. Enter the `uname -n` command.
What information displays? _____
- d. Enter the `uname -srn` command.
What information displays? _____
10. Display the online manual pages for the `passwd` command first, then the `passwd` file.
- a. Execute the `man passwd` command.
The `man` command moves to Section 1 of the online manual pages to locate information on the `passwd(1)` command.
- b. Move to the end of this online manual page by typing: `/SEE`.
The forward slash (/) character and the word SEE are a search string.

Under the section SEE ALSO, locate the entry: `passwd(4)`. This indicates information on the `passwd` file is located in Section 4 of the online manual pages.

- c. Execute the `man -s4 passwd` command.
The `man` command moves to Section 4 of the online manual pages to locate information on the `passwd` file.

11. Using the `-k` option with `man` command, find the online manual page that describes how to clear a terminal window (use the keyword “clear”). How would this command be entered on the command line?
-

12. Clear the terminal window.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

Invoking the User's Last Desktop Session

4. Display the date and time using the **date** command.
Display the date and time using the **date -u** command.
What is the difference in the information displayed on the screen?

The **date** command displays the current date and time to the terminal window.

The **date -u** command displays the date and time in Greenwich Mean Time (GMT-universal time) to the terminal window.

5. Display this month's calendar? How was this command entered on the command line?

cal

6. Display the user name you entered to log in to the system? What command did you use to display this information?

who am i

7. What command displays information about other users currently logged onto your workstation?

who

8. What command can be used to discover the user name, UID, group name, and GID of any user on the workstation?

id or **id username**

- a. Enter the **id** command.
What information is displayed?
Your UID number, current user name, GID number, and group name.
- b. Enter the **id -a root** command.
What information is displayed?
The user's UID number, user name, GID numbers and group names for all groups to which this user belongs.

9. To display information about the operating system and the workstation name, what command could be used?

uname -a or uname -ns

- a. Enter the `uname -s` command.
What information is displayed? The operating system name, for example, SunOS.
 - b. Enter the `uname -r` command.
What information is displayed?
The operating system version, for example, 5.8.
 - c. Enter the `uname -n` command.
What information is displayed?
The host name of the workstation.
 - d. Enter the `uname -srn` command.
What information is displayed?
The operating system name, host name, and version number.
11. Using the `-k` option with `man` command, find the online manual page that describes how to clear a terminal window (use the keyword “clear”). How would this command be entered on the command line?

man -k clear

12. Clear the terminal window.

clear

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ List the characteristics of an effective password
- ☐ Log in and log out of a system from the command line
- ☐ Log in and log out of a CDE session
- ☐ Execute basic commands
- ☐ Change your password
- ☐ Identify and describe the parts of a command line
- ☐ Use control characters to erase a command line, stop the execution of a command, and stop and start screen output
- ☐ Display online manual pages
- ☐ Search the online manual pages by keyword
- ☐ Identify users logged on to the system using the commands `who`, `who am i`, and `id`
- ☐ Enter multiple commands on a single command line

Objectives

Upon completion of this module, you should be able to:

- Demonstrate the difference between absolute and relative pathnames
- Access files and directories within the directory tree using absolute and relative pathnames
- Use pathname abbreviations to access files and directories within the directory tree
- List the contents of directories and determine file types
- Identify various shell metacharacters to abbreviate file names and pathnames

Additional Resources



Additional resources – The following reference can provide additional details on the topics discussed in this module:

- *Solaris User's Guide*, Part Number 802-6499

The Directory Tree

Figure 3-1 illustrates a portion of a sample Solaris directory tree, showing the placement of a particular user's files and directories.

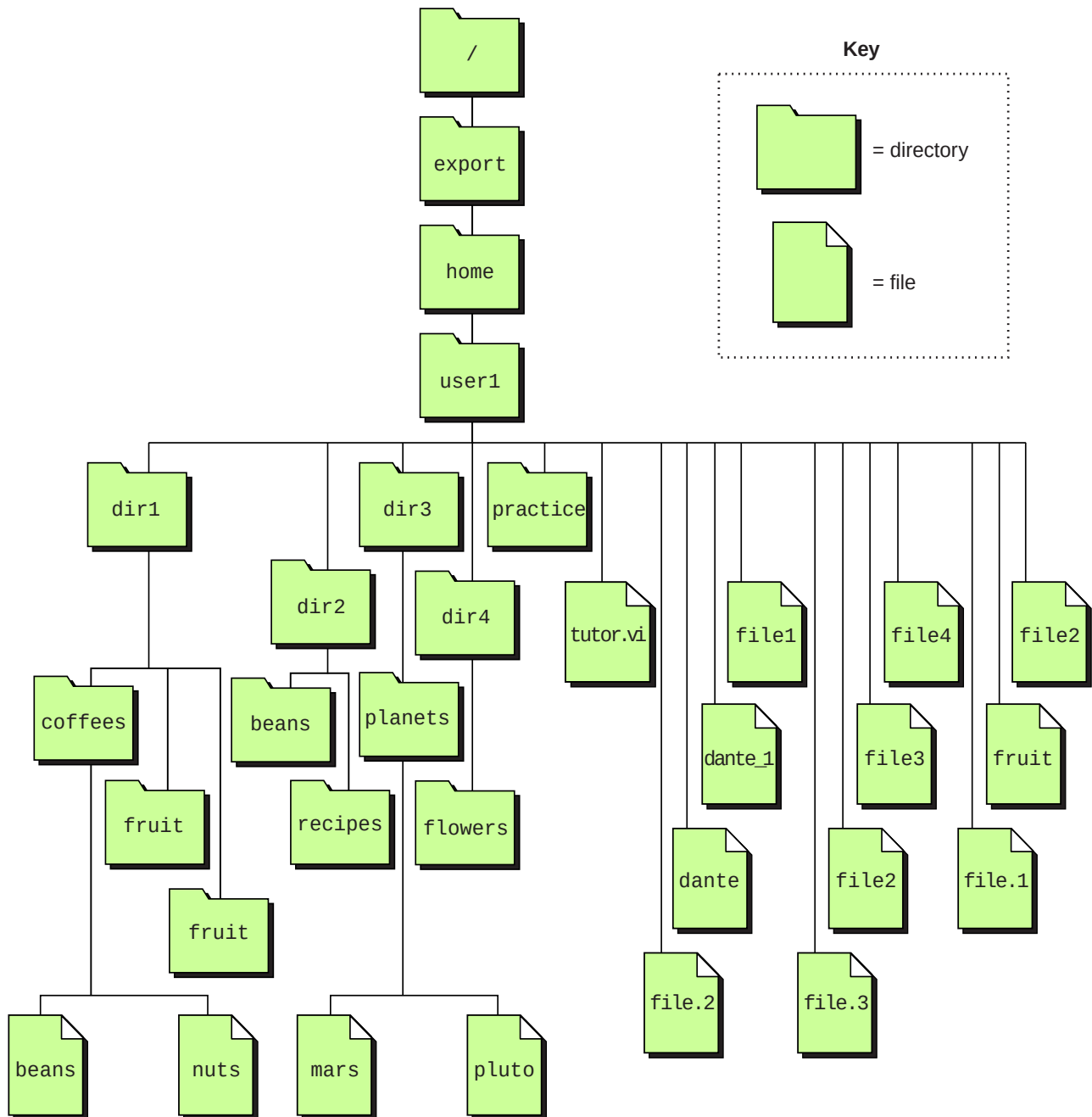


Figure 3-1 Solaris Directory Tree

Pathnames

A *pathname* uniquely identifies a particular file or directory by specifying its location in the directory tree. Pathnames are similar to a road map which shows how to get from one place in the directory tree to another.

The slashes (/) within a pathname are delimiters between object names. An object name can be a directory name or a file name. The first slash in a pathname always represents the root (/) directory. For example:

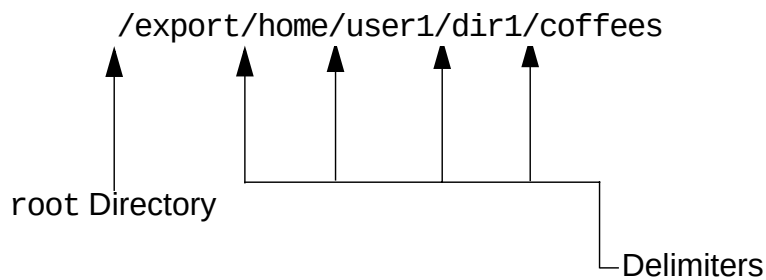


Figure 3-2 Pathname with Delimiters

Note – Depending on the system setup, your home directory can be located in either the `/home` directory or the `/export/home` directory.

Pathname Types

There are two types of pathnames: absolute and relative.

Absolute Pathname

An *absolute pathname* specifies a file or directory in relation to the entire Solaris directory tree. Absolute pathnames always:

- Start at the root (/) directory, then list each directory along the path to the final destination, which can be a file name or another directory
- Use a slash (/) between each object name in the path. The slashes (/) within a pathname are delimiters between object names. An object name can be a directory name or a file name.

Note – An absolute pathname can also be referred to as a *full pathname*.

Refer to Figure 3-1 for a visual representation of the following pathnames:

- Absolute pathname to the `user1` directory
`/export/home/user1`
- Absolute pathname to the `dir1` directory
`/export/home/user1/dir1`
- Absolute pathname to the `coffees` directory
`/export/home/user1/dir1/coffees`

Relative Pathname

A *relative pathname* describes the location of a directory or a file as it relates to the current directory.

A relative pathname does not ever begin with a slash (/) character. However, it does use slashes (/) within the pathname as delimiters between object names, (for example, directory name or file name).

If you are in a directory and you want to move down to access another directory in the hierarchy, you do not have to enter an absolute pathname. Simply enter the path starting with the name of the next directory down in the tree structure.

Refer to Figure 3-1 for a visual representation of the following examples:

- The current directory is:
`/export/home`
- From `/export/home`, the relative pathname to access the `user1` directory is:
`user1`
- From `/export/home`, the relative pathname to access `dir1` is:
`user1/dir1`
- From `/export/home`, the relative pathname to access the `coffees` directory is:
`user1/dir1/coffees`

File and Directory Naming Conventions

When creating files and directories, it is important that you use the following conventions:

- Directory and file names can contain up to a maximum of 255 alphanumeric characters.
- Non-alphanumeric characters, such as underscores (_), hyphens (-), and periods (.), are allowed in file and directory names.
- Special characters, such as asterisks (*), ampersands (&), pipes (|), quotes (" "), and dollar signs (\$) should not be used. These particular characters hold special meaning to the shell.
- Directory names, as a rule, do not contain extensions; however, you can use extensions with directory names.

Changing Directories

At any given time, you are located in current working directory within the directory tree. When you initially log in to the system, the current directory is set to your **home** directory.

You can change your current working directory at any time by using the **cd** command.

Command Format

```
cd directory_name
```

Moving Around the Directory Tree

The following examples show how to change directories within the directory tree:

- Using an absolute pathname:

```
$ cd /export/home/user1/dir1/coffees
```

- Using a relative pathname:

```
$ cd user1/dir1/coffees
```

You can always return to your own **home** directory by typing the **cd** command without an argument. For example:

```
$ cd
```

Displaying the Current Directory

The **pwd** command, or print working directory command, determines in which directory you are currently working.

The **pwd** command displays the absolute pathname of the current working directory.

Command Format

pwd

Determining the Current Working Directory

To determine the current working directory, execute the following:

```
$ pwd
/export/home/user1
$ cd /export/home
$ pwd
/export/home
$ cd
$ cd practice
$ pwd
/export/home/user1/practice
$
```

Changing Directories with Pathname Abbreviations

Pathname abbreviations are used as a quick method for moving to or referring to directories on the command line.

Table 3-1 Pathname Abbreviations

Symbol	Meaning
.	Current or working directory
..	Parent directory; the directory directly above the current working directory

The examples below show how to use the `cd` command to give pathname abbreviations to move around the Solaris directory tree.

```
$ pwd
/export/home/user1/dir1
$ cd ..
$ pwd
/export/home/user1
$ cd ../../..
$ pwd
/
$
```

Displaying the Contents of a Directory

To display the contents of a directory, use the `ls` command. This command lists the files and directories within the specified directory.

Using the `ls` command with no argument simply displays the contents of the current directory.

Command Format

```
ls -option(s) pathname(s)
```

Listing the Contents of a Directory

To list the contents of a directory, execute the following:

```
$ cd
$ pwd
/export/home/user1
$ ls
dante      dir2      file.1    file1     file4     practice
dante_1    dir3      file.2    file2     fruit     tutor.vi
dir1       dir4      file.3    file3     fruit2
$ ls dir1
coffees    fruit     trees
$ ls /var/mail
:saved user1
$
```

Displaying Hidden Files

File names that begin with a period (.) are called *hidden files*. Hidden files are frequently used to customize your work environment.

Use `ls -a` to list all files in a directory, including any hidden (.) files.

To display hidden files, execute the following:

```
$ pwd
/export/home/user1
$ ls -a
.          .dtprofile  dir1       file.2     file4
..         .sh_history dir2       file.3     fruit
.TTauthority .solregis  dir3       file1      fruit2
.Xauthority dante     dir4       file2      practice
.dt        dante_1    file.1     file3      tutor.vi
$
```

Displaying File Types

Use `ls -F` to display file types. Use the symbols in Table 3-2 to display the various file types:

Table 3-2 File Type Symbols

File Type	Symbol
Directory	/
Executable	*
Plain text file/ASCII	(none)
Symbolic link	@

Execute the following to show a file type:

```
$ pwd
/export/home/user1
$
$ ls -F
dante    dir2/    file.1   file1    file4    practice/
dante_1  dir3/    file.2   file2    fruit    tutor.vi
dir1/    dir4/    file.3   file3    fruit2
$
```

Note – A *symbolic link* is a special type of file. It contains an absolute pathname of the file to which it is linked to which points it.

Displaying a Long Listing

To see detailed information about the contents of a directory, use the `ls -l` command.

To get detailed information using the `ls -l` command, execute the following:

```
$ ls -l
total 88
-rw-r--r--  1 user1  staff    1320 Feb 22 14:51 dante
-rw-r--r--  1 user1  staff    368 Feb 22 14:51 dante_1
drwxr-xr-x  5 user1  staff    512 Feb 22 14:51 dir1
drwxr-xr-x  4 user1  staff    512 Feb 22 14:51 dir2
drwxr-xr-x  3 user1  staff    512 Feb 22 14:51 dir3
drwxr-xr-x  3 user1  staff    512 Feb 22 14:51 dir4
-rw-r--r--  1 user1  staff      0 Feb 25 12:54 file.1
-rw-r--r--  1 user1  staff      0 Feb 25 12:54 file.2
-rw-r--r--  1 user1  staff      0 Feb 25 12:54 file.3
-rw-r--r--  1 user1  staff   1696 Feb 22 14:51 file1
-rw-r--r--  1 user1  staff    105 Feb 22 14:51 file2
-rw-r--r--  1 user1  staff    218 Feb 22 14:51 file3
-rw-r--r--  1 user1  staff    137 Feb 22 14:51 file4
-rw-r--r--  1 user1  staff     56 Feb 22 14:51 fruit
-rw-r--r--  1 user1  staff     57 Feb 22 14:51 fruit2
drwxr-xr-x  2 user1  staff    512 Feb 22 14:51 practice
-rw-r--r--  1 user1  staff  28738 Feb 22 14:51 tutor.vi
$
```

The `ls -l` command provides the file information shown in Figure 3-3.

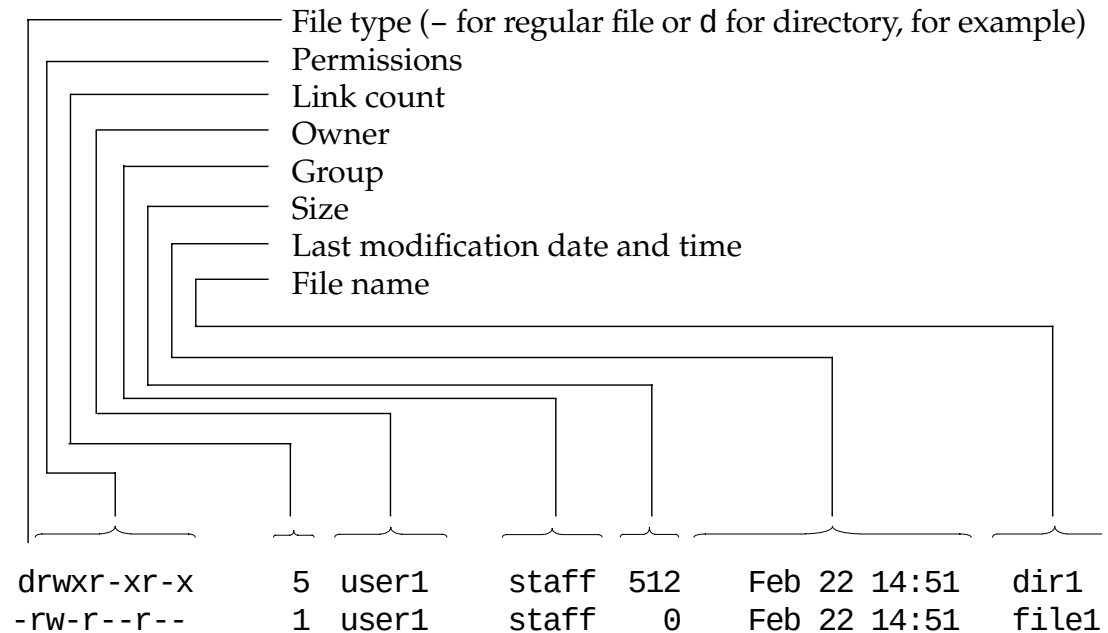


Figure 3-3 Long Listing File Information

Listing Individual Directories

Use `ls -ld` to display detailed information for the current working directory only, not its contents.

To obtain detailed directory information, execute the following:

```
$ cd
$ ls -l dir1
total 6
drwxr-xr-x 2 user1 staff 512 Feb 22 14:51 coffees
drwxr-xr-x 2 user1 staff 512 Feb 22 14:51 fruit
drwxr-xr-x 2 user1 staff 512 Feb 22 14:51 trees
$ ls -ld dir1
drwxr-xr-x 5 user1 staff 512 Feb 22 14:51 dir1
$
```

Use `ls -R` to display the contents of a directory and all of its subdirectories.

This is also known as a *recursive* listing.

To display a recursive listing, execute the following:

```
$ pwd
/export/home/user1
$ ls -R dir1
dir1:
coffees  fruit   trees

dir1/coffees:
beans          beans.backup  nuts

dir1/fruit:

dir1/trees:
$
```

To display a listing sorted by the file's time stamp, with the latest file first, execute the following

```
$ ls -t
file.1          script6          file3
dante           file5            file4
script1         dir1             fruit
script2         dir2             fruit2
file.txt        dir3             practice
public_directory dir4             tutor.vi
shared_directory file2            dante_1
$
```

To display a listing showing the latest file modification time in reverse order, execute the following:

```
$ ls -tr
dante_1          file2            shared_directory
tutor.vi         dir4             public_directory
practice         dir3             file.txt
fruit2           dir2             script2
fruit            dir1             script1
file4            file5            dante
file3            script6          file.1
$
```

Metacharacters

Metacharacters are specific characters, generally symbols, used to represent a special meaning to the shell. Some examples of metacharacters include:

~ - + * ? [] ; \$ < > |

Using the Tilde (~) Character

The shell substitutes the tilde (~) character with the home directory of the current user. It is an abbreviation of the absolute pathname. For example:

```
$ cd /etc
$ pwd
/etc
$ cd ~/dir1
$ pwd
/export/home/user1/dir1/
$
```

Note – The tilde (~) character is available in all shells except the Bourne shell.

Using ~username

Attaching a user name to the tilde (~) character refers to the home directory of the specified user (login ID). For example:

```
$ cd ~user2
$ pwd
/export/home/user2
$
```

Using ~+ and ~-

The tilde and plus symbols (~+) refer to the current working directory. The tilde and dash symbols (~-) refer to the previous working directory.

For example:

```
$ pwd
/export/home/user1
$ cd dir2
$ ls ~+
beans notes recipes
$ cd ~-
$ pwd
/export/home/user1
$ cd ~-
$ pwd
/export/home/user1/dir2
$
```

Using the Dash

To switch quickly between two specific directories, use the Korn shell dash (-) symbol. The Korn shell automatically displays the current directory path with this particular option.

```
$ pwd
/export/home/user1
$ cd /tmp
$ pwd
/tmp
$ cd -
/export/home/user1
$ cd -
/tmp
$
```

Using the Asterisk

The asterisk (*) represents zero or more characters, excluding the leading period (.) on a hidden file. The asterisk is often referred to as a *wildcard* character.

For example:

```
$ ls f*
file.1  file2    file4    fruit
file.txt file3    file5    fruit2
$
```

```
$ ls d*
dante    dante_1
```

```
dir1:
coffees  fruit    trees
```

```
dir2:
beans    notes    recipes
```

```
dir3:
planets
```

```
dir4:
flowers
$
```

```
$ ls *3
file3
```

```
dir3:
planets
$
```

Using the Question Mark

The question mark (?) matches any single character, excluding the leading period (.) on a hidden file.

For example:

```
$ ls dir?
```

```
dir1:
```

```
coffees  fruit      trees
```

```
dir2:
```

```
beans    notes     recipes
```

```
dir3:
```

```
planets
```

```
dir4:
```

```
flowers
```

```
$
```

The following example shows the error message that is displayed if there are no file names matching the wildcard character.

```
$ ls z?
```

```
z?: No such file or directory
```

```
$
```

Using the Square Brackets

Use square brackets ([]) to match a set or range of characters for a single character position.

When looking for a set of characters, the characters inside the brackets do not generally need to be in any order; for example, [abc] is the same as [cab].

However, when looking for a range of characters, they must be in proper order; for example, [a-z] or [0-9].

To search for all alphabetic characters, whether lowercase or uppercase, use [A-Z] or [a-z] as the pattern to match.

For example:

```
$ ls [a-f]*
dante    file.1  file.3  file2   file4   fruit2
dante_1  file.2  file1   file3   fruit

dir1:
coffees  fruit   trees

dir2:
beans    notes   recipes

dir3:
planets

dir4:
flowers
$

$ ls [af]*
file.1  file.2  file.3  file1   file2   file3   file4   fruit   fruit2
$
```

Note – You must not use these metacharacters when creating file and directory names.

Exercise: Accessing Files and Directories



Exercise objective – In this exercise, you use the commands discussed in this module to list and change directories.

Tasks

Referring to Figure 3-1 on page 3-2, identify the pathnames for the following objects:

1. Specify the absolute pathnames for:

- ▼ user1_____
- ▼ coffees_____
- ▼ dir4_____

Assume `/export/home/user1` is the current directory for the next two questions.

2. Specify the relative pathnames for:

- ▼ dir3_____
- ▼ flowers_____
- ▼ recipes_____

3. Specify the relative pathnames for the `dir1` subdirectories and files.

Using Figure 3-1 on page 3-2 as a reference, perform each of the following tasks on your system. Use pathname abbreviations whenever possible.

4. Change to your home directory.

5. Change to the `dir1` directory.

6. Change to the `fruit` directory.

7. Change to the `planets` directory.

8. Change to your home directory.

9. Change to the `/etc` directory.

10. Change to the `recipes` directory.

11. Change to the `flowers` directory.

Use the `ls` and `cd` commands to complete the following steps. Refer to Figure 3-2 on page 3-3 for steps 12 through 17 if needed.

12. Return to your home directory.

13. Change to the `dir1` directory.

14. List the contents of the `dir1` directory.

15. Display a recursive listing of the contents of the `dir2` directory.

16. Use the `ls` command to display a detailed listing of your `home` directory, including hidden files.

17. Use the `ls` option that recursively displays all contents in your `home` directory.

Is there a directory in the `root` directory (`/`) called `kernel`?

Is there a directory in `/var/spool` called `cron`?

18. Without changing directories, execute the `ls` command that displays all the file names that end with the number 1 in your `home` directory.

19. On one command line, change to your `home` directory and list the contents of the directory.

20. Issue the `ls` command that displays the file and directory names of any length beginning with the letters `d` or `f`.

21. Issue the `ls` command that displays all files starting with `file` followed by any one character.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

1. Specify the absolute pathnames for:

- ▼ user1
/export/home/user1
- ▼ coffees
/export/home/user1/dir1/coffees
- ▼ dir4
/export/home/user1/dir4

2. Specify the relative pathnames for:

- ▼ dir3
dir3
- ▼ flowers
dir4/flowers
- ▼ recipes
dir2/recipes

3. Specify the relative pathnames for the dir1 subdirectories and files.

dir1/coffees; dir1/coffees/nuts; dir1/coffees/beans
dir1/fruit; dir1/trees

4. Change to your home directory (~).

cd
or
cd ~

5. Change to the dir1 directory.

cd dir1

6. Change to the fruit directory.

```
cd fruit  
or  
cd ~/dir1/fruit
```

7. Change to the planets directory.

```
cd ~/dir3/planets  
or  
cd ../../dir3/planets
```

8. Go back to your home directory (~).

```
cd  
or  
cd ~
```

9. Change to the /etc directory.

```
cd /etc
```

10. Change to the recipes directory.

```
cd ~/dir2/recipes  
or  
cd /export/home/user1/dir2/recipes
```

11. Change to the flowers directory.

```
cd ../../dir4/flowers  
or  
cd ~/dir4/flowers
```

12. Return to your home directory.

```
cd  
or  
cd ~
```

13. Change to the dir1 directory.

```
cd dir1
```

14. List the contents of the dir1 directory.

```
ls
```

15. Display a recursive listing of the contents of the `dir2` directory.

```
ls -R ../dir2  
or  
ls -R ~/dir2
```

16. Use the `ls` command to display a detailed listing of your home directory, including hidden files.

```
ls -la ~
```

17. Use the `ls` option that recursively displays all contents in your home directory.

```
ls -R ~
```

Is there a directory in the root directory (`/`) called `kernel`?

```
Yes; ls -F /  
or  
ls -ld /kernel
```

Is there a directory in `/var/spool` called `cron`?

```
Yes; ls -F /var/spool
```

18. Without changing directories, execute the `ls` command that displays any file names that end with the number 1 in your home directory.

```
ls ~/*1
```

19. On one command line, change to your home directory and list the contents of the directory.

```
cd; ls
```

20. Issue the `ls` command that displays file and directory names of any length beginning with the letters `d` or `f`.

```
ls [df]*
```

21. Issue the `ls` command that displays all files starting with `file` followed by any one character.

```
ls file?
```

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Demonstrate the difference between absolute and relative pathnames
- ☐ Access files and directories within the directory tree using absolute and relative pathnames
- ☐ Use pathname abbreviations to access files and directories within the directory tree
- ☐ List the contents of directories and determine file types
- ☐ Identify various shell metacharacters to abbreviate file names and pathnames

Objectives

Upon completion of this module, you should be able to:

- Determine file types with the `file` command
- Display the contents of text files using the `cat`, `more`, `pg`, `head`, and `tail` commands
- Determine character, word, and line count using the `wc` command
- Create empty files or update modification times on existing files using the `touch` command
- Use the `tee` command to create text within a file
- Create and remove directories using `mkdir` and `rmdir`
- Manage files and directories using the `mv`, `cp`, and `rm` commands
- Use commands to print a file, check print queue status, and cancel a print request
- Format and print the contents of files using the `pr` command

Additional Resources



Additional resources – The following reference can provide additional details on the topics discussed in this module:

- *Solaris User's Guide*, Part Number 802-6499

Determining File Type

There are many different file types found in the Solaris Operating Environment. Use the `file` command to easily determine file type.

This information is important when you want to open or read a file. Knowing the file type helps you decide which command or program you need to use.

Command Format

```
file filename(s)
```

The output from the `file` command is most often one of the following:

- Text – Examples include ASCII text, English text, commands text, and executable shell scripts.
- Data – Data files are those that are created by an application. In some cases, the type of data file is indicated, for example, a FrameMaker document. When the `file` command cannot determine the application in which a file was created, the output simply indicates data file.
- Executable or Binary – Examples include 32-bit executable and extensible linking format (ELF) code files and other dynamically linked executables. This file type indicates that the file is a command or program.

Example Text File

The following is an example of a text file:

```
$ file dante
dante:                               English text
```

Example Data File

The following is an example of a data file:

```
$ cd /export/home/user1/dir1/coffees
$ file beans
beans:                               Frame Maker Document
```

Example Executable File

The following is an example of an executable file:

```
$ file /usr/bin/cat
/usr/bin/cat: ELF 32-bit MSB executable SPARC
Version 1, dynamically linked, stripped
```

Displaying the Contents of a Text File

The `cat` command, or concatenate command, displays the contents of a text file on the screen. It is often used to display short text files, because `cat` displays the entire contents of the file to the screen without pausing.

The `cat` command is also used to create short text files without having to use an editor.

Command Format

```
cat filename(s)
```

```
cat > filename
```

Using the `cat` Command to Display a Short Text File

To display a short text file, execute the following:

```
$ cat dante
```

```
The Life and Times of Dante  
by Dante Pocaí
```

```
Mention "Alighieri" and few may know about whom you  
are talking. Say "Dante," instead, and the whole world  
knows whom you mean. For Dante Alighieri, like Raphael,  
Michelangelo, Galileo, etc. is usually referred to by his  
first name. ...
```

```
$
```

If the contents of the file fills more than one screen, the top of file scroll off the screen. If using a scrolling window, such as a terminal window within the CDE, move the scroll bar or arrows up or down to view the entire file.

Using the cat Command to Create a Short Text File

To create a short text file, execute the following:

```
$ cat > newfile
```

You can then begin typing text in this new file. To save the contents of the file, press **Control+d** on an empty line.



Caution – If the file name already exists, the new file overwrites the existing file.

Joining Multiple Files

Use the **cat** command to join the contents of two files into a new file. For example:

```
$ cat filename1 filename2 > newfile1
```

Using the strings Command

The `strings` command finds printable strings in an object or binary file. This enables you to read text strings imbedded within a binary file that can be useful for programming.

For example:

```
$ strings /usr/bin/cat
SUNW_OST_OSCMD
usvtebn
usage: cat [ -usvtebn ] [-|file] ...
cat: Cannot stat stdout
cat: cannot open %s
cat: cannot stat %s
cat: input/output files '%s' identical
cat: close error
cat: close error
cat: close error
cat: cannot read %s:
cat: write error:
cat: mmap error
cat: no memory
cat: output error (%d/%d characters written)
cat: input error on %s:
standard input
$
```

Note – Someone with a programming background can easily interpret the output produced by the `strings` command. The command is introduced here solely as a method for demonstrating the printable characters of an executable file.

More Metacharacters

Two useful metacharacters are the redirect (>) character and the pipe (|) character.

The redirect (>) character takes a command output and directs it to a specified file.

The pipe (|) character is used on the command line to take the output of one command and feed it as input to another command.

Browsing the Contents of a File

To browse, or page, through the contents of a long text file, use the `more` command. With this command, the contents of a text file display one screen at a time, and the following message appears at the bottom of the screen:

```
--More-- (n%)
```

The `n` is the percentage of the file already displayed. When the entire file content has been displayed, the shell prompt appears.

The online man pages use the `more` utility for display purposes, so the scrolling keys in the following table are the same as those used to control the display of man pages.

Note – Using `cat` or `more` to read executable or binary files can cause a terminal or window to freeze. If this occurs, close the window and open a new window.

Command Format

```
more filename(s)
```

Scrolling Keys

At the `--More--` prompt, you can use the keys described in Table 4-1 to control scrolling capabilities.

Table 4-1 Scrolling Keys for the `more` Command

Scrolling Keys	Purpose
Spacebar	Scrolls to the next screen
Return	Scrolls one line at a time
b	Moves back one screen
f	Moves forward one screen
h	Displays a Help menu of <code>more</code> features
q	Quits and returns to the shell prompt
/string	Searches forward for <i>string</i>
n	Finds next occurrence of <i>string</i>

Note – The `more` command is used to read text files not binary files.

Viewing Long Files

The `pg` command also gives you the ability to view a file that is longer than one screen.

This command pauses after displaying a screen of text and displays the `:` prompt at the bottom of the page. Press the Return key to display another screen of text, or use one of the scrolling keys mentioned in Table 4-2.

The `pg` command displays an (EOF) `:` prompt when the end of file is reached. Press the Return key to return to the shell prompt.

Command Format

`pg filename(s)`

Scrolling Keys

Table 4-2 shows the keys used to control screen scrolling.

Table 4-2 Scrolling Keys for the `pg` Command

Scrolling Keys	Purpose
Return	Scrolls to the next screen
l <Return>	Displays the next line
d <Return>	Displays half a page more
. <Return>	Re-displays current page
+/ <i>pattern</i> / <Return>	Searches forward for <i>pattern</i>
\$ <Return>	Moves to last page
h <Return>	Displays Help menu of more features
q <Return>	Quits and returns to the shell prompt

Note – The `pg` command is used to read text files — not binary files.

Displaying the First Few Lines of a File

The `head` command enables you to view the first few lines of one or more files. The first 10 lines are displayed by default if you omit the `-n` option.

Command Format

```
head -n filename(s)
```

Displaying a Specific Number of Lines at the Beginning of a File

To display a specific number of lines at the beginning of a file, execute the following:

```
$ head -6 /usr/dict/words
10th
1st
2nd
3rd
4th
5th
$
```

In this example, the `head` command with the `-6` option displays the first six lines of the `/usr/dict/words` file.

Displaying the Last Few Lines of a File

Use the `tail` command to display the last few lines of one or more files. The last 10 lines are displayed by default if either you omit the `-n` or `+n` option.

If you use the `-n` option, the `tail` command counts relative to the end of the file. When you use the `+n` option, the `tail` command displays the number of lines relative to the beginning of the file.

Command Format

```
tail -n filename(s)
tail +n filename(s)
```

Displaying a Specified Number of Lines at the End of a File

For example, to display a specified number of lines at the end of a file, execute the following:

```
$ tail -5 /usr/dict/words
zounds
z's
zucchini
Zurich
zygote
$
```

In this example, the `tail` command with the `-5` option displays the last five lines of the `/usr/dict/words` file.

Displaying Lines From a Specific Point in the File

To display lines from a specific point in a file, execute the following:

```
$ tail +23 /usr/dict/words
<output omitted>
```

In this example, the `tail` command with the `+23` option displays lines 23 through to the end of the `/usr/dict/words` file.

Displaying Lines, Words, and Characters in a File

The `wc` command displays the number of lines, words, and characters contained in a file.

Command Format

```
wc options filename(s)
```

Using the `wc` Command with Options

You can use the following options with the `wc` command:

<code>-l</code>	Prints line count
<code>-w</code>	Prints word count
<code>-c</code>	Prints byte count
<code>-m</code>	Prints character count

Using the `wc` Command without Options

Using the `wc` command without options displays the number of lines, words, and characters contained in the file.

For example, to use the `wc` command without options, execute the following:

```
$ wc dante
33      223      1320  dante
$
```

Determining the Number of Lines in a File

To determine the number of lines in a file, execute the following:

```
$ wc -l dante
    33  dante
$
```


Creating Empty Files

The most frequent use of the `touch` command is creating an empty file. If the file name or directory name already exist, then the `touch` command simply updates its modification and access time to the current date and time.

Command Format

```
touch filename(s)
```

You can specify absolute and relative pathnames on the command line when creating new files or directories.

Creating Empty Files

To create an empty file, execute the following:

```
$ cd ~/practice
$ touch mailbox project projection research
$ ls
mailbox project projection research
$
```

Creating and Appending Files Using the tee Command

This course uses the `tee` command to create short text files.

If the file name does not exist, it is created and text is written into the file. If the file name already exists, its contents are overwritten with the new text.

Each line typed is written into the file by pressing the Return key. Once you press the Return key, the line of text is also displayed on the screen. This is notification to you that the line of text has been entered into the file.

To stop entering text and close the file, press `Control+d` on a line by itself.

Command Format

```
tee filename
```

Creating a Text File

In this example, the `tee` command is used to create a new text file that contains names and phone numbers.

```
$ cd
$ pwd
export/home/user1
$ tee phone_list.txt
Laura O. 808-555-9876
Laura O. 808-555-9876
Fred P. 808-555-6543
Fred P. 808-555-6543
^d
$ cat phone_list.txt
Laura O. 808-555-9876
Fred P. 808-555-6543
$
```

Appending Text to a File

The `-a` option on the `tee` command allows the file's current contents to be appended without overwriting the original contents.

For example:

```
$ tee -a phone_list.txt
Terry G. - 808-555-7941
Terry G. - 808-555-7941
^d
$ cat phone_list.txt
Laura O. - 808-555-9876
Fred P. - 808-555-6543
Terry G. - 808-555-7941
$
```

Capturing and Displaying Output

In the following example, the output from the `ls` command is captured in a file called `logfile` and is displayed on the screen one screenful at a time.

```
$ ls -lR | tee logfile | more
.:
total 4860
-rw-r--r--  1 user1  prigroup  54120 Apr 16 21:04 dante
-rw-r--r--  1 user1  prigroup   368 Mar 22 14:05 dante_1
drwxr-sr-x  5 user1  other    512 Mar 22 14:05 dir1
drwxr-xr-x  4 user1  other    512 Mar 22 14:05 dir2
drwxr-xr-x  3 user1  other    512 Mar 22 14:05 dir3
drwxr-xr-x  3 user1  other    512 Mar 22 14:05 dir4
-rw-r--r--  1 user1  staff  2368592 Apr 18 09:14 file.1
-rwxr----- 1 user1  staff      0 Apr 12 18:34 file.txt
-rw-r--r--  1 user1  prigroup  105 Mar 22 14:05 file2
-rw-r--r--  1 user1  prigroup  218 Mar 22 14:05 file3
-rw-r--r--  1 user1  prigroup  137 Mar 22 14:05 file4
-rw-r--r--  1 user1  prigroup   35 Mar 22 15:36 file5
-rw-r--r--  1 user1  prigroup   56 Mar 22 14:05 fruit
-rw-r--r--  1 user1  prigroup   57 Mar 22 14:05 fruit2
-rw-r--r--  1 user1  staff    2483 May  1 13:43 logfile
<output omitted>
```

Creating Directories

The `mkdir` command is used to create directories. You can create directories can be using absolute or relative pathnames.

This command enables you to specify more than one directory name on the same command line, creating several directories at the same time.

Command Format

```
mkdir directory_name(s)
mkdir -p directory_pathname(s)
```

Creating a New Directory

In this example, the `mkdir` command is used to create a new directory in user1's home directory.

```
$ cd
$ pwd
/export/home/user1
$ mkdir Reports
$ ls -dl Reports
drwxr-xr-x 2 user1 staff 512 Mar 1 16:24 Reports
$
$ mkdir Reports/Weekly
$ ls Reports
Weekly
$
$ cd Reports/Weekly
$ mkdir dir1 dir2 dir3
$ ls -F
dir1/ dir2/ dir3/
$
$ mkdir ~/games
$ cd
$ ls -F
dante      dir2/      file.1     file1      file4      games/     tutor.vi
dante_1    dir3/      file.2     file2      fruit      practice/
dir1/      dir4/      file.3     file3      fruit2     Reports/
$
```

Note – Remember the tilde (~) character is used only by the Korn and C shells to indicate your home directory. It is not used in the Bourne shell.

You must have the appropriate permissions to create a directory. (Permissions are discussed in detail later in the course.) If you do not have the correct permissions, an error message similar to the one shown below is displayed:

```
$ mkdir /export/home/Olympic
mkdir: Failed to make directory "/export/home/Olympic"; Permission denied
```

Creating Multiple Levels of Directories

To create multiple levels of directories at one time, use the `-p` option.

```
$ mkdir -p practice/dir1/admin
$
```

In the following example, `mkdir` creates a directory named `practice` as a subdirectory of the current working directory. At the same time, it creates a directory named `dir1` as a subdirectory of `practice`, and a directory named `admin` as a subdirectory of `dir1`.

Copying Files and Directories

The `cp` command copies files and directories.

Copying Files

The `cp` command can copy the contents of a file to another file, or it can copy multiple files while preventing overwrites to existing files.

Command Format

```
cp source_file destination_file
cp source_file(s) destination_directory
cp -i source_file destination_file
cp -i source_file(s) destination_directory
```

Copying a File to Another File within a Directory

The following example shows how to copy one file to a new file in the same directory.

```
$ cd
$ pwd
/export/home/user1
$ cp file3 feathers
$ ls
dante      dir2      dir5      file.2    file2     fruit     practice
dante_1    dir3      feathers  file.3    file3     fruit2    Reports
dir1       dir4      file.1    file1     file4     games     tutor.vi
$ cp feathers feathers_6
$
```

Copying Multiple Files

The following example demonstrates how to copy multiple files into a directory other than the current directory:

```
$ pwd
/export/home/user1
$ ls dir1
coffees  fruit      trees
$ cp feathers feathers_6 dir1
$ ls dir1
coffees  feathers  feathers_6  fruit      trees
$
```

Preventing Overwrites to an Existing File While Copying

Use the `cp -i` command as a precaution. The `-i` option prompts for confirmation before overwriting any existing file with the new file.

- Answering with a yes response means the copy should proceed.
- Answering with a no response prevents `cp` from overwriting the target.

For example:

```
$ cp -i feathers feathers_6
cp: overwrite feathers_6 (yes/no)? n
$
```

Copying Directories

Use the `cp -r` (recursive) command to copy a directory and its contents to another directory. If that directory does not exist, it is created by the `cp` command.

Command Format

```
cp -r source_directory(s) destination_directory
```

Without the `-r` option, the files and subdirectories contained within a directory are not copied.

When used with the `-i` option, `cp` prompts for verification before overwriting an existing directory or file.

Copying the Contents of a Directory to a New Directory

The following example demonstrates how to copy a directory and all of its contents to a new directory in the current directory. If you do not use the `-r` option, you will receive an error message.

```
$ cd
$ pwd
/export/home/user1
$ ls dir3
planets
$ cp dir3 ski.places
cp: dir3: is a directory
$ cp -r dir3 ski.places
$ ls ski.places
planets
$
```

The following example demonstrates how to copy a directory to another directory that is not in the current directory:

```
$ cd
$ pwd
/export/home/user1
$ cd dir3
$ cp -r planets ../dir1/constellation
$ cd
$ cp -ri dir1 ski.places /tmp
```

NEW OUTPUT HERE

```
$ ls -F /tmp
dir1/ ski.places/
$
```


Moving and Renaming Files and Directories

To move or rename a file or directory, use the `mv` (move) command. This command does not affect the contents of the file or directory, it simply changes its location or name from the old name to the new name. The old name equates to the source, and the new name equates to the target. If the target directory does not exist, it is created.

Command Format

```
mv [-i] source target_file
```

```
mv [-i] source target_directory
```

The `mv -i` option prompts for confirmation whenever the move would overwrite an existing target.

- Answering with a yes response means the move should proceed.
- Answering with a no response prevents `mv` from overwriting the target.

Renaming Files in the Current Directory

The following example demonstrates how to rename a file in the current directory:

```
$ cd ~/dir1/coffees
$ ls
beans      nuts
$ mv nuts brands
$ ls
beans      brands
$
```

Moving Files to Another Directory

The following example demonstrates how to move a file into another directory:

```
$ pwd
/export/home/user1/dir1/coffees
$ ls
beans    brands
$ mv brands ~
$ ls
beans
$ ls ~/b*
/export/home/user1/brands
$
```

Note – If you are moving a directory and the target directory does not exist, the target directory is created.

Renaming Directories

The following example demonstrates how to use the `mv` command to rename directories within the current directory:

```
$ cd
$ mkdir maildir
$ ls
brands    dir3      file.1    file3     maildir
dante     dir4      file.2    file4     practice
dante_1   dir5      file.3    fruit     Reports
dir1      feathers  file1     fruit2    ski.places
dir2      feathers_6 file2     games     tutor.vi
$
$ mv maildir monthly
$ ls
brands    dir3      file.1    file3     monthly
dante     dir4      file.2    file4     practice
dante_1   dir5      file.3    fruit     Reports
dir1      feathers  file1     fruit2    ski.places
dir2      feathers_6 file2     games     tutor.vi
$
```

Moving a Directory and its Contents

The following example demonstrates how to use the `mv` command to move a directory and its contents into another directory:

```
$ pwd
/export/home/user1
$ ls practice
dir1      mailbox      project      projection    research
$ mv practice letters
$ ls letters
dir1      mailbox      project      projection    research
$
```

Renaming Files in Another Directory

The following example demonstrates how to use the `mv` command to rename a file in a directory other than the current directory:

```
$ pwd
/export/home/user1
$ mv letters/project letters/project2
$ ls letters
dir1      mailbox      project2      projection      research
$
```

Removing Files and Directories

Once a file is no longer needed, you can permanently remove it by using the `rm` command.

Removing Files

You can use the `rm` command to remove a single file or several files at once.

Command Format

```
rm -i filename(s)
```

Removing Several Files

you can remove several files at the same time. For example:

```
$ cd ~/letters
$ pwd
/export/home/user1/letters
$ ls
dir1    mailbox    project2  projection  research
$ rm research project2
$ ls
dir1    mailbox    projection
$
```

Use the `rm -i` command as a precaution. The `-i` option prompts for confirmation before removing any file.

- Answering with a yes response means the deletion should proceed.
- Answering with a no response prevents `rm` from overwriting the target.

For example:

```
$ rm -i projection
rm: remove projection: (yes/no) ? y
$ ls
dir1      mailbox
$
```

Removing Directories

You can remove unwanted directories by using the `rmdir` and `rm` commands.

- The `rmdir` command deletes only empty directories.
- The `rm` command removes a directory that contains files.

Command Format

```
rmdir directory_name(s)
```

```
rm -ir directory_name(s)
```

Removing an Empty Directory

Use the `rmdir` command to remove an empty directory. For example:

```
$ cd
$ pwd
/export/home/user1
$ mkdir -p newdir/empty
$ cd newdir
$ ls -F
empty/
$ rmdir empty
$ ls
$
```

Removing a Directory with Contents

Use `rm -r` to remove a directory that is not empty. For example:

```
$ cd
$ pwd
/export/home/user1
$ ls letters
dir1      mailbox
$ rm -r letters
$ ls letters
letters: No such file or directory
$
```

Interactively Removing Directories

Use `rm -ir` to interactively remove directories. For example:

```
$ mkdir -p ~/practice/dir1
$ ls ~/practice
dir1
$ rm -ir practice
rm: examine files in directory practice (yes/no)? y
rm: examine files in directory practice/dir1 (yes/no)? y
rm: remove practice/dir1: (yes/no)? y
rm: remove practice: (yes/no)? y
$
```

Note – You cannot remove a directory while you are in it. You must move to the parent directory to remove a subdirectory.

Command-line Printing

The `lp` command queues text files for printing.

From the command line, you can print American Standard Code for Information Interchange (ASCII) text or PostScript™ files. Do not use this method to print data files, such as binary files or files created in applications, such as FrameMaker.

Command Format

```
lp -options filename(s)
```

Options

Use the following options with the `lp` command:

<code>-d</code>	Specifies the desired printer. If printing to the default printer, the <code>-d</code> option is not necessary.
<code>-o nobanner</code>	Specifies that the banner page is not to be printed.
<code>-n</code>	Prints a specified number of file copies.
<code>-m</code>	Sends a mail message to you after a job is complete.

Printing a File

Sending Files to a Printer

To print the file `feathers` located in your `home` directory on the default printer, execute the following command:

```
$ lp ~/feathers
request id is printerA (1 file(s))
$
```

To specify another printer, for example one named `printerB`, use the `-d` option to specify another printer. For example:

```
$ lp -d printerB ~/feathers
request id is printerB (1 file(s))
$
```


Displaying Printer Status and Queues

The `lpstat` command displays the status of the printer queue.

Command Format

```
lpstat -options
```

Options

Use the following options with the `lpstat` command:

-p	Displays the status of all printers
-o	Displays the status of all output requests
-d	Displays which printer is the system default
-t	Displays complete status information for all printers
-s	Displays a status summary for all printers
-a	Displays which printers are accepting requests

Note – When a print request has been sent to the printer, the output of `lpstat` can show the print request as *filtered*. Filtering indicates that a print request is in the process of printing.

Displaying the Status of All Print Requests

To display the status of all print requests, execute the following:

```
$ lpstat -o
printerA-7   user1   391   Dec 10 16:30   on printerA
printerB-1   user2   551   Dec 10 16:45   filtered
$
```

Displaying Requests on a Specific Printer's Queue

To display requests on a specific printer's queue, execute the following:

```
$ lpstat printerA
printerA      user2      551      Dec 10 16:45
printerB      user3      632      Dec 10 16:47
$
```

Determining the Status of All Configured Printers

To determine the status of all configured printers, execute the following:

```
$ lpstat -t
scheduler is running
system default destination: printerA
system for printerA: host1
system for printerB: host2
printerA accepting requests since Wed May 6 08:20:00 EST
1998
printerB accepting requests since Mon May 25 09:43 EST
1998
$
```

Determining Which Printers Are Configured on the System

To determine which printers are configured on the system, execute the following:

```
$ lpstat -s
scheduler is running
system default destination: printerA
system for printerA: host1
system for printerB: host2
$
```

Displaying Which Printers Are Accepting Requests

To display which printers are accepting requests, execute the following:

```
$ lpstat -a  
printerA accepting requests since Wed May 6 08:20:00 EST 1998  
printerB accepting requests since Mon May 25 09:43:19 PST 1998  
$
```

Removing a Print Request

The `cancel` command enables you to cancel print requests previously sent with the `lp` command. To do this, you must first use the `lpstat` command to identify the *request-ID*.

Command Format

```
cancel request-ID
```

Canceling a Print Request

For example, to cancel a print request, execute the following:

```
$ lpstat printerB
printerB-5 user2 551      Dec 10 16:45
printerB-6 user3 632      Dec 10 16:47
$ cancel printerB-6
request "printerB-6" cancelled
$
```

Use the `cancel -u username` (login ID) to remove all requests owned by a specific user. For example:

```
$ cancel -u user2
request "printerB-5" cancelled
$
```

Note – You can remove only your own print jobs. When using the CDE Printer Manager, it appears that a user can cancel another person's print job, but the job is immediately reinstated.

Format and Print a File

Use the `pr` command to format and print the contents of a file according to different format options. This command automatically prints the file contents to the terminal screen for viewing.

By default, the `pr` command prints a header, up to 66 lines of text, and a trailer that contains five blank lines, for each page.

The header of each page includes the name of the file, the date and time last modified, and a page number.

Command Format

```
pr -options filename(s)
```

Options

You can use the following options when executing the `pr` command:

<code>+page</code>	Begins printing on page number specified
<code>-column</code>	Prints file in multi-column format (default is one column)
<code>-d</code>	Prints file in double-spaced format
<code>-h header</code>	Replaces the file name contained in the page header with the pattern <i>header</i>
<code>-l lines</code>	Resets page length to <i>lines</i> (default is 66 lines)
<code>-m</code>	Merges files side-by-side into text columns
<code>-t</code>	Prints file(s) without the header
<code>-n</code>	Numbers each line

Format and Print Files to the Screen

The following examples print the file contents to the terminal screen, using the various format options.

- To begin printing a file on page 4:

```
$ pr +4 dante | more
```
- To format a file into three text columns:

```
$ pr -3 dante | more
```
- To print a file double-spaced with only 20 lines per page:

```
$ pr -d -l 20 file1 | more
```
- To merge three files side-by-side in text columns:

```
$ pr -m file1 file2 file3 | more
```
- To send a file to the default printer:

```
$ pr file3 | lp
```
- To print two columns, double-sided, with the header "TITLE" for file1 and file2 on the default printer:

```
$ pr -2dh "TITLE" file1 file2 | lp
```

Exercise: Using Directory and File Commands



Exercise objective – In this exercise, you will use the commands described in this module to display the contents of files, compare files; determine the file type, and create, move, and remove files and directories.

Tasks

Complete or answer the following:

1. Determine the file type of `/etc/passwd` and display the contents of the file.

2. Display the contents of the file `/usr/dict/words` one screen at a time using the `more` command. Exit after displaying two screens.

3. Display the contents of the file `/usr/dict/words` on the screen using the `pg` command.
 - a. Display half a page more. _____
 - b. Display the next line. _____
 - c. Move to the last page. _____
 - d. Quit and return to the shell prompt. _____
4. Display only the first five lines of the file `/usr/dict/words` on the screen.

5. Display only the last eight lines of the file `/usr/dict/words` on the screen.

6. Print the file `/usr/dict/words` to the screen in an eight-column format.

7. Determine the total number of lines contained in the file `/usr/dict/words`.

8. What command would you most likely use to read the contents of the binary file `/usr/bin/cp`?

9. Return to your home directory (if you need to) and list the contents.

10. Copy the `dir1/coffees/beans` file into the `dir4` directory and call it `roses`.

11. Create a directory called `vegetables` in `dir3`.

12. Move the `dir1/coffees/beans` file into the `dir2/recipes` directory.

13. From your home directory, make a directory called `practice`.

14. Copy `dir3/planets/mars` to the `practice` directory and name the file `addresses`.

15. Create a directory called `play` in your `practice` directory and move the `practice/addresses` file to that `play` directory.

-
16. Copy the `play` directory in the `practice` directory to a new directory in the `practice` directory called `appointments`.
-
17. Recursively list the contents of the `practice` directory.
-
18. In your `home` directory and with one command, create a directory called `house` with a subdirectory of `furniture`.
-
19. Create an empty file called `chairs` in the new directory `furniture`.
-
20. Using one command, create three directories called `letters`, `memos`, and `misc` in your `home` directory.
-
21. Using one command, delete the directories called `memos` and `misc` in your `home` directory.
-
22. Try to delete the directory called `house/furniture` with the `rm` (no options) command. What happens?
-
23. Identify the command to delete a directory that is not empty. Delete the directory `house/furniture`. List the contents of `house` to verify that `furniture` has been deleted.
-
24. From your `home` directory, redirect the output of the `ls` command to a file called `file.list`.
-
25. Display the contents of `file.list` using the `cat` command.
-

26. Append the output of the `date` command to `file.list`.

27. Display a calendar and using the `tee` command, append the output to `file.list`.

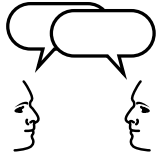
28. Use the `tee` command to create a file with the names of five of the students in the class in it. Examine the contents of your class list with the `cat` command.

29. Send the files `file.list` and `classlist` to the printer.

30. Check the print queue to display the status of all print requests.

31. Cancel a print request, removing it from the print queue.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

1. Determine the file type of `/etc/passwd` and display the contents of the file.

```
$ file /etc/passwd
$ cat /etc/passwd
```

2. Display the contents of the file `/usr/dict/words` one screen at a time, using the `more` command. Exit after displaying two screens.

```
$ more /usr/dict/words
```

To display two screens, press the space bar twice.
To exit, type “q” or Control-c

3. Display the contents of the file `/usr/dict/words` on the screen using the `pg` command. Display half a page more. Display the next line. Move to the last page. Quit and return to the shell prompt.

```
$ pg /usr/dict/words
```

To display half a page more, type a “d” at the : prompt.
To display the next line, type an “l” at the : prompt.
To move to the last page, type a “\$” at the : prompt.
To quit and return to the shell prompt, type “q”.

4. Displays the first five lines of the file `/usr/dict/words` on the screen.

```
$ head -5 /usr/dict/words
```

5. Display the last eight lines of the file `/usr/dict/words` on the screen.

```
$ tail -8 /usr/dict/words
```

6. Print the file `/usr/dict/words` to the screen in an eight-column format.

```
$ pr -8 /usr/dict/words
```

7. Determine the total number of lines contained in the file `/usr/dict/words`.

```
$ wc -l /usr/dict/words
```

8. What command would you most likely use to read the contents of `/usr/bin/cp`?

```
$ strings /usr/bin/cp
```

9. Return to your home directory (if you need to) and list the contents.

```
$ cd; ls
```

10. Copy the `dir1/coffees/beans` file into the `dir4` directory and call it `roses`.

```
$ cp dir1/coffees/beans dir4/roses
```

11. Create a directory called `vegetables` in `dir3`.

```
$ mkdir dir3/vegetables
```

12. Move the `dir1/coffees/beans` file into the `dir2/recipes` directory.

```
$ mv dir1/coffees/beans dir2/recipes
```

13. From your home directory, make a directory called `practice`.

```
$ mkdir practice
```

14. Copy `dir3/planets/mars` to the `practice` directory and name the file `addresses`.

```
$ cp dir3/planets/mars practice/addresses
```

15. Create a directory called `play` in your `practice` directory and move the `practice/addresses` file to that `play` directory.

```
$ mkdir practice/play  
$ mv practice/addresses practice/play
```

16. Copy the `play` directory in the `practice` directory to a new directory in the `practice` directory called `appointments`.

```
$ cp -r practice/play practice/appointments
```

17. Recursively list the contents of the `practice` directory.

```
$ ls -R practice
```

18. In your home directory and with one command, create a directory called `house` with a subdirectory of `furniture`.

```
$ cd; mkdir -p house/furniture
```

19. Create an empty file called `chairs` in the new directory `furniture`.

```
$ touch house/furniture/chairs
```

20. Using one command, create three directories called `letters`, `memos`, and `misc` in your home directory.

```
$ mkdir letters memos misc
```

21. Using one command, delete the directories called `memos` and `misc` in your home directory.

```
$ rmdir memos misc
```

22. Try to delete the directory called `house/furniture` with the `rm` (no options) command. What happens?

```
$ rm house/furniture  
rm: house/furniture is a directory
```

23. Identify the command to delete a directory that is not empty. Delete the directory `house/furniture`. List the contents of `house` to verify that `furniture` has been deleted.

```
$ rm -r house/furniture; ls house
```

24. From your home directory, redirect the output of the `ls` command to a file called `file.list`.

```
$ ls > file.list
```

25. Display the contents of `file.list` using the `cat` command.

```
$ cat file.list
```

26. Append the output of the `date` command to `file.list`.

```
$ date >> file.list
```

27. Display a calendar and using the `tee` command, append the output to `file.list`.

```
$ cal | tee -a file.list
```

28. Use the `tee` command to create a file with the names of five of the students in the class in it. Examine the contents of your class list with the `cat` command.

```
$ tee classlist  
  name1  
  name1  
  name2  
  name2  
  name3  
  name3  
  name4  
  name4  
  name5  
  name5  
  ctrl-d  
$ cat classlist
```

29. Send the files `file.list` and `classlist` to the printer.

```
$ lp file.list classlist
```

30. Check the print queue to display the status of all print requests.

```
$ lpstat -o
```

31. Cancel a print request, removing it from the print queue.

```
$ cancel request-id
```

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Determine file types with the `file` command
- ☐ Display the contents of text files using the `cat`, `more`, `pg`, `head`, and `tail` commands
- ☐ Determine character, word, and line counts using the `wc` command
- ☐ Create empty files or update modification time on existing files using the `touch` command
- ☐ Use the `tee` command to create text within a file
- ☐ Create and remove directories using the `mkdir` and `rmdir` commands
- ☐ Manage files and directories using the `mv`, `cp`, and `rm` commands
- ☐ Use commands to print a file, check print queue status, and cancel a print request
- ☐ Format and print the contents of files using the `pr` command

Objectives

Upon completion of this module, you should be able to:

- Use the `find` command to locate files in the Solaris Operating Environment directory tree using specific search criteria
- Use the `cmp` and `diff` commands to compare the contents of text files for differences
- Sort the content of text files in alphabetical and numerical order using the `sort` command
- Search for regular expressions in the contents of one or more files using the commands `grep`, `egrep`, and `fgrep`

Additional Resources



Additional resources – The following references can provide additional details on the topics discussed in this module:

- *Solaris Advanced User's Guide*, Part Number 801-6615-10
- *Solaris System Administrator AnswerBook*

Locating Files Using the `find` Command

To locate a file in the directory tree, use the `find` command. This command gives you the ability to locate files based on specific criteria, such as file name, size, owner, modification time, or type.

The `find` command recursively descends the directory tree in the pathname list, seeking those files that match the criteria.

As the files that match those values are located, the path to each file is displayed on the screen.

Command Format

```
find pathname(s) expression(s) action(s)
```

The first argument on the command line is the pathname where the search starts; this can be an absolute or a relative pathname.

The remainder of the arguments specify the criteria by which to find the file(s), and what action to take once located.

Table 5-1 through Table 5-3 show the arguments, expressions, and actions that can be used with the `find` command.

Table 5-1 Arguments Used with the `find` Command

Argument	Definition
pathname	The directory path where the search originates.
expression	The search criteria specified by one or more options. Specifying multiple options causes <code>find</code> to treat the statement as an “and” request so all listed expressions must be verified as true.

Table 5-2 Expressions Used with the `find` Command

Expression	Definition
<code>-name <i>filename</i></code>	Finds files matching the specified <i>filename</i> . Metacharacters are acceptable if placed inside quotes.
<code>-size [+/-]<i>n</i></code>	Finds files that are larger than <i>+n</i> , or smaller than <i>-n</i> . The <i>n</i> represents 512-byte blocks.
<code>-atime [+/-]<i>n</i></code>	Finds files that have been accessed more than <i>+n</i> or less than <i>-n</i> days.
<code>-mtime [+/-]<i>n</i></code>	Finds files that have been modified more than <i>+n</i> or <i>-n</i> days.
<code>-user <i>loginID</i></code>	Finds all files that are owned by the <i>login ID</i> name.
<code>-type</code>	Finds a file type, for example: <code>f</code> (file) <code>d</code> (directory).
<code>-perm</code>	Finds files that have certain access permission bits.
<code>-print</code>	Instructs <code>find</code> to print the current pathname to the terminal screen. This is the default.

Table 5-3 Actions Used with the `find` Command

Action	Definition
<code>-exec <i>command</i> {} \;</code>	Executes the specified <i>command</i> on each file located, automatically. You must include the string <code>{ } \;</code> to instruct <code>find</code> to execute the <i>command</i> on every file identified by the search criteria. There must be a space before the backslash (<code>\</code>).
<code>-ok <i>command</i> {} \;</code>	Interactive form of <code>-exec</code> . Requires input before <code>find</code> applies the <i>command</i> on the file(s).

The following examples illustrate the power of the `find` command.

- To search for `core` files starting at the root (`/`) directory:

```
$ find / -name core
```

- To search for `core` files starting at your home directory and delete each one as it is found:

```
$ find ~ -name core -exec rm {} \;
```

- To look for all files, starting at the current directory that have not been modified in the last 90 days:

```
$ find . -mtime +90
```

- To find files larger than 57 blocks (512-byte blocks) starting at the home directory:

```
$ find ~ -size +57
```

- To search for files that end with the characters “tif,” starting at the `/usr` directory:

```
$ find /usr -name *tif
```

Finding Differences Between Files

*Locating Text Differences Using the **cmp** Command*

Comparing files to determine text differences can be easily accomplished using the **cmp** command. This command prints results only if there are differences between the files. If no results are shown, the files are exactly the same.

The **cmp** command performs a byte-by-byte comparison of each file. If the text within the files differ, then **cmp** prints the byte count and the line number of the first difference and then stops.

This command works with both binary and ASCII files.

Command Format

```
cmp filename filename
```

*Using the **cmp** Command to Compare Files that Appear the Same*

```
$ cmp fruit fruit2  
fruit fruit2 differ: char 27, line 5
```

This output identifies the first occurrence of a difference between the two files. The difference occurred at the 27th character position and was found on line 5.

*Locating Text Differences Using the **diff** Command*

The **diff** command is another command used for finding differences between files.

The results of this command display line-by-line differences between two text files, providing you with instructions on how to edit one file to make it the same as the other.

Command Format

`diff -option filename filename`

Table 5-4 shows the options that can be used when using the `diff` command.

Table 5-4 Options for the `diff` Command

Options	Definition
-i	Ignores the case of letters; for example, A is equal to a.
-c	Produces a listing of differences. The results begin with the name of the files and creation dates, followed by a dozen asterisk's (*). The line numbers of <i>file1</i> are displayed, followed by the actual lines from the file. Lines that differ from <i>file2</i> are tagged by a dash (-). The line numbers of <i>file2</i> are then displayed, followed by the actual lines from the file. Lines that differ from <i>file1</i> are tagged by a plus (+) symbol.

Using `diff` with the `-c` option

When using `diff` with the `-c` option to compare files, the results are displayed in three sections.

The first section shows the names of the files being compared and their creation dates, followed by a string of asterisks (*).

The second section shows, for *file1*, a line count of those lines that differ from *file2*, followed by a comma and a total line count for all lines contained in *file1*. Actual text lines from *file1* are printed with each line that differs from *file2* being tagged with a dash (-) character.

The third section shows, for *file2* a line count of those lines that differ from *file1*, followed by a comma and a total line count for all lines contained in *file2*. Actual text lines from *file2* are printed with each line that differs from *file1* being tagged with a plus (+) character.

Note – In the `diff -c` example that follows, the first line of text in both files (the word `lemon`) is replaced in the `diff` output by `diff` results.

For example, execute the following to use `diff` to compare files:

```
$ cat fruit
lemon
orange
apple
banana
pear
mango
tomato
pomegranate
$ cat fruit2
lemon
orange
apple
banana
tomato
guava
mango
pomegranate
$ diff -c fruit fruit2
*** fruit      Wed May 31 16:45:05 2000
--- fruit2     Wed May 31 16:45:05 2000
*****
*** 2,8 ****
    orange
    apple
    banana
-   pear
-   mango
    tomato
    pomegranate
--- 2,8 ----
    orange
    apple
    banana
    tomato
+   guava
+   mango
    pomegranate
$
```

Sorting Data

The `sort` command sorts text lines in one or more files and prints the results to the screen.

The `sort` command provides a quick and easy method to organize data in numeric or alphabetic order.

By default, `sort` relies on white space to delimit the various fields within the data of a file.

Command Format

```
sort -options filename(s)
```

Options

The options available with the `sort` command enable you to define the type of sort to perform and identify which field to begin the sorting.

Table 5-5 Options for Using the `sort` Command

Option	Definition
-n	Performs a numeric sort.
(+ -)n	Begins (+n) or end (-n) the sort with the field following the <i>n</i> separator.
-r	Reverses the order of the sort.
-f	Folds uppercase and lowercase characters together; ignore case in sort order.
+nM	Sorts the first three characters of the field as an abbreviated month name.
-d	Uses dictionary order. Only letters, digits, and spaces are compared; all other symbols are ignored.
-o filename	Prints results into the file <i>filename</i> .

Table 5-5 Options for Using the `sort` Command (Continued)

Option	Definition
<code>-b</code>	Ignores leading blank characters when determining the starting and ending positions of a restricted sort key.
<code>-t char</code>	Uses <code>char</code> as the field separator character. If <code>-t</code> is not specified, blank characters are used as default field separators.

Using `sort` with Different Options

The following examples show various ways to use the `sort` command with different options.

```
$ cat fileA
Annette      48486
Jamie        48481
Fred         48487
Sondra       48483
Janet        48482
$
$ sort fileA
Annette      48486
Fred         48487
Jamie        48481
Janet        48482
Sondra       48483
$ sort +1n fileA
Jamie        48481
Janet        48482
Sondra       48483
Annette      48486
Fred         48487
```

The first example shows the contents of `fileA` using the `cat` command.

The first `sort` command produces an alphabetic sort, beginning with the first character of each line.

The second sort produces a numeric sort on the second field (sort skips one separator with the +1 syntax).

Using sort on Different Fields Within a File

The following examples show how to use the sort command on different fields within a file.

```
$ ls -l f* > list
$ cat list
-rw-r--r-- 1 user1 staff 218 Mar 1 11:36 feathers
-rw-r--r-- 1 user1 staff 218 Mar 1 11:36 feathers_6
-rw-r--r-- 1 user1 staff 0 Feb 25 12:54 file.1
-rw-r--r-- 1 user1 staff 0 Feb 25 12:54 file.2
-rw-r--r-- 1 user1 staff 0 Feb 25 12:54 file.3
-rw-r--r-- 1 user1 staff 1696 Feb 22 14:51 file1
-rw-r--r-- 1 user1 staff 156 Mar 1 14:48 file2
-rw-r--r-- 1 user1 staff 218 Feb 22 14:51 file3
-rw-r--r-- 1 user1 staff 137 Feb 22 14:51 file4
-rw-r--r-- 1 user1 staff 56 Feb 22 14:51 fruit
-rw-r--r-- 1 user1 staff 57 Feb 22 14:51 fruit2
$ sort -rn +4 list -o num.list
$ cat num.list
-rw-r--r-- 1 user1 staff 1696 Feb 22 14:51 file1
-rw-r--r-- 1 user1 staff 218 Mar 1 11:36 feathers_6
-rw-r--r-- 1 user1 staff 218 Mar 1 11:36 feathers
-rw-r--r-- 1 user1 staff 218 Feb 22 14:51 file3
-rw-r--r-- 1 user1 staff 156 Mar 1 14:48 file2
-rw-r--r-- 1 user1 staff 137 Feb 22 14:51 file4
-rw-r--r-- 1 user1 staff 57 Feb 22 14:51 fruit2
-rw-r--r-- 1 user1 staff 56 Feb 22 14:51 fruit
-rw-r--r-- 1 user1 staff 0 Feb 25 12:54 file.3
-rw-r--r-- 1 user1 staff 0 Feb 25 12:54 file.2
-rw-r--r-- 1 user1 staff 0 Feb 25 12:54 file.1
$ sort +5M +6n list -o update.list
$ cat update.list
-rw-r--r-- 1 user1 staff 56 Feb 22 14:51 fruit
-rw-r--r-- 1 user1 staff 57 Feb 22 14:51 fruit2
-rw-r--r-- 1 user1 staff 137 Feb 22 14:51 file4
-rw-r--r-- 1 user1 staff 218 Feb 22 14:51 file3
-rw-r--r-- 1 user1 staff 1696 Feb 22 14:51 file1
-rw-r--r-- 1 user1 staff 0 Feb 25 12:54 file.1
-rw-r--r-- 1 user1 staff 0 Feb 25 12:54 file.2
-rw-r--r-- 1 user1 staff 0 Feb 25 12:54 file.3
-rw-r--r-- 1 user1 staff 156 Mar 1 14:48 file2
```

```
-rw-r--r--  1 user1    staff      218 Mar  1 11:36 feathers
-rw-r--r--  1 user1    staff      218 Mar  1 11:36 feathers_6
$
```

The first previous example takes the output of the `ls` command and places it in the file named `list`. The contents of this file are then viewed with the `cat` command.

The first `sort` command produces a reverse, numeric order sort on the fifth field and places the results into a file called `num.list`.

The second `sort` example represents a multilevel sort on fields 6 and 7 in the file named `list`. The option `+5M` performs an alphabetic sort on the month in the sixth field. The `+6n` option performs a second-level numeric sort on the day in the seventh field, and places all results in the `update.list` file.

Searching for Text in Files

The Solaris Operating Environment provides a family of commands used to search the contents of one or more files for a specific character pattern. A *pattern* can be a single character, a string of characters, a word, or a sentence.

By definition, the pattern of characters used to match the same characters in a search is called a regular expression (RE). The commands used to search for regular expressions in files include `grep`, `egrep`, and `fgrep`.

- The `grep` command globally searches for regular expressions in files and prints all lines containing the regular expression to standard output.

Note – The `grep` command is derived from the phrase “globally search for a regular expression and print if found.” Originally the command was `g/re/p`.

- The `egrep` and `fgrep` command are variants of `grep`; they are known as *extended grep* and *fast grep*.

Using the `grep` Command

The `grep` command searches the contents of one or more files for a regular expression or character pattern.

If found, `grep` prints every line containing the pattern to the screen and does not change the file content in any way.

Command Format

`grep option(s) pattern filename(s)`

Options

The **grep** command provides a number of options to modify the way it does its search or displays lines. Some useful options are described in Table 5-6.

Table 5-6 Options for **grep**

Option	Definition
-i	Ignores case. Uppercase and lowercase are considered identical.
-l	Lists only the names of files with matching lines.
-n	Precedes each line with its relative line number in the file.
-v	Inverts the search to display only lines that do not match the pattern.
-c	Prints only a count of the lines that contain <i>pattern</i> .

Examples of Searching for Regular Expressions with **grep**

The following are examples of using **grep** to search for regular expression:

- To search for all lines containing the pattern “root” in the `/etc/group` file, execute the following:

```
$ grep -n root /etc/group
1:root::0:root
3:bin::2:root,bin,daemon
4:sys::3:root,bin,sys,adm
5:adm::4:root,adm,daemon
6:uucp::5:root,uucp
7:mail::6:root
8:tty::7:root,tty,adm
9:lp::8:root,lp,adm
10:nuucp::9:root,nuucp
12:daemon::12:root,daemon
$
```

- To search for all lines that do not contain the pattern “root” in the /etc/group file, execute the following:

```
$ grep -v root /etc/group
```

```
other::1:
staff::10:
nobody::60001:
noaccess::60002:
sun4prog::100:
funda::105:
cprg::170:
csh::180:
sysadmin::14:
fire::20:
sl300::30:
java::190:
cplus::700:
sysint::1000:
internals::2000:
sunnetmgr::3000:
OpenWindows::5000:
SAE::5500:
guest::20000:guest,kbrown
$
```

- To search for just the names of the files that contain the pattern “root,” execute the following:

```
$ cd /etc
```

```
$ grep -l root group passwd hosts
```

```
group
passwd
$
```

Note – For multiple file searches, the results are listed with the file name in which the pattern was found. For single file searches, only the results are displayed.

- To search for the pattern “the” in all files in the /etc directory, listing only the names of files with lines that match the pattern “the” or “The,” execute the following:

```
$ cd /etc
$ grep -li the *
$ cd /etc
$ grep -li the *
aliases
asppp.cf
dacf.conf
device.tab
devlink.tab
dgroup.tab
fmthard
format
```

<output omitted>

```
syslog.conf
system
termcap
TIMEZONE
ttsrchr
umountall
grep: can't open utmpipe
$
```

- To search for the pattern “root” in the /etc/group file, printing only a count of the lines that contain the pattern, execute the following:

```
$ grep -c root group
10
$
```

- To search for the pattern “mar 1” in the output from the `ls -la` command, execute the following:

```
$ ls -la | grep -i 'mar 1'
$
$ ls -la | grep -i 'mar 1'
prw----- 1 root    root      0 Mar  1 11:05 initpipe
-r--r--r-- 1 root    root     806 Mar  1 13:39 mnttab
prw----- 1 root    root      0 Mar  1 11:06 utmpipe
$
```

If the date is a single numeral, the `grep` pattern must have two spaces placed between the month and the day. For example: Jan 3.

Regular Expression Metacharacters

The `grep` command supports several regular expression metacharacters to further define a search pattern. Table 5-7 describes some basic, yet useful metacharacters.

Table 5-7 Regular Expression Metacharacters

Metacharacter	Purpose	Sample	Result
<code>^</code>	Beginning of line marker	<code>'^pattern'</code>	Matches all lines beginning with "pattern."
<code>\$</code>	End of line anchor	<code>'pattern\$'</code>	Matches all lines ending with "pattern."
<code>.</code>	Matches one character	<code>'p.....n'</code>	Matches lines containing a "p", followed by five characters, followed by an "n."
<code>*</code>	Matches proceeding item one or more times	<code>[a-z]*</code>	Matches lowercase alphanumeric characters.
<code>[]</code>	Matches one character in the pattern	<code>'[Pp]attern'</code>	Matches lines containing "Pattern" or "pattern."
<code>[^]</code>	Matches one character not in the pattern	<code>'[^a-m]attern'</code>	Matches lines not containing "a" through "m" followed by "attern."

Examples

The following are examples of using regular expression metacharacters:

- To print all lines that begin with the word “root” in the /etc/passwd file:

```
$ grep '^root' /etc/passwd
```

- To print all lines containing an “A,” followed by three characters, followed by an “n” in the /etc/passwd file:

```
$ grep 'A...n' /etc/passwd
```

- To print all lines that end with the word “adm” in the /etc/group file:

```
$ grep 'adm$' /etc/group
```

Using the egrep Command

The egrep command searches the contents of one or more files for a regular expression, using extended regular expression metacharacters.

The egrep command uses all the same options and metacharacters as grep.

Command Format

```
egrep option(s) pattern filename(s)
```

Extended Regular Expression Metacharacters

Table 5-8 show the regular expression metacharacters that are new with `egrep`.

Table 5-8 Extended Regular Expression Metacharacters

Metacharacter	Purpose	Sample	Result
<code>+</code>	Matches one or more of the preceding characters	<code>'[a-z]+ark'</code>	Matches one or more lowercase letters, followed by "ark". For example: bark, dark, lark, mark, park.
<code>x y</code>	Matches either x or y	<code>'apple orange'</code>	Matches for either expression.
<code>()</code>	Groups characters	<code>'search(es ing)+'</code> <code>'(12)+'</code>	Matches for one or more occurrences. For example: searches or searching; 1 or 2.

Examples Using `egrep`

The following example show various ways to use the `egrep` command:

- To search for lines containing one or more consecutive occurrences of the pattern "in," execute the following:

```
$ egrep '(in)+' /etc/group
```

- To search for all lines containing the pattern "N," followed by either an "e" or "o" one or more times, execute the following:

```
$ egrep 'N(e|o)+' /etc/passwd
```

- To search for lines containing the pattern "Network Admin" or "uucp Admin," execute the following:

```
$ egrep '(Network|uucp) Admin' /etc/passwd
```

Using the fgrep Command

The `fgrep` command searches a file for a pattern. It differs from `grep` and `egrep` in that it does not recognize any RE metacharacters specified on the command line. It recognizes only the literal meaning of these characters; treating a (?) as a question mark, and a (\$) as a dollar sign.

Use `fgrep` to search for a specific pattern in a file that includes metacharacter symbols.

Command Format

```
fgrep option(s) pattern filename(s)
```

Example Using fgrep

The following example shows how to use the `fgrep` command:

- To search for all lines in the file containing the string of text and symbols, execute the following:

```
$ fgrep '* /etc/system'
```

Exercise: Locating Files and Text



Exercise objective – The purpose of this exercise is to give you practice searching for files and directories using the `find` command, and displaying and manipulating text in files.

Tasks

Complete the following steps. In the space provided, write the commands you would use to perform each task.

1. Starting in your `home` directory, find all files ending with a “2”.

2. Use the `find` command to search the `/usr` directory and display the file names of any length that end with “ln”.

3. Starting in your `home` directory, find all files of type `file`, printing the full pathname of each file located.

4. In your `home` directory, find all files of type `directory`.

5. From your `home` directory, find all files in the `/etc` directory that have access permissions `644` (read and write for the owner, read for the group and others).

6. From your `home` directory, find files of size 0 (zero) in the `/tmp` directory and ask if it is permissible to remove them.

-
7. Display all the files created in your home directory using `ls`, `grep`, and today's date.

 8. Search for the text string "other" in the `/etc/group` file, and have it display to the screen.

 9. Using the `grep` command, look for all lines in the `file4` file located in your home directory that do not contain the letter "M".

 10. Display all lines in the files `dante`, `file1`, and `dante_1` that contain the pattern "he".
 11. Display all the lines in the file `file4` that contain the pattern either "Sales" or "Finance".

 12. Sort the lines in the file `fruit` alphabetically.

 13. Sort the file `fruit` in reverse order.

Note – To create a file for use in the next step, execute:
\$ `ls -la > ls.output`

14. Sort the `ls.output` file. Produce a numerical listing by size of the files, in reverse order. What command did you use?

15. Perform a multilevel sort of the `ls.output` file that places the data in chronological order, then alphabetically by name. What command did you use?

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

Complete the following steps. In the space provided, write the commands you would use to perform each task.

1. In your home directory, find all files ending with a “2”.

```
$ find . -name '*2'
```

2. Use the `find` command to search the `/usr` directory and display the file names of any length that end with “ln”.

```
$ find /usr -name '*ln'
```

3. In your home directory, find all files of type `file`, printing the full pathname of each ordinary file.

```
$ find . -type f
```

4. In your home directory, find all files of type `directory`.

```
$ find . -type d
```

5. From your home directory, find all files in the `/etc` directory that have access permissions 644 (read and write for the owner, read for the group and others).

```
$ find /etc -perm 644
```

6. From your home directory, find ordinary files of size 0 (zero) in the `/tmp` directory, and ask if you can remove them.

```
$ find /tmp -type f -size 0 -ok rm -f "{}" \;
```

7. Display all the files created in your home directory using `ls`, `grep`, and today’s date. For example:

```
$ ls -l | grep 'Jun 9'
```

8. Search for the text string “other” in the `/etc/group` file, and have it display to the screen.

```
$ grep other /etc/group
```

9. Using the `grep` command, look for all lines in the `file4` file located in your home directory that do not contain the letter "M".

```
$ grep -v 'M' file4
```

10. Display all lines in the files `dante`, `file1`, and `dante_1` that contain the pattern `he`.

```
$ grep he dante file1 dante_1
```

11. Display all the lines in the file `file4` that contain the pattern either `Sales` or `Finance`.

```
$ egrep 'Sales|Finance' file4
```

12. Sort the lines in the file `fruit` alphabetically.

```
$ sort fruit
```

13. Sort the file `fruit` in reverse order.

```
$ sort -r fruit
```

Note – To create a file for use in the next step, execute:

```
$ ls -la > ls.output
```

14. Sort the `ls.output` file. Produce a numerical listing by size of the files, in reverse order. What command did you use?

```
$ sort -rn +4 ls.output
```

15. Perform a multilevel sort of the `ls.output` file that places the data in chronological order, then alphabetically by name. What command did you use?

```
$ sort +5M +6n +8d ls.output
```

If time of day created is an issue, the answer would be:

```
$ sort +5M +6n +7n +8d ls.output
```

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Use the `find` command to locate files in the Solaris Operating Environment directory tree using specific search criteria
- ☐ Use the `cmp` and `diff` commands to compare the contents of text files for differences
- ☐ Sort the content of text files in alphabetical and numerical order using the `sort` command
- ☐ Search for regular expressions in the contents of one or more files using the commands `grep`, `egrep`, and `fgrep`

Objectives

Upon completion of this module, you should be able to:

- Display file and directory permissions
- Define the standard permission types (read/write/execute)
- Use the `chmod` command to change permissions with symbolic mode or octal mode values
- Determine the default permissions assigned to newly created files and directories with `umask`

Additional Resources



Additional resources – The following reference can provide additional details on the topics discussed in this module:

- *Solaris Advanced User's Guide*, Part Number 801-6615-10

Security Overview

The most important function in a secure system is the ability to deny access to unauthorized users while maintaining access for authorized users. Maintaining a secure system is a primary function of the system administrator, but it is a responsibility of the user as well.

The Solaris Operating Environment provides two basic measures to prevent unauthorized access to a system and to protect its data.

The first measure is to authenticate a user's login by verifying the login ID name and password existing in `/etc/passwd` and `/etc/shadow`.

The second measure is to automatically protect file and directory access by placing a standard set of access permissions when files and directories are created.

Note – The Solaris Operating Environment also provides a special user account on every system, called `root`. The `root` user, often referred to as *superuser*, has complete access to every user account and all files and directories. The `root` user can override the permissions placed on files and directories.

Viewing File and Directory Permissions

To view the permissions on files and directories, use the `ls -l` command.

The first field of information that the `ls -l` command displays defines the type of file, followed by three distinct classes of users and their access permissions.

- File type — Identifies whether the item is a directory or a file
- User — Lists access permissions for the owner
- Group — Lists access permissions for a group of users; defined by the system administrator
- Others (public) — Lists access permissions for all other users

See Figure 6-1 for an illustration of permissions for each class of user.

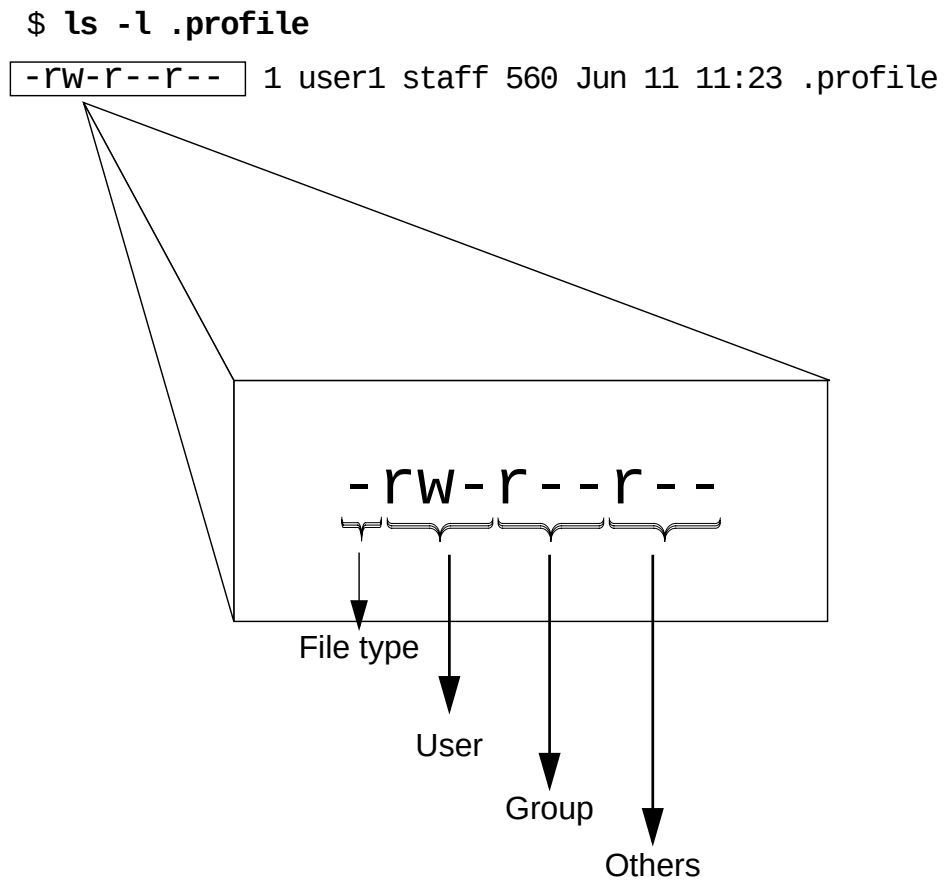


Figure 6-1 Permissions for Each Class of User

Permission Categories

The following sections describe the permission categories.

File Type

The first character in the `ls -l` listing defines the *file type*.

A directory file type is represented by the letter `d`.

An ordinary file's file type is represented by a hyphen (`-`).

A hyphen, sometimes referred to as a dash, located anywhere else in the permission set indicates that a particular permission is denied.

User (Owner) Permissions

The next three characters are the *owner's permissions*. These indicate what type of access the owner has on the file. For example, `user1`, the owner of this file, has read and write permissions.

Group

The second set of three characters are the *group permissions*. These identify the permissions being granted or denied for each user who is a member of the group that owns this file.

A group is a set of users who need to access the same files. All users in the same group can access each others' files based on these group permissions.

The system administrator creates and maintains the groups in the `/etc/group` file and assigns users to groups according to shared file access.

In the example, the file belongs to a group called `staff`, and all users who are members of this group have permission to read this file.

Others (Public)

The third set of three characters are the *other permissions*. These define the permissions for everyone else.

Other is any user who is not the file owner, nor a member of the group that owns the file, but who has access to the system.

Determining Access to a File or Directory

Access to a file or a directory is determined by the user identification number (UID) and group identification number (GID).

- UID — Identifies the user who created the directory or file and determines ownership.
- GID — Identifies the group of users who own the directory or file. A file or directory can belong only to one group at a time.

All files and directories contain a UID and GID number. The Solaris Operating Environment uses these numbers to track file and directory ownership and group membership.

To view these UID and GID numbers, use the `ls -n` command:

```
$ ls -n
total 108
-rw-r--r--  1 11001    10          0 Feb 22 14:51 brands
-rw-r--r--  1 11001    10        1320 Feb 22 14:51 dante
-rw-r--r--  1 11001    10        368 Feb 22 14:51 dante_1
```

Process for Determining Permissions

When a user attempts to access a file or directory, the Solaris Operating Environment compares the UID of the user to the UID of the file or directory being accessed.

If the UID values match, then the owner permissions are used to access the file or directory.

If the UID numbers do not match, then the user's GID and the GID number of the file or directory are compared. If these values match, the group permissions apply.

If the GID numbers do not match then the *other* category of permissions are used to determine file or directory access.

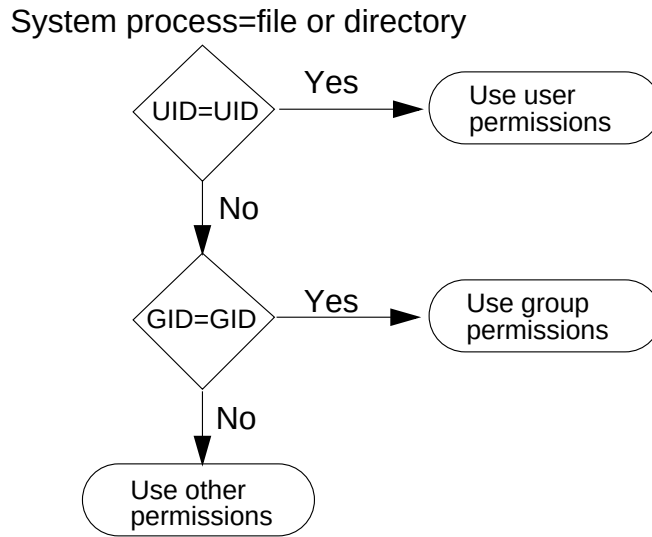


Figure 6-2 Process for Determining Permissions

Types of Permissions

File and directory access is protected by a standard set of default permissions, which are automatically assigned by the Solaris Operating Environment when a file or directory is created.

Permissions control who can do what to a file or directory and are represented by the characters *r* (read), *w* (write), *x* (execute), and *-* (denied).

When a user creates a new file or directory, by default, the Solaris Operating Environment automatically assigns the permissions on a file as *rw-r--r--* and on a directory as *rw-r--r--*.

Note – Execute permissions can be placed on files by the user with the *chmod* command, but these permissions are never assigned by the system when a file is created.

The read/write/execute permissions are interpreted differently when assigned to an ordinary file than when assigned to a directory. See Table 6-1 for a description of the differences.

Table 6-1 Permissions and Corresponding Symbols

Permission	Permission Symbol	File	Directory
Read	r	File can be displayed or copied	Contents can be listed with the ls command
Write	w	File contents can be modified	Files can be added or deleted
Execute	x	File can be executed (shell scripts or executables only)	User can cd to the directory and issue the ls -l command on the directory

Note – For a directory to be of general use, you must set the read and execute permissions.

The following are samples of different types of permissions set on files and directories.

- This file is read/write/execute for the file owner only. All other permissions for group and others are denied:
-rwx-----
- This directory is read/execute for the directory owner and the group only:
dr-xr-x---
- This file is read/write/execute for file owner and read/execute for the group and all others:
-rwxr-xr-x

Changing Permissions

You can modify the permissions set on newly created files or directories using the `chmod` command. This command can be used by either the owner of the file or directory or superuser to change permissions.

The `chmod` command can modify permissions specified in either *symbolic* mode or *octal* mode.

- Symbolic mode uses combinations of letters and symbols to add or remove permissions for each class of users.
- Octal mode uses numbers to represent each permission, often referred to as *absolute* mode.

Figure 6-3 illustrates the relationship between permissions and files.

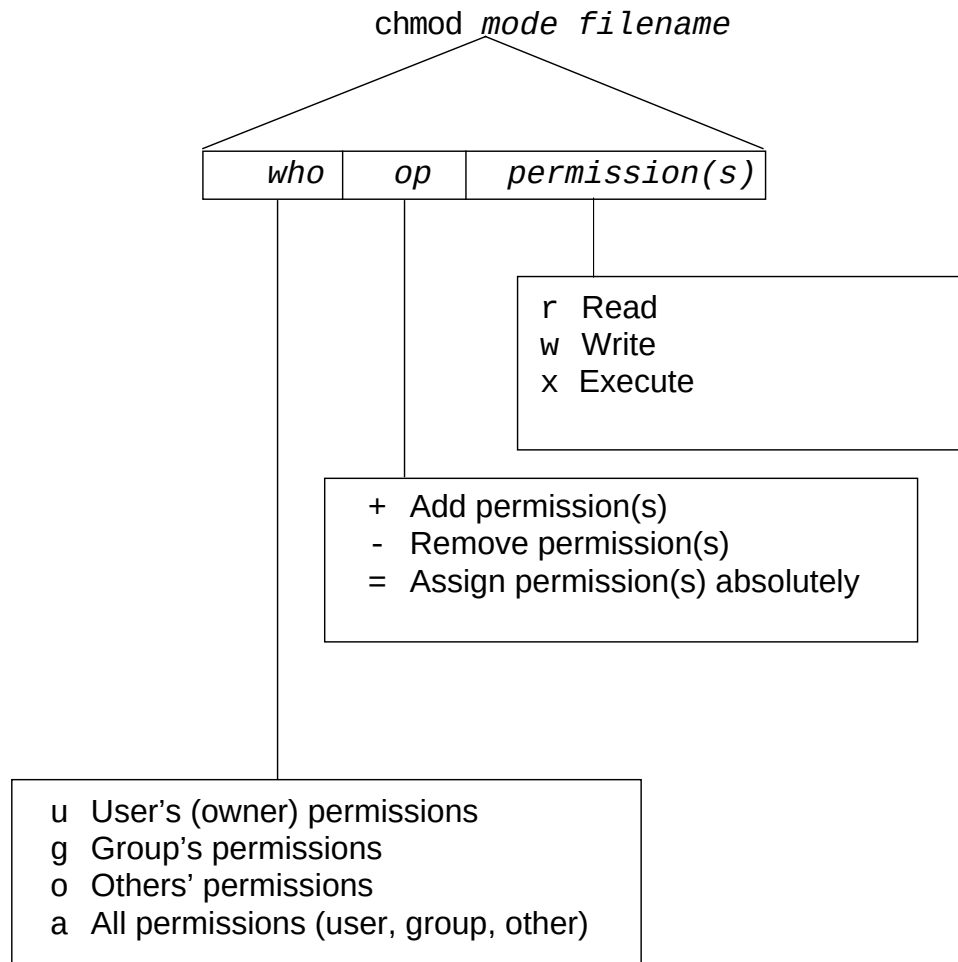


Figure 6-3 Symbolic Mode Command Format

Changing Permissions With Symbolic Mode

The following examples show how to modify permissions on files and directories using symbolic mode.

- Remove the read permission for other:

```
$ ls -l dante
-rw-r--r--  1 user1  staff    1320 Feb 22 14:51 dante
$
$ chmod o-r dante
$ ls -l dante
-rw-r-----  1 user1  staff    1320 Feb 22 14:51 dante
$
```

- Remove the read permission on group:

```
$ chmod g-r dante
$ ls -l dante
-rw----- 1 user1  staff      1320 Feb 22 14:51 dante
$
```

- Add an execute permission for the user (owner), and a read permission for the group and other:

```
$ chmod u+x,go+r dante
$ ls -l dante
-rwxr--r-- 1 user1  staff      1320 Feb 22 14:51 dante
$
```

- Assign read and write permissions for user, group, and other:

```
$ chmod a=rw dante
$ ls -l dante
-rw-rw-rw- 1 user1  staff      1320 Feb 22 14:51 dante
$
```

Octal (Absolute) Mode

You specify Octal mode using a combination of octal numbers. The numbers used include 0 to 7.

Command Format

```
chmod octal_mode filename
```

Each permission is represented by its own octal number.

Table 6-2 Permissions Assigned Octal Value

Octal Value	Permissions
4	Read
2	Write
1	Execute

Each octal number represents a permission set as shown in Table 6-3.

Table 6-3 Octal Digits for Permission Sets

Octal Value	Permission Sets
7	r w x
6	r w -
5	r - x
4	r - -
3	- w x
2	- w -
1	- - x
0	- - -

By combining octal numbers, a user can quickly modify the permissions for each class of users. The first octal number defines *owner permissions*, the second octal number defines *group permissions*, and the third octal number defines *other permissions*.

Table 6-4 Combined Values and Permissions

Octal Mode	Permissions
644	rw-r--r--
751	rwxr-x--x
775	rwxrwxr-x
777	rwxrwxrwx

When using octal mode with the `chmod` command, you must use all three numbers, one for *each* access category: user (owner), group, and other.

Changing Permission With Octal Mode

The following examples show how to modify permissions on files and directories using octal (absolute) mode.

Note – Each example builds on the resulting permissions from the previous example.

- To give owner, group, and others read and execute access only:

```
$ ls -l dante
-rw-rw-rw-  1 user1    staff      1320 Feb 22 14:51 dante
$ chmod 555 dante
$ ls -l dante
-r-xr-xr-x  1 user1    staff      1320 Feb 22 14:51 dante
$
```

- To change user and group permissions to include write access:

```
$ chmod 775 dante
$ ls -l dante
-rwxrwxr-x  1 user1    staff      1320 Feb 22 14:51 dante
$
```

- To change group permission to read and execute only:

```
$ chmod 755 dante
$ ls -l dante
-rwxr-xr-x  1 user1    staff      1320 Feb 22 14:51 dante
$
```


Default Permissions

The following sections describe the different types of default permissions.

The umask Filter

The umask filter controls the default permissions assigned to newly created files and directories. The umask is a three-digit octal value that refers to read/write/execute permissions for owner, group, and other.

Displaying the umask

```
$ umask
022
$
```

The first digit determines the default permissions for the owner; the second digit determines the default permissions for the group; and the third digit determines the default permissions for other.

In the Solaris Operating Environment, the default umask value is 022.

The umask operates as a filter to affect the initial permission values specified by the system during the creation of a file or directory.

The initial permission value specified by the system for a file creation is 666 (rw-rw-rw-).

The initial permission value specified by the system for a directory creation is 777 (rwxrwxrwx).

The umask value is automatically filtered, or subtracted from the initial permission value to determine the default permissions assigned to newly created files and directories.

Understanding the umask Filter

Another way to determine what the default permissions will be when creating new files is to take the initial value specified by the system; represented by symbolic mode: `rw-rw-rw-`.

Corresponds to read/write access for the user, group, and other, and represented in octal mode as:

`42-42-42-`

Use the default `umask` value of `022`, which removes (or denies) write permission for group and other.

For example:

`rw-rw-rw-` Initial value specified by the system for a new file

`---w--w-` Default `umask` filter value to be subtracted

`rw-r--r--` Default permissions assigned to newly created files

When the access permissions to be denied are masked out from the initial value, the default permissions assigned to the new files remain.

All newly created files are assigned read/write access for the user, and read access for group and other: `rw-r--r--`

You can apply this same process when determining what the default permissions are when creating new directories. In this case, take the initial value, specified by the system, represented by symbolic mode as: `rwxrwxrwx`.

Corresponds to read/write/execute access for the user, group, and others, and represented in octal mode as:

`421421421`

Use the default `umask` value of `022`, which removes (or denies) write permission for group and other.

For example:

`rw-rw-rw-` Initial value specified by the system for a new directory

`---w--w-` Default `umask` filter value to be subtracted

`rw-r-xr-x` Default permissions set for newly created directories

When the access permissions to be denied are masked out from the initial value, the default permissions assigned to the new files remain.

All newly created directories are assigned read/write/execute access for the user, and read/execute access for group and other:
`rw-r-xr-x`.

Changing the umask Value

Some user's require a more secure `umask` value of 027, which assigns the following access permissions to newly created files and directories.

- Files are given read/write permissions for the owner; read permission for the group; and denies all permissions for other, or `rw-r-----`.
- Directories are given read/write/execute permissions to the owner; read/execute permissions to the group; and denies all access permissions for other, or `rw-r-xr-x`.

You can also change the `umask` to a new value on the command line.

Changing umask

Changing the `umask` value on a file to 027 (`rw-r-----`) gives read and write permission to the owner of a new file, read permissions to group, and no permissions to others.

Changing the `umask` value on a directory to 027 (`rwxr-x---`) gives all permission to the owner, read and execute permissions to group, and no permissions to others.

For example, to change the `umask` value to 027, and then verify the new value has been set, execute the following:

```
$ umask 027
$ umask
027
$
```

This new `umask` value affect only those files and directories that are created from this point forward. However, because the `umask` value was changed on the command line, if the user logs out of the system, the new value (027) is replaced by the old value (022) on subsequent logins.

To retain the new `umask` value, you must place it in one of the shell initialization files.

Exercise: Changing File Permissions



Exercise objective – The purpose of this lab is to give you practice reading permissions on files and changing permissions using symbolic or octal notation.

Tasks

Complete or answer the following:

- Execute the following commands:

```
$ mkdir ~/perm
$ cd /etc
$ cp group passwd motd vfstab dumpdates shadow ~/perm
$ cd
$ cp -r /etc/skel perm
```

When trying to copy /etc/shadow, an error message was displayed. Why?

- Change directory to perm and complete the following table:

File or Directory	User Permissions	Group Permissions	Other Permissions	Octal Value
group	rw-			
passwd			r--	
vfstab	rw-			
skel				755

- Create a new file and a new directory.

▼ What are the default permissions given to the new file?

▼ What are the default permissions given to the new directory?

4. In a directory with permissions of `drwxr-xr--`, who can perform the following actions with the files shown below? Put an X next to each allowed action.

`-rw-r--r--`

user: read___ modify___ delete___ execute___

group: read___ modify___ delete___ execute___

others: read___ modify___ delete___ execute___

`-rwxrwxr-x`

user: read___ modify___ delete___ execute___

group: read___ modify___ delete___ execute___

others: read___ modify___ delete___ execute___

5. Using symbolic mode, add write permission for the group to the `motd` file.

6. Using symbolic mode, remove the group read permission on the file `dumpdates`.

7. Using octal mode, change the permissions on the file `motd` to `-rwxrw----`.

8. Using octal mode, change the permissions on the file `group` to add write permission for others.

9. Why is execute not a default permission for a newly created file?

10. Create a new file called `memo` in your `practice` directory.

11. Remove the read permission for the owner from the file `memo` in the `practice` directory. Use either symbolic or octal mode.

What happens when you try to use the `cat` command to view the `memo` file?

What happens when you try to copy the `memo` file?

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

Complete the following steps:

1. Execute the following commands:

```
$ mkdir ~/perm
$ cd /etc
$ cp group passwd motd vfstab dumpdates shadow ~/perm
$ cd
$ cp -r /etc/skel perm
```

When trying to copy /etc/shadow an, error message was displayed. Why?

Because, as a user who is not part of the sys group, you have no permissions on this file.

2. Change directory to perm and complete the following table:

```
$ cd perm
$ ls -l
```

File or Directory	User Permissions	Group Permissions	Other Permissions	Octal Value
group	rw-	r--	r--	644
passwd	rw-	r--	r--	644
vfstab	rw-	r--	r--	644
skel	rwX	r-X	r-X	755

3. Create a new file and a new directory.

▼ What are the default permissions given to the new file?

rw-r--r--

▼ What are the default permissions given to the new directory?

rwxr-xr-x

4. In a directory with permissions of drwxr-xr--, who can perform the following actions with the files shown below? Put an X next to each allowed action.

```
-rw-r--r--
user:   read X   modify X   delete X   execute__
group:  read X   modify__  delete__   execute__
others: read__   modify__  delete__   execute__
```

```
-rwxrwxr-x
user:   read X   modify X   delete X   execute X
group:  read X   modify X   delete__   execute X
others: read__   modify__  delete__   execute__
```

Using symbolic mode, add write permission for group to the `motd` file.

```
$ chmod g+w motd
```

5. Using symbolic mode, remove the group read permission on the file `dumpdates`.

```
$ chmod g-r dumpdates
```

6. Using octal mode, change the permissions on the file `motd` to `rwxrw----`.

```
$ chmod 760 motd
```

7. Using octal mode, change the permissions on the file `group` to add write permission for others.

```
$ chmod 646 group
```

8. Why is execute not a default permission for a newly created file?

Most files are not binary or executable scripts. Having execute as a default permission for a file would cause the system to see all new files as executables.

9. Create a new file called `memo` in your `practice` directory.

```
$ touch ~/practice/memo
```

10. Remove the read permission for the owner from the file `memo` in the `practice` directory. Use either symbolic or octal mode.

```
$ chmod u-r ~/practice/memo or  
chmod 244 ~/practice/memo
```

What happens when you try to use the `cat` command to view the `memo` file?

You cannot use the `cat` command because the read permission has been removed for the user. Even though you are part of the group, the permissions are looked at in the order they appear.

What happens when you try to copy the `memo` file?

You cannot copy the file because, as in step 4, the user has no read permission.

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

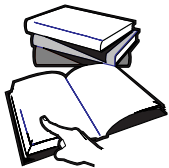
- ☐ Display file and directory permissions
- ☐ Define the standard permission types (read/write/execute)
- ☐ Use the `chmod` command to change permissions with symbolic mode or octal mode values
- ☐ Determine the default permissions assigned to newly created files and directories with `umask`

Objectives

Upon completion of this module, you should be able to:

- Define the three modes of operation used by the vi editor
- Start the vi editor
- Position and move the cursor in vi
- Switch between vi modes
- Create and delete text
- Copy or move text
- Set vi options
- Perform search and replace functions within vi
- Exit the vi editor

Additional Resources



Additional resources – The following reference can provide additional details on the topics discussed in this module:

- *Solaris 8 Reference Manual Collection: User Commands*, Part Number 806-0624-10

Introducing vi

The visual display (**vi**) editor is an interactive editor used to create or modify text files.

All text editing with the **vi** editor takes place in a buffer. Changes can either be written to the disk or be discarded.

For those who intend on becoming system administrators, it is important to know how to use **vi**. You need to know how to use the **vi** editor in case the windowing system is not available.

vi Modes

The *vi* editor is a command-line editor that has three basic modes of operation:

- Command mode
- Edit mode
- Last line mode

Command Mode

This is the default mode for *vi*. In this mode, you can enter commands to delete, change, copy and move text; position the cursor; search for text strings; or exit *vi*.

Edit Mode

In this mode, you can enter text into a file. To instruct *vi* to enter edit mode, type one of the following three commands:

- i (insert)
- o (open)
- a (append)

Last Line Mode

While in command mode, you can do advanced editing commands by typing a colon (:), which places you at the bottom line of the screen. This is called last line mode. However, all commands are initiated from command mode.

Switching Modes

By typing an `i`, `o`, or `a` command, `vi` leaves the default command mode and enters edit mode.

In edit mode, text is not interpreted as commands by `vi`. Now, everything you type entered into the file as text.

When you have finished entering text in the file, you can return `vi` to command mode by pressing the Escape key. Once back in command mode, you can then save the file and quit `vi`.

For example:

1. Type `vi filename` to create a file.
2. Type the command `i` to insert text.
3. Press the Escape key to return to command mode.
4. Type `:wq` to write and save the file and exit `vi`.

Invoking vi

To create a new file, invoke vi with a new file name by typing commands to create, edit, or view a file.

Command Format

```
vi option(s) filename  
view filename
```

Input Commands

To insert or append text, use the options in Table 7-1.

Table 7-1 Append and Insert Commands for vi

Command	Meaning
a	Appends text after the cursor
A	Appends text at the end of the line
i	Inserts text before the cursor
I	Inserts text at the beginning of the line
o	Opens a new line below the cursor
O	Opens a new line above the cursor

Note – The vi editor is case sensitive, so use the specified case when using these cursor positioning commands.

Positioning Commands

Table 7-2 shows the key sequences that control cursor movement in the vi editor.

Table 7-2 Key Sequences

Command	Meaning
h, ←, or Back Space	Moves left one character
j or ↓	Moves down one line
k or ↑	Moves up one line
l, →, or Space bar	Moves right (forward) one character
w	Moves forward one word
b	Moves back one word
e	Moves to the end of the current word
\$	Moves to the end of the line
0 (zero) or ^	Moves to the beginning of the line
Return	Moves down to the beginning of the next line
Control-f	Pages forward one screen
Control-d	Scrolls down one-half screen
Control-b	Pages back one screen
Control-u	Scrolls up one-half screen
Control-l	Refreshes the screen

Editing Commands

The following sections describe the commands in the `vi` editor.

Deleting Text

To delete text, use the options in Table 7-3.

Table 7-3 Text Deletion Commands for `vi`

Command	Meaning
<code>x</code>	Deletes a character at the cursor
<code>dw</code>	Deletes word (or part of the word to the right of the cursor)
<code>dd</code>	Deletes the line containing the cursor
<code>D</code>	Deletes the line to the right of the cursor (from cursor position to the end of the line)
<code>:5,10d</code>	Deletes lines 5 through 10

Undoing, Repeating, and Changing Text Commands

To change text, undo a change, or repeat an edit function, use the commands in Table 7-4. Many of these commands change `vi` to edit mode. To return to command mode, press the Escape key.

Table 7-4 Edit Commands for `vi`

Command	Meaning
<code>cw</code>	Changes word (or part of word) at the cursor location to the end of the word
<code>R</code>	Overwrites or replaces characters on the line
<code>C</code>	Changes from cursor to end of the line
<code>s</code>	Substitutes string for character(s)
<code>r</code>	Replaces the character at the cursor with one other the character

Table 7-4 Edit Commands for vi (Continued)

Command	Meaning
J	Joins the current line and the line below
xp	Transposes the character at the cursor and the character to the right
~	Changes the case of the letter (upper or lower) at the cursor
u	Undoes the previous command
U	Undoes all changes to the current line
:u	Undoes the previous last-line command
:r <i>filename</i>	Inserts the (read) file at the line after the cursor

To search and replace text, use the following options in Table 7-5.

Table 7-5 Search and Replace Commands

Command	Meaning
/string	Searches forward for the <i>string</i>
?string	Searches backward for the <i>string</i>
n	Finds the next occurrence of the string
N	Finds the previous occurrence of the string
:%s/old/new/g	Searches and replaces globally

Copying and Pasting Text

The copy commands write the copied text into a temporary buffer. The paste commands read the text from the temporary buffer and write the text into the current document at the specified location. To copy and paste text, use the options in Table 7-6.

Table 7-6 Copy and Paste Commands

Command	Meaning
yy (lowercase)	Yanks a copy of line
p (lowercase)	Puts yanked or deleted text after the current position
P (uppercase)	Puts yanked or deleted text before the current position
:1,3 co 5	Copies lines 1 through 3 and puts them after line 5
:4,6 m 8	Moves lines 4 through 6 to line 8 (line 6 becomes line 8; line 5 becomes line 7, and line 4 becomes line 6)

Note – Both delete and yank write to a buffer. When yanking, deleting, and pasting, the put commands insert the text differently depending on whether you are pasting words or lines.

Saving and Quitting Files

To save and quit a file, use the options in Table 7-7.

Table 7-7 Save and Quit Commands

Command	Meaning
:w	Saves changes (write buffer)
:w <i>new_filename</i>	Writes buffer to <i>new_filename</i>
:wq	Saves changes and quits vi
:x	Saves changes and quits vi

Table 7-7 Save and Quit Commands (Continued)

Command	Meaning
ZZ	Saves changes and quits vi
:q!	Quits without saving changes
:wq!	Saves changes and quits vi. (The ! overrides read only permissions for the owner of the file only.)

Customizing Your vi Session

The vi editor includes options for customizing edit sessions, such as:

- Displaying line numbers
- Displaying invisible characters, such as tab and end-of-line characters

You use the `set` command in command mode to control these options.

Table 7-8 Edit Session Customization Commands

Command	Meaning
<code>:set nu</code>	Shows line numbers
<code>:set nonu</code>	Hides line numbers
<code>:set ic</code>	Instructs searches to ignore case
<code>:set noic</code>	Instructs searches to be case sensitive
<code>:set list</code>	Displays invisible characters, such as tab and end-of-line
<code>:set nolist</code>	Turns off the display of invisible characters
<code>:set showmode</code>	Displays current mode of operation
<code>:set noshowmode</code>	Turns off mode display
<code>:set</code>	Displays all vi variables set
<code>:set all</code>	Displays all possible vi variables and their current settings

You can also place these options in a file you create in your home directory called `.exrc`. The set options are placed in this file, without the preceding colon, one command to a line. Once the `.exrc` file exists, it is read by the system each time you open a vi session.

To find a particular line, use the options in Table 7-9.

Table 7-9 Positioning Commands

Command	Meaning
G	Goes to last line of the file
1G	Goes to first line of the file
:21	Goes to line 21
21G	Goes to line 21

To clear the screen or insert files, use the options in Table 7-10.

Table 7-10 Refreshing Commands

Command	Meaning
Control-L	Refreshes the screen
:r <i>filename</i>	Inserts (reads) the file at line after the cursor

Exercise: Using the **vi** Editor



Exercise objective – In this exercise you will practice creating and modifying the files using the **vi** editor.

Tasks

Complete the following step:

1. In your **home** directory, there should be a file called **tutor.vi**. Make sure you are currently in your **home** directory, then open this file with the command:

```
$ vi tutor.vi
```

This will open a **vi** tutorial file.

2. Complete the lessons outlined in this tutorial.

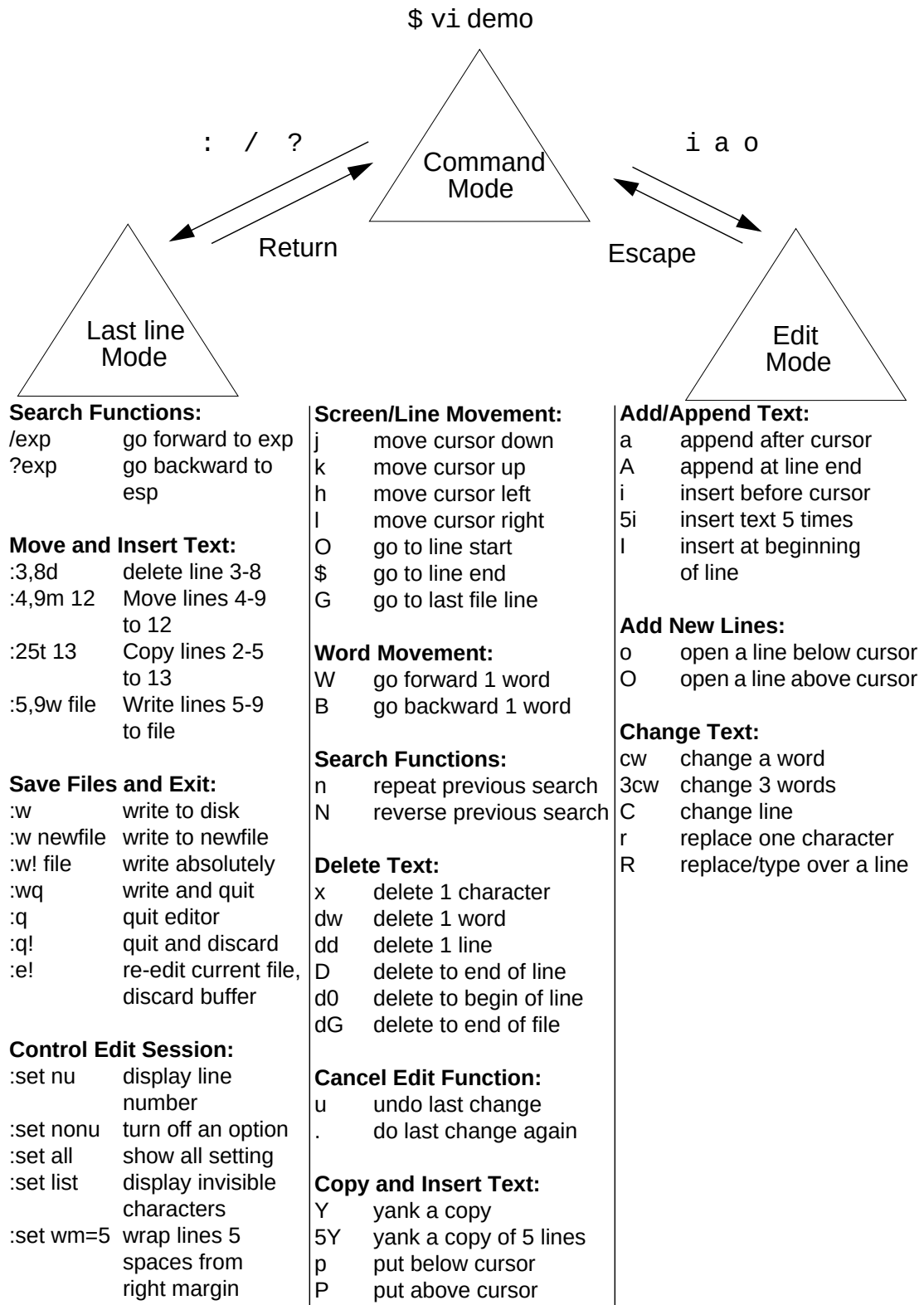


Figure 7-1 Quick Reference Chart for Using vi

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Define the three modes of operation used by the `vi` editor
- ☐ Start the `vi` editor
- ☐ Position and move the cursor in `vi`
- ☐ Switch between `vi` modes
- ☐ Create and delete text
- ☐ Copy or move text
- ☐ Set `vi` options
- ☐ Perform search and replace functions within `vi`
- ☐ Exit the `vi` editor

Objectives

Upon completion of this module, you should be able to:

- Determine which commands are suitable for storing, viewing, or retrieving different types of files
- Demonstrate how to reduce the size of files and directories and store them to tape using the `compress` and `tar` commands
- Describe the steps for uncompressing or viewing a compressed file with the `uncompress` and `zcat` commands
- Use the `gzip` and `gunzip` commands to compress and uncompress files
- Use the `zip` command to package and compress multiple files and use `unzip` to uncompress a zipped archive file
- Compress and copy multiple files to a single archive file in one step using the `jar` command
- Copy and extract files from an archive file or tape device with the `cpio` command
- Discuss the benefits of using volume management when working with removable media devices, such as disks and CD-ROMs
- Demonstrate how to access data on disks and CD-ROMs
- Use the `eject` command to remove disks and CD-ROMs from device drives

Overview of Archive Commands

To safeguard your files and directories, you should archive copies of them onto some form of removable media, such as a tape cassette. You need the archived tapes to retrieve lost, deleted, or damaged files.

The commands available to easily store, locate, and retrieve files on a tape device or an archive file, include:

- **tar** — Creates and extracts files from a tape device or file archive.
- **compress** and **uncompress** — Compresses and uncompresses a file.
- **zcat** — Views a compressed file.
- **gzcat** — Uncompresses and views a file at the same time
Uncompresses a list of files on the command line and writes the uncompressed data to the screen.
- **zip** and **unzip** — Packages and compresses files and uncompresses files.
- **gzip** and **gunzip** — Compresses and uncompresses a file.
- **jar** — Packages and compresses multiple files to a single archive file.
- **cpio** — Copies and extracts files from a file archive or tape device.

Note – When archiving, you should to use relative pathnames.

Archiving Files Using the **tar** Command

The **tar** command archives and extracts files to and from a single file called a *tar file*. A tar file is usually a magnetic tape, but it can be any file.

Command Format

```
tar function(s) archive_file/tape_device filename(s)
```

Functions

For a list of the various functions used with the **tar** command, see Table 8-1.

Table 8-1 Functions for the **tar** Command

Function	Definition
c	Creates a new tar file.
t	Lists the table of contents of the tar file.
x	Extracts the specified files from the tar file.
f	Specifies the archive file or tape device. The default tape device is <code>/dev/rmt/0</code> .
v	Executes in verbose mode to list the contents of the tar file.

Creating, Viewing, and Retrieving a Directory from a Tape

To create a tape archive of the `user1` directory, using the default tape device, execute the following:

```
$ cd /export/home
$ tar cv user1
a user1/ 0 tape blocks
a user1/dante 106 tape blocks
a user1/dante_1 1 tape blocks
a user1/logfile 5 tape blocks
```

```
a user1/file2 1 tape blocks
<OUTPUT OMITTED>
```

To view the contents of the directory copied to tape, execute the following command:

```
$ tar t
user1/
user1/dante
user1/dante_1
user1/logfile
user1/file2
user1/file3
user1/file4
user1/fruit
user1/fruit2
user1/tutor.vi
user1/dir1/
user1/dir1/coffees/
user1/dir1/coffees/beans
user1/dir1/coffees/nuts
user1/dir1/coffees/beans.backup
<OUTPUT OMITTED>
```

If the user1 directory was accidentally deleted or damaged, you could retrieve the entire directory by extracting its contents from the archive tape by execute the following command:

```
$ cd /export/home
$ tar xv user1
x user1, 0 bytes, 0 tape blocks
x user1/dante, 54120 bytes, 106 tape blocks
x user1/dante_1, 368 bytes, 1 tape blocks
x user1/logfile, 2483 bytes, 5 tape blocks
x user1/file2, 105 bytes, 1 tape blocks
x user1/file3, 218 bytes, 1 tape blocks
x user1/file4, 137 bytes, 1 tape blocks
x user1/fruit, 56 bytes, 1 tape blocks
x user1/fruit2, 57 bytes, 1 tape blocks
x user1/tutor.vi, 28738 bytes, 57 tape blocks
x user1/dir1, 0 bytes, 0 tape blocks
x user1/dir1/coffees, 0 bytes, 0 tape blocks
<OUTPUT OMITTED>
```

Note – Typically, home directories are scheduled for tape archive on a nightly basis by the system administrator. However, you should also archive the contents of your home directory on a regular basis. If a file is accidentally deleted, you can restore it quickly.

The following examples describe the steps for creating, viewing, and retrieving a directory from a secondary tape drive attached to the system rather than the default tape drive.

```
$ cd /export/home
$ tar cvf /dev/rmt/1 user1
$
$ tar tvf /dev/rmt/1
$
$ cd /export/home
$ tar xvf /dev/rmt/1 user1
```

Creating, Viewing, and Retrieving Files from an Archive File

The `tar` command is also used to create single archive files on disk to share with other users or for attaching to mail messages.

```
$ cd
$ tar cvf files.tar file1 file2 file3
a file1 2K
a file2 1K
a file3 1K
```

In this example, `file1`, `file2`, and `file3` are stored in an archive file call `files.tar`.

To view the contents of this archive file, execute the following command:

```
$ tar tf files.tar
file1
file2
file3
$
```

The following example shows how to extract files from this archive file for placement back into the current directory.

```
$ tar xvf files.tar
tar: blocksize = 11
x file1, 1696 bytes, 4 tape blocks
x file2, 156 bytes, 1 tape blocks
x file3, 218 bytes, 1 tape blocks
$
```

Compressing Files Using the `compress` Command

The `compress` command is used to reduce the size of a file. This is particularly useful when working with large files, which can consume disk space and take longer than smaller files to transfer from one system to another over a network.

Note – The amount of compression depends on the type of file being compressed, but typically a text file is reduced by 60 to 80 percent.

When a file is compressed, the named file is replaced by a new file with a `.Z` extension. The ownership and modification time of the original file remains the same, even though the file's contents are completely changed.

Command Format

`compress filename`

Compressing a File

This example compresses a file called `files.tar`:

```
$ compress -v files.tar  
files.tar: Compression: 70.20% -- replaced with files.tar.Z
```

The newly compressed file, which replaces `files.tar` is now called `files.tar.Z`.

This naming convention (`.Z`) reminds users the file is compressed and should not be viewed or printed without first decompressing it.



Caution – Compressing files that have already been compressed makes file sizes larger — not smaller.

Uncompressing Files Using the uncompress Command

The `uncompress` command restores a compressed file back to its original state.

Command Format

```
uncompress -c filename
```

Uncompressing a File

The following example uncompresses the file `files.tar.Z` and replaces it with the original file named `files.tar`.

```
$ uncompress files.tar.Z
```

Viewing the Contents of a Compressed File

Using the `uncompress` command with the `-c` option displays the contents of a compressed file on the screen, without uncompressing the `.Z` file.

```
$ uncompress -c file.tar.Z | more
```

Viewing Files Using the zcat Command

The `zcat` command also allows you to view a file that has been compressed with the `compress` command.

The `zcat` command interprets the compressed data and displays the contents of the file as though it were not compressed. After running `zcat`, the contents of the compressed file are unchanged; it is still stored on the disk in compressed form.

Command Format

```
zcat filename
```

Viewing the Contents of a Compressed File

To view the contents of a compressed file, execute the following:

```
$ zcat file.tar.Z | tar tvf -
```

Compressing Files Using the `jar` Command

The `jar` command is similar to the `tar` command, but it compresses the named file(s) in the same step. This command copies and compresses multiple files into a single archive file. Zip programs can read `jar` files.

Note – This was originally created for Java™ programmers to allow downloading multiple files at one time rather than issuing a download for each separate file. The `jar` command is standard with the Solaris 8 Operating Environment, and it is available on any system that has Java Virtual Machine™ (JVM™) installed.

Command Format

The syntax for the `jar` command is almost identical to the syntax for the `tar` command.

```
jar options output_file filename(s)/directory(s)
```

Options

See Table 8-2 for the options used with the `jar` command.

Table 8-2 Options for the `jar` Command

Option	Definition
<code>M</code>	Does not create manifest.
<code>c</code>	Creates a new <code>jar</code> file.
<code>t</code>	Lists the table of contents of a <code>jar</code> file.
<code>x</code>	Extracts the specified files from the <code>jar</code> file .
<code>f</code>	Specifies the <code>jar</code> file (<code>/tmp/file.jar</code>) or tape drive (<code>/dev/rmt/x</code>) if other than default.
<code>v</code>	Executes in verbose mode.

Note – You do not have to use a dash (-) before options when issuing the `jar` command.

Adding All the Files in a Directory to an Archive

The following example copies and compresses multiple files into the single archive file called `bundle.jar`.

```
$ ls
brands      dir4      file.3    fruit     num.list
dante      dir5      file1     fruit2    perm
dante_1    feathers  file2     games     Reports
dir1       feathers_6 file3     list      ski.places
dir2       file.1    file4     monthly   tutor.vi
dir3       file.2    files.tar.Z newdir     update.list
$
$ jar cvf /tmp/bundle.jar *
added manifest
adding: dante (in=54120) (out=1095) (deflated 97%)
adding: dante_1 (in=368) (out=242) (deflated 34%)
adding: dir1/ (in=0) (out=0) (stored 0%)
adding: dir1/coffees/ (in=0) (out=0) (stored 0%)
adding: dir1/coffees/beans (in=12288) (out=3161) (deflated 74%)
adding: dir1/coffees/nuts (in=0) (out=0) (stored 0%)
adding: dir1/coffees/beans.backup (in=0) (out=0) (stored 0%)
adding: dir1/fruit/ (in=0) (out=0) (stored 0%)
adding: dir1/trees/ (in=0) (out=0) (stored 0%)
adding: dir1/feathers (in=51) (out=50) (deflated 1%)
adding: dir2/ (in=0) (out=0) (stored 0%)
adding: dir2/recipes/ (in=0) (out=0) (stored 0%)
adding: dir2/beans/ (in=0) (out=0) (stored 0%)
adding: dir2/notes (in=0) (out=0) (stored 0%)
adding: dir3/ (in=0) (out=0) (stored 0%)
adding: dir3/planets/ (in=0) (out=0) (stored 0%)
adding: dir3/planets/mars (in=68) (out=61) (deflated 10%)
adding: dir3/planets/pluto (in=42) (out=44) (deflated -4%)
adding: dir4/ (in=0) (out=0) (stored 0%)
<output omitted>
```

Note – The `jar` command is not intended for backing up symbolic links. Error messages displayed if you attempt this.

Other Tools

A more effective pair of compression tools are `zip` and `gzip`. These commands tend to create smaller compressed files.

Using the zip Command

The `zip` command is similar to the `jar` command in that it compresses named files. The `zip` command is a UNIX utility, while `jar` is a Java application tool.

When `zip` compresses a file, it adds the extension `.zip` to the file name.

Note – You can type `zip` or `unzip` on the command line to see a list of options used with each command.

Command Format

`zip filename filename`

For example:

```
$ zip filename1 filename2 filename3 filename4
adding: filename2 filename3 filename4 (deflated 65%)
$
$ ls -l
filename1
filename1.zip
filename2
filename3
filename4
```

Where `filename1.zip` is the packaged and compressed zip file.

Restoring a zip File

To restore a zip file to its original state, execute the unzip command:

```
$ unzip file.zip
```

Note – You can uncompress jar files using the unzip command. This is similar to how zip files are uncompressed with the unzip command.

Using the gzip Command

When gzip compresses a file, it adds the extension .gz to the file name.

Command Format

```
gzip filename filename filename
```

For example:

```
$ gzip filename1 filename2 filename3 filename4
$
$ ls -l
filename1.gz
filename2.gz
filename3.gz
filename4.gz
```

Note – The gzip utility is useful for working with Solaris Operating Environment patches.

Restoring a gzip File

To restore a file that has been compressed with gzip, use the gunzip command.

```
$ gunzip file.gz
```

Note – Although the `gzip` and `gunzip` executables are included in the Solaris 8 Operating Environment, they were not included in the Solaris 7 Operating Environment and earlier versions. However, you can download these executables from the `gzip` Web home page at <http://www.gzip.org/>.

Using the `cpio` Command

The `cpio` (copy in/copy out) command archives or extracts a file to a tape (or a single file).

Some advantages of the `cpio` command are:

- It packs data onto tape more efficiently than the `tar` command
- It skips over any bad spots in a tape when restoring files
- It can write files with different header formats for portability between different system types
- It creates multiple tape volumes
- It can archive files without changing the files' access time

Command Format

```
cpio options filename(s)
```

Options

Table 8-3 shows the options available for use with the `cpio` command.

Table 8-3 Options for the `cpio` Command

Options	Definition
<code>o</code>	Creates a file archive. Copies a list of files or pathnames to the tape device or file (copy out).
<code>i</code>	Extracts the file archive from the tape device or file (copy in).
<code>c</code>	Reads or writes header information in ASCII character form for portability.
<code>t</code>	Lists the table of contents for the files on the tape in the tape drive specified.
<code>v</code>	Prints a list of file names in a format similar to the <code>ls -l</code> command.

Table 8-3 Options for the `cpio` Command (Continued)

<code>M message</code>	Defines a message to inform the user when the end of the media is reached and what action is needed.
<code>p</code>	Reads from the tape device or file to get pathnames.
<code>a</code>	Resets the access times of files after they have been copied.
<code>I filename</code>	Reads the content of a file as an input archive.
<code>O filename</code>	Directs the output of the <code>cpio</code> command to a file.

Note – You must specify one of the following options: `o`, `i`, or `p` on the `cpio` command line.

Creating File Archives

The following example demonstrates how to use the `find` command with `cpio` to create an archive of the current directory contents and copy it to a file called `dir.list`.

```
$ find . | cpio -ocv -O dir.list
```

You can also use the `find` command with the `cpio` command to create an archive of only those files that have been changed within the last week, and copy it to a file called `modify.list`.

```
$ find . -mtime -7 | cpio -ocv -O modify.list
```

The following example shows how to view the list of file names in the archive files created above.

```
$ cpio -ivt -I dir.list
$ cpio -ivt -I modify.list
```

Copying All Files in a Directory to a Tape

This example describes how to use the `cpio` command to copy files and directories to a cartridge tape.

```
$ cd /export/home/user1
$ ls | cpio -oc -0 /dev/rmt/0
16 blocks
$
```

Listing the Files on a Tape

The following example shows how to list the table of contents of a tape.

```
$ cpio -civt -I /dev/rmt/0
```

The table of contents displays eight fields of information for each file contained on the tape. The first field shows the permissions in octal mode; the second field shows the owner of the file; the third field displays the number of characters (bytes) in the file; the fourth, fifth, sixth, and seventh fields show the month, date, time, and year the file was last modified, and the last field shows the name of the file.

Retrieving All Files from a Tape

To retrieve all files from a tape, you must first change to the directory where the files are to be placed, then you can execute the following command:

```
$ cpio -icv -I /dev/rmt/0
```

The Volume Management Feature

To store and retrieve files on floppy disks and CD-ROMs, the Solaris Operating Environment provides the volume management feature.

Volume management provides ordinary users with a standard method for handling data on floppy disks and CD-ROMs.

With this feature, you have access to your floppy disks and CD-ROMs automatically.

Note – If volume management is not running on a system, only a superuser can access disks and CD-ROMs.

The volume management service is invoked by a system daemon named `vold`. The superuser can stop and start this feature on any system, by administering the `vold` program. By default, volume management is always running on the system to automatically manage CD-ROMs and disks for ordinary users.

Detecting Removable Media Devices

When you insert a CD-ROM or disk into the appropriate drive slot on the system, the volume manager needs to detect the new device to provide access.

Volume management provides automatic detection of CD-ROMs. However, for floppy disks, the volume manager does not detect the presence of a new disk. It needs to be informed every time you insert a disk in the drive.

Note – Automatic detection of disks would cause excessive reads, which would quickly wear out the floppy disk drive.

To inform volume manager that you have inserted a disk you invoke the `volcheck` command.

Command Format

```
volcheck options device_pathname
```

For example, to ask volume management to examine the floppy disk drive for newly inserted media, execute the following:

```
$ volcheck -v /dev/disk
```

The `volcheck` command reports back with either one of the following messages.

- If a floppy disk has been inserted in the drive, `volcheck` displays the message:

`/dev/disk has media`
- If the floppy disk drive does not contain a disk, `volcheck` displays the message:

`/dev/disk has no media`

Accessing Removable Media Devices

The following section describe how to access removable media devices.

CD-ROMs and Volume Management

When the volume manager detects the presence of a CD-ROM, it automatically places the CD-ROM under a standard directory in the directory tree, called `/cdrom`.

You have immediate access to the files on the CD-ROM device by moving to the `/cdrom` directory using the `cd` command.

To access files from the local CD-ROM drive:

1. Insert the CD-ROM (label side up) into the drive slot. The CD-ROM is automatically placed under the `/cdrom` directory.

Note – If the CDE File Manager is running, a new File Manager window displays the contents of the CD-ROM. You can access data from this window and the command line interchangeably.

2. In a terminal window, type `cd /cdrom/cdrom0` and press Return.
3. Type `ls` and press the Return key. The list of files in the `/cdrom/cdrom0` directory is displayed.

Floppy Disks and Volume Management

Volume management does not automatically check for the presence of a disk in the drive. You must use `volcheck` to invoke the volume manager to check the drive.

If a disk is present, the volume manager places the disk under a standard directory in the directory tree called `/floppy`. You can then access files on the disk by moving to this directory using the `cd` command.

To access files on a disk in the drive:

1. Insert a formatted disk (label side up) in the drive.
2. Type **volcheck** and press Return. Volume management places the disk under the **/floppy** directory.

Note – If a disk is not in the drive, an error message is not displayed. The **volcheck** command redisplay the prompt.

3. Type **cd /floppy** and press Return.
4. Type **ls** and press Return. The name of the disk is displayed.
5. Type **cd *disk_name*** and press Return.
6. Type **ls** and press Return. The names of the files on the disk are displayed. You can copy files to and from the disk using the **cp** command.

Note – You can access the files on a disk either from the command line or from the File Manager window by selecting Open Floppy from the File Menu.

Ejecting Removable Media Devices

Once you have finished working with the CD-ROM or a floppy disk, you can use the eject command to remove the device from the drive.

Ejecting a CD-ROM

To eject a CD-ROM from the drive:

1. Click on File in the CD-ROM File Manager window and select the Eject option.

or

1. Close the CD-ROM File Manager window.
2. Type **cd** and press Return to change out of the /cdrom directory.
3. Type **eject cdrom** and press Return. After a few seconds the CD-ROM will eject from the drive.

Note – If the specific name of the media device is not known, you can type **eject -q** on the command line to ask the system if a removable media device is on the system.

Ejecting a Floppy Disk

To eject a floppy disk from the disk drive:

1. Click on File in the File Manager disk window, and select the Eject option.

or

1. Close the File Manager disk window.
2. Type **cd** and press Return to change out of the /floppy directory.

3. Type `eject floppy` and press Return. After a few seconds, the disk will eject from the drive or a window is displayed to allow you to manually eject the disk.

Device Busy Message

If the CD-ROM or disk does not eject from the drive and the message “Device busy” appears, you might still be in the working directory on the CD-ROM or disk.

You cannot eject removable devices while you are in the device’s current working directory.

Check to see if you is in the `/cdrom` or `/floppy` directory by typing `pwd`. If either is the current directory for you, type `cd` to return home and then type the `eject` command.

Exercise: Saving and Restoring Files



Exercise objective – In this exercise you practice compressing and uncompressing file, viewing compressed files, and archiving files to tape or disk.

Tasks

Note – If you get a “Permission Denied” error while performing the following exercises, check the write protect switch on the tape.

Complete the following steps and write the commands you would use to perform each task in the space provided.

1. Make a `cpio` copy of the files in your `home` directory in a file called `newfile`. View the contents of `newfile`.

2. In your `home` directory, compress the files `dante` and `file1` using the `compress` command.

3. What are the new names for the compressed versions of these two files?

4. What two commands that can be used to view the contents of a file that was compressed with the `compress` command?

5. Compress the files `file2` and `dante_1` using the `gzip` command.

6. What are the new names for the compressed versions of these two files?

7. Compress the files `file3`, `fruit2`, and `tutor.vi` using the `zip` command.

8. What is the new name for the packaged and compressed version of these three files?

Do the original versions of these three files still exist?

9. Uncompress the files `dante` and `file1`. What command should you use?

Do these files still have a `.Z` extension on the file names?

10. Uncompress the files `file2` and `dante_1`. What command should you use?

Do these files still have a `.gz` extension on the file names?

11. Uncompress the files `file3`, `fruit2`, and `tutor.vi`. What command should you use?

Does the zipped file `file3.zip` still exist in the directory?

12. Archive your `home` directory to a file using the `tar` command.

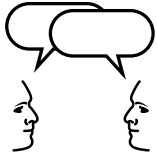
13. Compress the `tar` file using the `compress` command and archive it to tape.

14. Use the `jar` command to archive `~/practice`.

15. Use the `tar` command to archive `~/practice` and compress the file.

16. Compare the `tar` and `jar` file archives of `~/practice` for current size.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

1. Make a `cpio` copy of the files in your home directory in a file called `newfile`. View the contents of `newfile`.

```
$ ls | cpio -oc > newfile  
$ cat newfile | more
```

2. In your home directory compress the files `dante` and `file1` using the `compress` command.

```
$ compress dante file1
```

3. What are the new names for the compressed versions of these two files?

```
dante.Z  
file1.Z
```

4. What two commands can be used to view the contents of a file that was compressed with the `compress` command?

```
$ uncompress -c filename
```

and

```
$ zcat filename
```

5. Compress the files `file2` and `dante_1` using the `gzip` command.

```
$ gzip file2 dante_1
```

6. What are the new names for the compressed versions of these two files?

```
file2.gz  
dante_1.gz
```

7. Compress the files `file3`, `fruit2`, and `tutor.vi` using the `zip` command.

```
$ zip file3 fruit2 tutor.vi
```


8. What is the new name for the packaged and compressed version of these three files?

`file3.zip`

Do the original versions of these three files still exist?

Yes.

9. Uncompress the files `dante` and `file1`. What command should you use?

`$ uncompress dante file1`

Do these files still have a `.Z` extension on the file names?

No.

10. Uncompress the files `file2` and `dante_1`. What command should you use?

`$ gunzip file2 dante_1`

Do these files still have a `.gz` extension on the file names?

No.

11. Uncompress the files `file3`, `fruit2`, and `tutor.vi`. What command should you use?

`$ unzip file3 fruit2 tutor.vi`

Does the zipped file, `file3.zip` still exist in the directory?

Yes.

12. Archive your home directory to a file using the `tar` command.

`$ cd`

`$ cd ..`

`$ tar cvf /tmp/homedir.tar login-ID`

`$ tar tvf /tmp/homedir.tar`

13. Compress the `tar` file using the `compress` command and archive it to tape.

`$ cd /tmp`

`$ compress homedir.tar`

`$ tar cv homedir.tar.Z`

14. Use the `jar` command to archive your `practice` directory.

```
$ cd ~  
$ jar cvf ~/practice.jar practice
```

15. Use the `tar` command to archive your `practice` directory and compress the file.

```
$ cd ~  
$ tar cvf ~/practice.tar practice  
$ compress -v practice.tar
```

16. Compare the `tar` and `jar` file archives of your `practice` directory for current size.

```
$ ls -l *jar *tar.Z
```

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Determine which commands are suitable for storing, viewing, or retrieving different types of files
- ☐ Demonstrate how to reduce the size of files and directories and store them to tape using the `compress` and `tar` commands
- ☐ Describe the steps for uncompressing or viewing a compressed file with the `uncompress` and `zcat` commands
- ☐ Use the `gzip` and `gunzip` commands to compress and uncompress files
- ☐ Use the `zip` command to package and compress multiple files and use the `unzip` command to uncompress a zipped archive file.
- ☐ Compress and copy multiple files to a single archive file in one step using the `jar` command
- ☐ Copy and extract files from an archive file or tape device with the `cpio` command
- ☐ Discuss the benefits of using volume management when working with removable media devices, such as disks and CD-ROMs
- ☐ Demonstrate how to access data on disks and CD-ROMs
- ☐ Use the `eject` command to remove disks and CD-ROMs from device drives

Objectives

Upon completion of this module, you should be able to:

- Open a session on a remote machine using `telnet`
- Log in remotely to another machine on the network
- Use `ftp` to get a file from a remote system

Additional Resources



Additional resources – The following reference can provide additional details on the topics discussed in this module:

- *Solaris User's Guide*, Part Number 802-6499

Example Networking Environment

Figure 9-1 shows the relationship between the network and the host.

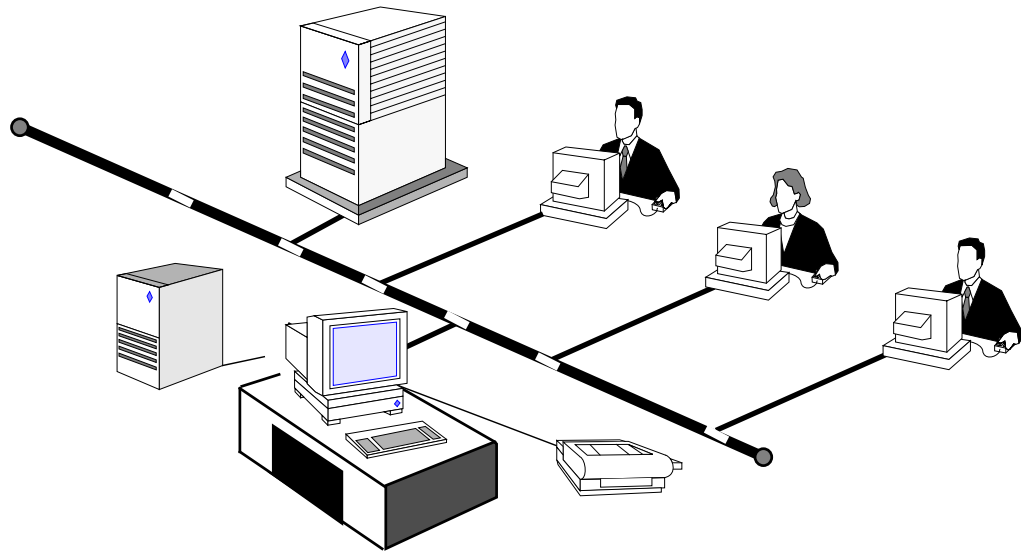


Figure 9-1 Example Networking Environment

Network

A *network* is a connection that enables an exchange of information between machines. Two types of networks are:

- **Local area network (LAN)** — A network that covers a small area, usually less than a few thousand feet or meters
- **Wide area network (WAN)** — A network that can span thousands of miles or meters.

Host

A *host* is a computer system on a network. The *local host* is the machine on which you are currently working. A *remote host* is a system that is being accessed by a user from another system.

Using the telnet Command

The `telnet` command is an application that is part of the Solaris 8 Operating Environment. It uses Transmission Control Protocol/Internet Protocol (TCP/IP) to connect to another system.

The `telnet` connection enables you to log in to a remote system and work in that environment. When using `telnet`, you can

- Open a session on a remote machine
- Access machines that do not run under the UNIX environment

Command Format

```
telnet hostname
```

The following is an example of using `telnet` to connect to a remote system called `host1`:

```
$ telnet host1
Trying host1
Connected to host1
Escape character is '^]'.

SunOS 5.8

login: user2
Password:
Last login: Mon Mar  6 14:13:40 from host1
Sun Microsystems Inc.    SunOS 5.8          Generic February 2000
$
$ uname -a
$ exit
Connection closed by foreign host.
$
```

Using the rlogin Command

Use the `rlogin` command to establish a remote login session on another workstation.

Command Format

`rlogin hostname -option`

Example

To remotely log into another host, execute the following:

```
$ rlogin host2
Password:
Last login: Mon Mar  6 16:22:12 from host1
Sun Microsystems Inc.   SunOS 5.8           Generic February 2000
$ id
uid=11001(user1) gid=10(staff)
$ uname -n
host2
$ pwd
/export/home/user1
$ exit
Connection closed.
$
```


Specifying a Different Login ID

Use the `-l` option to specify a different login ID for the remote login session.

Command Format

```
rlogin -l hostname username
```

Before attempting to remotely log in to another system as a different user, be sure an account for that user exists on the desired remote machine. Check with the system administrator if a user account is required on the remote machine. The information to know includes:

- Machine name
- Login ID
- Password of the new account

Logging in Remotely as Another User

The following is an example on logging in remotely as another user:

```
$ rlogin host1 -l user2
Password:
Last login: Mon Mar  6 16:36:35 from host2
Sun Microsystems Inc.   SunOS 5.8           Generic February 2000
$ id
uid=11002(user2) gid=10(staff)
$ pwd
/export/home/user2
$ uname -n
host2
$ exit
Connection closed.
$
```

Executing a Program on a Remote Machine

The `rsh` command is used to execute a program on a remote system.

Command Format

```
rsh hostname -l username command  
rsh hostname -l username IP_Address command
```

Example

For example, to run commands remotely, execute the following:

```
$rsh host1 showrev
```

```
$rsh host1 -l ssa1 ls /var/mail
```

Copying to and from Another Machine

The `rcp` command enables you to copy files or directories to and from another machine.

Command Format

```
rcp source_file hostname:destination_file
```

```
rcp hostname:source_file destination_file
```

Copying Files Across the Network

- To copy files from a local directory to a remote host, use the following syntax:
\$ rcp dante saturn:/tmp
- To copy files from a remote host to `/tmp`, use the following syntax:
\$ rcp saturn:/tmp/dante /tmp
- To remotely copy directories with the `-r` option, use the following syntax:
\$ rcp -r \$HOME/perm saturn:/tmp

If you are in a directory that contains the file or directory that you want to copy to another machine, type the file or directory name. It is not necessary to type the absolute pathname.



Caution – The `/tmp` directory is used to store files temporarily. Do not use `/tmp` for long-term storage of important files. The files in `/tmp` are cleared out each time the machine is rebooted.

Using the ftp Command

The `ftp` (File Transfer Protocol) command, part of an industry-standard application, is used to transfer files using ASCII or binary mode between systems using similar or dissimilar operating systems.

Once you have successfully used `ftp` to access a remote site, some familiar file and directory access commands, such as `cd` and `ls` are available.

If permissions are set by the site's system administrator for a user to see the contents of a directory, the `ls` command displays files in that directory.

If permissions are set such that you do not have access to the files, when you enter the `ls` command, a prompt is returned in response.

As on your local system, `cd` changes directories on the remote system.

To change directories on your own system in the middle of the `ftp` session, use the `lcd` (local change directory) command.

To end an `ftp` session, type `bye` at the prompt.

Command Format

`ftp hostname`

Examples

The following examples show how to use ftp:

```
$ ftp host1
Connected to host1.
220 host1 FTP server (SunOS 5.8) ready.
Name (host1:user1): user2
331 Password required for user2.
Password:
230 User user2 logged in.
ftp> ls
200 PORT command successful.
150 ASCII data connection for /bin/ls (192.9.200.1,32970) (0 bytes).
dante
dante_1
dat
dir1
dir2
dir3
dir4
file1
file2
file3
file4
fruit
fruit2
practice
tst
tutor.vi
226 ASCII Transfer complete.
113 bytes received in 0.0033 seconds (33.84 Kbytes/s)
ftp> cd dir1
250 CWD command successful.
ftp> ls
200 PORT command successful.
150 ASCII data connection for /bin/ls (192.9.200.1,32971) (0 bytes).
coffees
constellation
feathers
feather_6
fruit
planets
trees
226 ASCII Transfer complete.
23 bytes received in 0.0021 seconds (10.83 Kbytes/s)
```

```
ftp> cd coffees
250 CWD command successful.
ftp> ls
200 PORT command successful.
150 ASCII data connection for /bin/ls (192.9.200.1,32973) (0 bytes).
beans
226 ASCII Transfer complete.
27 bytes received in 0.0022 seconds (12.06 Kbytes/s)
ftp> bin
ftp> get beans
200 PORT command successful.
150 ASCII data connection for beans (192.9.200.1,32974) (0 bytes).
226 ASCII Transfer complete.
ftp> lcd
Local directory now /export/home/user1
ftp> bye
221 Goodbye.
$
```

Exercise: Performing Network Basics



Exercise objective – In this exercise you use some of the networking commands introduced in this module.

Tasks

Complete the following steps:

1. Use the `rlogin` command to log in to another machine in your classroom.

In what directory are you placed on the remote machine?

2. Issue the command that shows you the host name of the current machine.
-

3. Log out of the remote machine. Display the host name of your current machine to determine if you are back to your own host.
-

4. Use the `rlogin` command and option to log in to another machine as the user `guest` with a password of `guest` (or as another user as specified by your instructor).
-

5. Log out of the remote machine.
-

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

Complete the following steps:

1. Use the `rlogin` command to log in to another machine in your classroom.

```
$ rlogin hostname
```

In what directory are you placed on the remote machine?

A home directory on the remote machine (either `/home/username` or `/export/home/username`) or the root (`/`) directory if no home directory exists.

2. Issue the command that shows you the host name of the current machine.

```
$ uname -n
```

3. Log out of the remote machine. Display the host name of your current machine to determine if you are back to your own host.

```
$ exit  
$ uname -n
```

4. Use the `rlogin` command and option to log in to another machine as the user `guest` with a password of `guest`.

```
$ rlogin hostname -l guest
```

5. Logout of the remote machine.

```
$ exit
```

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Open a session on a remote machine using `telnet`
- ☐ Log in remotely to another machine on the network
- ☐ Use `ftp` to get a file from a remote system

Objectives

Upon completion of this module, you should be able to:

- Describe how processes are created
- View active processes on the system using the `ps` command
- Find a specific process using the `pgrep` command
- Discuss the purpose of signals for controlling process activity
- Terminate a process using the `kill` and `pkill` commands
- Use job control commands to manage jobs running in the shell

Additional Resources



Additional resources – The following reference can provide additional details on the topics discussed in this module:

- *Solaris User's Guide*, Part Number 802-6499

Process Overview

Every program you run in the Solaris Operating Environment creates a *process*. When you log in and start the shell, it is a process. When you execute a command or you open an application, it starts a process.

A process is any program running on the system.

The system also starts processes called daemons. Daemons are programs started at boot time and are critical to the functionality of the operating system.

Every process is assigned a unique process identification number (PID), which is used by the kernel to track and manage the process. For a user, the PID number is used to identify and control the process.

Process UID and GID

For the kernel to know what a process is allowed to do, it must store information about who owns the process. The kernel stores two types of UIDs and GIDs for this purpose.

The real UID and GID of a process are the same as the UID and GID of the user who started the process. These are used for accounting (tracking) purposes.

Parent Process

When a process is started, a duplicate of that process is created. This new process is called the child, and the process that created it is called the parent. The child process then changes the code used by the parent process to that of the command you had requested to be executed.

While the command is executing the shell blocks until the child process has completed. Once complete, the waiting parent process terminate the child process, and the shell displays the prompt and waits for a new command.

Viewing Processes and PIDs

The `ps` (process status) command lists the processes currently running on the system.

For each process, `ps` displays the process ID (PID), terminal identifier (TTY), cumulative execution time (TIME), and the command (CMD).

Command Format

`ps -options`

Options

You can use the following options with the `ps` command:

- e Prints information about every process on the system; includes PID, TTY, TIME, and CMD.
- f Generates a full listing, which adds the fields UID (owner of process), PPID (parent process ID), and STIME (process start time).

Displaying a Full Listing of all Processes

The following example displays a listing of all processes.

```
$ ps -ef | more
UID      PID    PPID    C   STIME          TTY          TIME      CMD
root      0        0      0   16:46:41       ?            0:01      sched
root      1        0      0   16:46:44       ?            0:40      /etc/init -
root      2        0      0   16:46:44       ?            0:00      pageout
root      3        0      0   16:46:44       ?            4:33      fsflush
root     236      1      0   16:48:08       ?            0:01      /usr/lib/saf/sac
root     844      1      0   12:12:10       ?            0:00      /usr/lib/lpsched
aster   1292      1      0   06:48:51      console     0:01      -ksh
root     241     236      0   16:48:14       ?            0:01
/usr/lib/saf/ttymon
rose    1400    321      0   20:03:11       ?            0:00
/usr/openwin/bin/clock
--More--
```

Table 10-1 contains descriptions for the column headings.

Table 10-1 Column Headings for `ps -ef` Output

Value	Description
UID	The login ID name of the owner of the process.
PID	The unique process identification number of the process.
PPID	The parent process identification number for the process.
C	CPU utilization for scheduling - OBSOLETE.
STIME	Time the process started (hh:mm:ss).
TTY	The controlling terminal where the process started ^a .
TIME	Cumulative execution time for the process.
CMD	The command name.

^a The controlling terminal for system daemons appears as a question mark (?).

Searching for a Specific Process

To quickly locate a specific process, you can pipe the output of the `ps` command to `grep` for a specific command name. For example, to find all active processes related to printing, execute the following:

```
$ ps -e | grep lp
217  ?  0:0  lpsched
$
```

Note – `lpsched` is a daemon responsible for controlling print services on the system.

The pgrep Command

The `pgrep` command gives you an even more efficient method for quickly finding a specific process by name.

Note – The `pgrep` command was released with the Solaris 8 Operating Environment.

By default, `pgrep` displays the PID of any process that matches the specified criteria on the command line. For example:

```
$ pgrep lp
217
$
```

Command Format

`pgrep -options pattern`

Options

You can use the following options with the `pgrep` command.

-x	Displays only those PIDs that exactly match <i>pattern</i> .
-n	Displays only the newest (most recently created) PID that matches <i>pattern</i> . (Displays every process ID containing <i>pattern</i> .)
-U	Displays only those PIDs that belong to the specified owner. (Use either the login ID name or a UID number.)
-l	Displays the name of the process along with its PID.

Note – The `pgrep` command has several other useful options. View the man page for this command for a list of all the available options.

In the examples below, using the `-l` option displays both the names of the process and their PIDs.

```
$ pgrep -l lp
      217      lpsched
$
```

```
$ pgrep -l mail
      215      sendmail
     12047     dtmail
$
```

In the next example, combining the `-l` and `-x` options displays only those processes that match the exact names.

```
$ pgrep -lx dtmail
     12047     dtmail
$
```


Sending Signals to Processes

Signals

Signals are used to control processes running on the system. Signals are sent to processes to indicate that an event has occurred and the process must respond.

For example, if you type **Control+c** to terminate a command, this sends an interrupt signal to the process and the processes responds by exiting.

A signal is a simple message containing a signal number as information to a process.

There are a number of signals available in the Solaris Operating Environment. Each signal is associated with a unique number, a name, and an action. You can choose to associate an action with a signal.

You can find a complete list of these signals and their default actions on the following man page.

```
$ man -s3head signal
```

Table 10-2 describes some signals and names.

Table 10-2 Signal Number and Names

Signal Number	Signal Name	Action	Response
1	SIGHUP	Hangup	Exit
2	SIGINT	Interrupt	Exit
9	SIGKILL	Kill	Exit
15	SIGTERM	Terminate	Exit

- 1, SIGHUP A hangup signal causes a phone line or terminal connection to be dropped.
- 2, SIGINT — An interrupt signal generated from your keyboard — usually by **Control-c**.

- 9, SIGKILL — A signal used to kill a process. You cannot ignore this signal.
- 15, SIGTERM — A signal to terminate a process in an orderly fashion. Some processes can ignore this signal.

Terminating Processes

This following section will illustrate how to execute the `kill` Command.

The kill Command

The `kill` command is used to send a signal to one or more running processes. This command is often used to terminate a process.

Note – Ordinary users can kill only processes that they own. The `root` user can kill any process.

Command Format

```
kill -signal process-ID
```

Terminating a Process

Before you can terminate a process, you must know its PID. You can use either the `ps` or `pgrep` command to locate the PID for the process.

To terminate a process, issue the command:

```
$ kill PID
```

For example:

```
$ pgrep -l mail
215 sendmail
12047 dtmail
$
```

```
$ kill 12047
$
```

To terminate more than one process at the same time:

```
$ kill PID PID PID PID
```

Using the `kill` command without specifying a signal on the command line sends the default Signal 15 to the process. This signal usually causes the process to terminate.

Some processes ignore a Signal 15. These can be processes waiting for a resource, for example a process that is waiting for a tape drive to complete an operation so it can continue.

Those processes that do not respond to a Signal 15 can be terminated by force using Signal 9 with the `kill` command. For example:

```
$ kill -9 PID
```



Warning – Use the `kill -9` command as a last resort to terminate a process. Using `kill -9` on a process that controls a database application or a program that updates files can be disastrous! The process is terminated instantly with no opportunity to perform an orderly shutdown.

The pkill Command

The `pkill` command is also used to terminate a process, by default using Signal 15. This command can terminate a process using its process name.

Command Format

```
pkill [-options] pattern
```

The options used with `pkill` are the same used by the `pgrep` command. For example:

To terminate the desktop mail process by its PID, execute the following:

```
$ pgrep -l mail
215 sendmail
470 dtmail
$ pkill 470
```

To terminate the mail session process by its name, execute the following:

```
$ pkill dtmail
$
```

Use the `-n` option to kill the most recently created Korn shell process. For example:

```
$ pkill -n ksh
$
```

Killing Processes Remotely

When a workstation is not responding to your keyboard or mouse input, the session must be frozen. In such cases, you can remotely access your workstation using `rlogin` (or `telnet`) from another system.

Once connected, you can invoke the `pkill` command to terminate the corrupted session on your workstation. For example

```
$ rlogin host1
Password:
Last login: Fri Feb 04 16:50:30 from host1
Sun Microsystems Inc. SunOS 5.8 Generic February 2000

$ pkill -9 loginshell
```

Managing Jobs

The shell gives you the ability to execute a variety of jobs simultaneously. Opening an application, sending a print request, or executing an `ls` command on a directory are all examples of jobs.

When a job is executed in the window environment, it runs in the foreground and ties up that particular window until the job is done.

However, you can execute jobs by the shell in the background, freeing up the window so you can start another job in the foreground.

Every job is assigned a *job ID* by the shell. The handling of multiple jobs within the shell is called *job control*. A job is a process controlled by a terminal and containing a process ID.

You can control jobs by their job ID numbers using the commands in Table 10-3.

Table 10-3 Job Control Commands

Command	Value
<code>jobs</code>	Displays which jobs are currently running.
<code>bg %n</code>	Places a job in the background (<i>n</i> = job ID)
<code>fg %n</code>	Places a job in the foreground (<i>n</i> = job ID)
<code>^Z</code>	Suspends a job
<code>stop %n</code>	Stops specified background job (<i>n</i> = job ID)

Note – You can control a job only by using these commands in the window where the job started.

To have a job run in the background, you type the command to be executed, followed by an ampersand (&) symbol.

For example, this command line executes the `find` command in the background. It locates all files named `core` in the current working directory. It then prints the full pathname of each `core` file into a new file called `list`.

```
$ find . -name core > list &  
[1]      3028  
$
```

The shell returns a job ID number contained in brackets and a PID number for the command. The job ID number gives you the ability to control the actual job. The PID number is used by the kernel to manage the job.

When you press Return in the window, a message is displayed indicating that the background job has completed.

```
[1] +  Done      find . -name core > list &  
$
```

- Use the `jobs` command to list your current jobs.

```
$ jobs  
[1] + Running find . -name core > list &
```

- Use the `fg` command to bring a background job to the foreground.

```
$ fg %1  
find . -name core > list
```

Note – This occupies the window until the job is completed or placed back into the background (or stopped).

- To return this job to the background, suspend it first using the Control+Z keys, then use the **bg** command.

```
find . -name core > list
^Z
[1] + Stopped(SIGTSTP) find . -name core > list &

$ jobs
[1] + Stopped(SIGTSTP) find . -name core > list &

$ bg %1
[1] find . -name core > list &
$
```

Note – Placing a stopped job into either the foreground or the background restarts the job.

- To stop a background job, use the specific job number as the argument to the **stop** command.

```
$ stop %1
[1] + Stopped (SIGSTOP)          script1 &
$
```


Exercise: Manipulating System Processes



Exercise objective – In this exercise, you use the commands learned in this module to determine PID numbers, kill processes, and control jobs.

Tasks

Complete the following steps and write the commands you would use to perform each task in the space provided.

1. Issue the following command in the background:

\$ sleep 500 &

2. Using the `jobs` command, find the job number of the `sleep` command in step 1.

Job Number: _____

3. Bring the job to the foreground, then put it back in the background.

4. Kill the job running the `sleep` command.

5. Use the following `ps` commands to list the processes currently running on your system. What information does each command provide?

\$ ps

\$ ps -f

```
$ ps -e
```

```
$ ps -ef
```

6. In a terminal window, run the `ps -ef` command. Identify the PID of this command.

```
$ ps -ef
```

PID Number: _____

7. In a terminal window, issue the following command:

```
$ cat -v /dev/zero
```

Note – This command is being used to produce a continuously running command for demonstration purposes only. For information on the `/dev/zero` file, see `man zero`.

8. Open another terminal window and use the `ps -ef` command to identify the PID of the `cat` command.

PID Number: _____

9. From this terminal window, kill the `cat` command using the PID number.
-

10. From the same window, enter the `tty` command to identify the name of this terminal window. The name displays as `/dev/pts/#`, where `#` is an actual number. For example: `/dev/pts/4`.

```
$ tty
```

`/dev/pts/`_____

11. Move back to the original terminal window, and use `pgrep` to find the PID associated with the name of the second terminal window.

\$ pgrep -t pts/#

PID Number: _____

12. In the current window, use the `kill` command to try and kill the second terminal window.

\$ kill PID#

Did it work? _____

13. Use the `kill` command with the `-9` option to try and kill the second terminal window.

\$ kill -9 PID#

Did it work? _____

14. Start a Korn shell in the remaining window.

\$ ksh

15. Run the following `kill` commands to identify the signals it sends when you use the specified options.

\$ kill -l 9 *(-l is the letter l)*

Signal: _____

\$ kill -l 15 *(-l is the letter l)*

Signal: _____

16. Exit the Korn shell.

\$ exit

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

Complete the following steps and write the commands you would use to perform each task in the space provided.

1. Issue the following command in the background:

```
$ sleep 500 &
```

2. Using the `jobs` command, find the job number of the `sleep` command started in step 1.
3. Bring the job to the foreground and then put it back in the background.

```
$ fg %1
```

```
^Z
```

```
$ bg %1
```

4. Kill the job running the `sleep` command.

```
$ kill %1
```

5. Use the following `ps` commands to list the processes currently running on your system. What information does each command provide?

```
$ ps
```

Prints information for the current user and terminal.

```
$ ps -f
```

Prints a full listing of the above.

```
$ ps -e
```

Prints information about every process running.

```
$ ps -ef
```

Prints a full listing of the above.

6. In a terminal window, run the `ps -ef` command. Identify the PID of this command.

PID Number: This number will differ from system to system.

7. In a separate terminal window, issue the following command:

```
$ cat -v /dev/zero
```

8. Open another terminal window and use the `ps` command to identify the process ID of the `cat` command.

```
$ ps -ef | grep cat
```

9. From this terminal window, kill the `cat` command using the PID number.

```
$ kill PID
```

where PID is the PID of the `cat` command

10. From the same window, enter the `tty` command to identify the name of this terminal window. The name will display as `/dev/pts/#`, where # is an actual number. For example: `/dev/pts/4`.

```
$ tty
```

`/dev/pts/#` — This name will differ from system to system.

11. Move back to the original terminal window, and use `pgrep` to find the PID associated with the name of the second terminal window.

```
$ pgrep -t pts/#
```

`PID#` — This name will differ from system to system.

12. In the current window, use the `kill` command to try and kill the second terminal window.

```
$ kill PID#
```

Did it work?

No.

13. Use the `kill` command with the `-9` option to try and kill the second terminal window.

```
$ kill -9 PID#
```

Did it work?

Yes.

14. Start a Korn shell in the remaining window.

```
$ ksh
```

15. Run the following `kill` commands to identify the signals it sends when you use the specified options.

```
$ kill -l 9
```

Signal *KILL*

```
$ kill -l 15
```

Signal *TERM*

16. Exit the Korn shell.

```
$ exit
```

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Describe how processes are created
- ☐ View active processes on the system using the `ps` command
- ☐ Find a specific process using the `pgrep` command
- ☐ Discuss the purpose of signals for controlling process activity
- ☐ Terminate a process using the `kill` and `pkill` commands
- ☐ Use job control commands to manage jobs running in the shell

Objectives

Upon completion of this module, you should be able to:

- Describe the functions of the Korn shell as a command interpreter
- Demonstrate the use of quoting to mask the special meaning of metacharacters by the Korn shell
- Define the terms standard input, standard output, and standard error
- Use metacharacters to redirect standard input, standard output and standard error
- Connect two or more commands together using the pipe feature
- Implement the Korn shell file name completion mechanism
- Describe the two types of variables used by the Korn shell
- Use commands to view, set, and unset shell variables
- Invoke the `history` mechanism to repeat or edit previously executed commands
- Use the `alias` utility to customize and abbreviate UNIX commands
- Create Korn shell functions to group UNIX commands into separate routines
- Define the Korn shell initialization files used to customize a user's environment

The Shell as a Command Interpreter

In any UNIX environment, the shell is commonly referred to as a *command interpreter*. It allows you to interact with the kernel by interpreting commands that are typed either on the command line or placed within a script file.

The shell as a command interpreter accepts, analyzes, and processes the input from users or script files and also generates appropriate error messages.

It is interchangeable, so users can chose which command interpreter to use at any time—changing easily between the Bourne shell, the C shell, and the Korn shell.

Although the shell is ultimately responsible for ensuring that any commands typed at the command line prompt get properly executed, it has other important functions too.

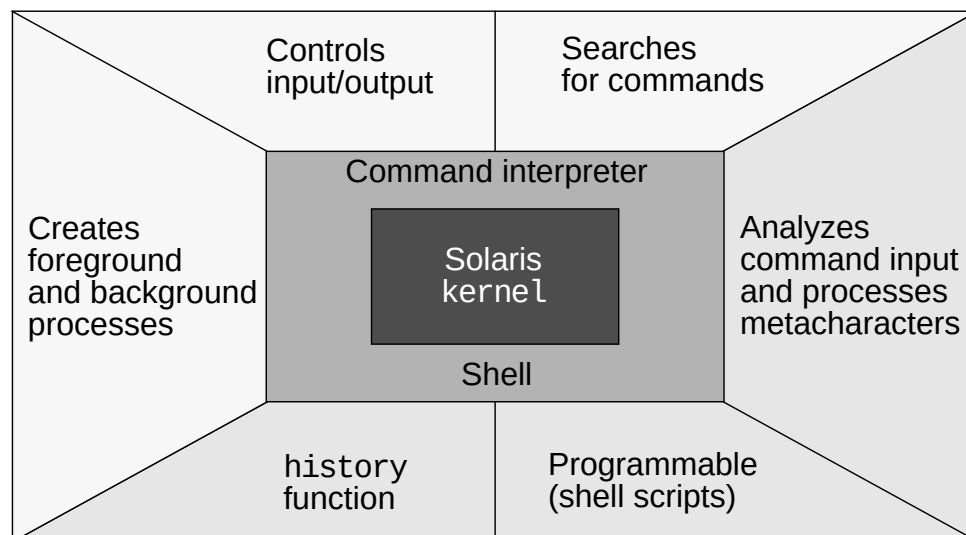


Figure 11-1 Tasks Accomplished by the Shell

Responsibilities of the Shell as a Command Interpreter

The following section describes the responsibilities of the shell:

- The shell searches for commands in all the directory locations defined in the shell variable `PATH`, a colon-separated list of directories searched by the shell from left to right.
- The shell sets up pipes, input/output redirection, and background processing.
- The shell can be custom-tailored by each individual user by creating aliases (abbreviated names) for a command or series of commands or by creating functions when an alias is not sufficient.
- The shell can be adapted to different terminal environments by setting the shell variable `TERM`.
- The shell saves typing with the command-line completion mechanism.
- The shell stores previously executed commands in a history list that can be re-run and edited.
- The shell can be customized for each individual user on the system through the use of shell initialization files.

Input and Output Redirection and Piping

The following section discusses the redirection and piping of input and output.

Redirecting Data I/O

The shell typically receives, or reads, command input from the keyboard and displays, or writes, command output to the terminal screen.

You, however, can instruct the shell to redirect command input and command output by using the redirect symbols `<` and `>`.

I/O redirection is commonly used for redirecting input and output to files on the command line or within shell scripts.

Input redirection is the ability to force a command to read any necessary input from a file instead of from the keyboard.

Output redirection is the ability to send the output from a command into a file or to another command (using a pipe) instead of sending the output to the terminal screen.

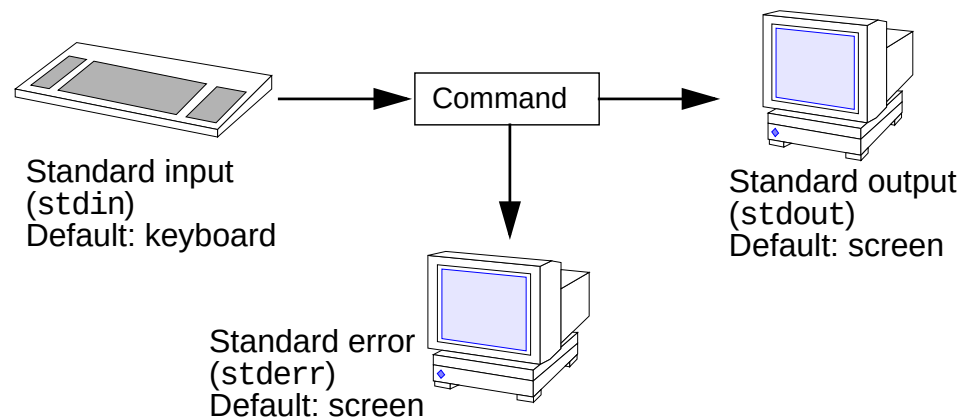


Figure 11-2 Standard Command I/O in the Shell

File Descriptors

Each process created by the shell is associated with the file descriptors (FD) listed in Table 4-1

Table 11-1 File Descriptors

File Descriptor	Meaning
0 <code>stdin</code>	Standard input
1 <code>stdout</code>	Standard output
2 <code>stderr</code>	Standard error

These file descriptors are used by the shell to determine where input to the command comes from and where the output and error messages sent.

Standard Command Input and Output

The following covers the standard commands:

- Standard input (`stdin`) is always File Descriptor 0 (zero).
- Standard output (`stdout`) is always File Descriptor 1.
- Standard error (`stderr`) is always File Descriptor 2.

All commands processing file content are implemented to read from standard input and write to standard output. This is demonstrated by invoking the `cat` command without arguments and typing the two lines of text. To exit, press the `Control+d` keys.

```
$ cat
First line
What's going on?
^d
$
```

The `cat` command is taking its standard input from the keyboard and displaying the standard output to the terminal window.

```
$ cat
First line (read from stdin)
First line (write to stdout)
What's going on? (read from stdin)
What's going on? (write to stdout)
^d
$
```

Redirecting stdin, stdout, and stderr

You can change the default behavior of standard input, standard output, and standard error within the shell.

Redirecting stdin

The following explains what a `stdin` command will do:

- `command < filename` or `command 0< filename`

The command reads its input from the file called *filename* instead of reading input from the keyboard. For example:

```
$ mailx user1 < ~/dante
```

Redirecting stdout

The following example shows the redirecting of `stdout`:

- `command > filename` or `command 1> filename`

The command output is directed to a file called *filename*. If *filename* does not exist, it is created. If *filename* exists, the redirection overwrites the contents of the file. For example:

```
$ ps -ef > process_list
```

Redirecting stdout Using Append Mode

The following example shows the redirecting of stdout using Append:

- `command >> filename`

The command output is directed to a file called *filename* and appended to the end of the existing file content. If *filename* does not exist, it is created. For example:

```
$ cat /etc/passwd > my_file;cat my_file
$ echo "That's my passwd file" >> my_file;cat my_file
```

Redirecting stderr

The following example shows the redirecting of stderr:

- `command 2> /dev/null`

Any command error messages are redirected to the file `/dev/null`. This is useful to suppress error messages considered insignificant, so that no error messages display on the terminal screen. For example:

```
$ find /etc -type f -exec grep PASSREQ {} \; -print 2> /dev/null
# PASSREQ determines if login requires a password.
PASSREQ=YES
/etc/default/login
$ ^C
```

Redirecting stdout and stderr

The following example show the redirecting of stdout and stderr:

- `command 1> filename 2>&1`

The syntax `2>&1` instructs the shell to redirect stderr (2) to the same file that stdout (1) is directed to write its results. For example:

```
$ ls /var /no 1> dat 2>&1
$ cat dat | more
/no: No such file or directory
/var:
adm
audit
cron
<OUTPUT OMITTED>
```

The Pipe Feature

The shell gives you the ability to effectively connect two commands together. This connection is known as a pipe (`|`). Building a pipe on the command line takes the output from one command and feeds it directly into the input of another command.

A pipe is indicated by the character `|` and is placed between two commands.

Command Format

```
command | command
```

Some Basic Examples Using a Pipe

You can make a pipe can be made between any two commands, provided the first command writes its output to standard output, and the second command reads its input from standard input.

For example:

```
$ who | wc -l
5
$
```

The standard output of the first command is connected directly to the standard input of the second command. The output of the `who` command never appears on the terminal screen, because it is *piped* directly into the `wc` command.

To obtain the total number of active processes on the system:

```
$ ps -ef | wc -l
62
$
```


To get a listing of all the subdirectories located in /etc:

```
$ ls -F /etc | grep "/"
acct/
cron.d/
default/
...
```

Building a Pipeline

You can create a pipeline that consists of more than two commands. You can connect an unlimited number of commands by pipes. For example:

```
$ head -10 dante | tail -3 | lp
request id is printerA-177 (standard input)
$
```

Options of the Korn Shell

Options are switches that control the behavior of the Korn shell. They are boolean, in that each can be either ON or OFF.

To switch an option ON, type:

```
$ set -o option_name
```

To switch an option OFF, type:

```
$ set +o option_name
```

Protecting File Content During I/O Redirection

Redirecting standard output into an existing file overwrites the previous file content which results in data loss. To prevent this from occurring, the shell supports a `noclobber` option.

When the `noclobber` option is set, the shell refuses to redirect standard output into the existing file and displays an error message to the screen.

The `noclobber` option is activated in the shell with the `set` command. For example:

```
$ set -o noclobber
$ set -o | grep noclobber
noclobber          on
$ ps -ef > file_new
$ cat /etc/passwd > file_new
ksh: new-file: file already exists
$
```

Deactivating the noclobber Feature

To temporarily deactivate the noclobber option use the syntax `>|` on the command line. The noclobber option is ignored for this command line only, and the contents of the file are overwritten.

```
$ ls -l >| file_new
```

- To turn off the noclobber feature permanently, execute the following:

```
$ set +o noclobber
$ set -o | grep noclobber
noclobber      off
$ ls -l > file_new
$
```

File Name Completion in the Korn Shell

File name completion, often referred to as file name expansion, is a mechanism that allows you to type the first few characters of a file name, then, by pressing a specific sequence of keys, instruct the Korn shell to complete the remainder of the file name.

The key sequence used for file name completion in the Korn shell Escape (Esc) and backslash (\).

Using File Name Completion

To activate the file name completion mechanism, you must set the **EDITOR** variable to `/usr/dt/bin/vi`. This mechanism also requires that the user switch on the **vi** option within the Korn shell, to start **vi** in insert mode.

```
$ set -o vi
```

To invoke file name completion, type the `ls` command followed by one or more characters of a file name, then press the following keys in sequential order: Escape (Esc) and backslash (\).

Note – If you are using an editor other than **vi**, for example **emacs**, the escape sequence is different.

If the shell finds a file name beginning with the specified characters, it prints the complete file name or file names to the command line.

For example, to expand a file name beginning with the characters “de” in the `/usr` directory:

```
$ cd /usr
$ ls de [Press Escape (Esc) and backslash (\)]
```

The shell completes the remainder of the file name, displaying:

```
$ ls demo/
```

You can request the shell present all possible alternatives of a partial file name from which you could then select. This action is invoked by pressing the following keys in sequential order: Escape (Esc) and the equal (=) key.

To request that the shell present all file names beginning with the letter “g” in the /etc directory, type:

```
$ cd /etc
$ cat g (Press Escape (Esc), press the equal (=) key)
1) getty
2) group
3) grpck
4) gss/
$ cat g
```

The cursor is positioned on top of the letter “g” at this point.

Variables of the Korn Shell

A variable is a name that refers to a temporary storage area in memory. Variables contain information used for customizing the shell and information required by other processes so they function properly. The shell allows you to store values into variables.

Shell programming uses two types of variables: local variables and environment variables. By convention, local variables are in lowercase characters and environment variables are in uppercase.

- Local variables — Are private to the shell they were created in and are not passed on to any processes started in that shell.
- Environment variables — Are passed from parent process to child process, and so on. Some variables are inherited by the login shell, other variables are created in user's shell initialization files, in shell scripts, or at the command line.

The following table summarizes the Korn shell commands to set, unset or view both variable types:

Table 11-2 Korn Shell Commands for Variable Types

Action	Environment Variables	Local Variables
To set a variable	<code>export VAR=value</code>	<code>var=value</code>
To unset a variable	<code>unset VAR</code>	<code>unset var</code>
To display all variables	<code>env</code> , <code>export</code> (lists only environment variables)	<code>set</code> (lists all local and environment variables)
To display the values stored in variables	<code>echo \$VAR</code>	<code>print \$var</code>

If a valid shell variable name follows the \$ sign, then the shell takes this as an indication that the value stored inside that variable is to be substituted at that point.

Referencing Values in Variables

The `echo` command is used to display the value that is stored inside a shell variable. For example:

```
echo $SHELL
/bin/ksh
```

Display of Variables

The `set` command lists all shell variables (local and environment) with their current values. For example:

```
$ set
DISPLAY=:0.0
EDITOR=/usr/bin/vi
ERRNO=13
FCEDIT=/bin/vi
HELPPATH=/usr/openwin/lib/locale:/usr/openwin/lib/help
HOME=/home/user1
HZ=100
IFS=
LANG=C
LINENO=1
LOGNAME=user1
MAIL=/var/mail/user1
MAILCHECK=600
MANPATH=/usr/man:/usr/openwin/share/man
OLDPWD=/home/user1
OPENWINHOME=/usr/openwin
PATH=/usr/openwin/bin:/bin:/usr/bin:/usr/ucb:/usr/sbin
PPID=596
PS1='$ '
PS2='> '
PS3='#? '
PS4='+ '
PWD=/tmp
SHELL=/bin/ksh
TERM=sun-cmd
TERMCAP='sun-cmd:te=\E[>4h:ti=\E[>4l:tc=sun:'
TMOUT=0
TZ=MET
USER=user1
_=set
office=/home/user1/office
private=/home/user1/private
```

To make the value of an environment variable known to a subshell, it must be exported with the **export** command. You can view the list of all these environment variables and their current values with the **export** command.

```
$ export
DISPLAY=:0.0
EDITOR=/usr/bin/vi
HELPPATH=/usr/openwin/lib/locale:/usr/openwin/lib/help
HOME=/home/user1
LANG=C
LOGNAME=user1
MAIL=/var/mail/user1
MANPATH=/usr/openwin/share/man:/usr/man
OPENWINHOME=/usr/openwin
PATH=/usr/openwin/bin:/bin:/usr/bin:/usr/ucb:/usr/sbin
PWD=/etc
SHELL=/bin/ksh
TERM=sun-cmd
TERMCAP='sun-cmd:te=\E[>4h:ti=\E[>4l:tc=sun:'
TZ=MET
USER=user1
__=export
office=/home/user1/office
$
```

Setting Shell Variables

Local Variables

A local variable is set and a value is assigned with the following syntax:

```
var=value
```

There must be no space on either side of the equal (=) sign. For example:

```
$ private=/home/user1/private
$ set | grep private
private=/home/user1/private
$ cd $private; pwd
/home/user1/private
$
```


Environment Variables

An environment variable is set and a value is assigned using the following syntax

```
export VAR=value
```

For example:

```
$ export office=/home/user1/office
$ echo $office
/home/user1/office
$
```

Unsetting Shell Variables

The values stored in both environment and local variables can be deleted with the following syntax:

```
unset VAR
unset var
```

For example, to unset a variable:

```
$ unset private
$ echo $private
$
```

Variables Set by the Shell on Login

Table 11-3 shows a basic list of variables that are assigned default values by the shell on login.

Table 11-3 Variables Set by the Shell on Login

Variable	Meaning
EDITOR	Defines the default editor for the shell.
FCEDIT	Defines the editor for the <code>fc</code> command. Used with the history mechanism for editing previously executed commands.

Table 11-3 Variables Set by the Shell on Login

Variable	Meaning
HOME	The directory that cd changes to when no argument is supplied on the command line
LOGNAME	Sets the login name of the user
PATH	A colon delimited list of directories to be searched when the shell needs to find a command to be executed
PS1	The primary Korn shell prompt: \$
PS2	The secondary command prompt, normally: >
SHELL	The name of the shell (for example: /bin/ksh)

Customizing Korn Shell Variables

The following describes how to customize korn shell variables.

The Prompt Variable PS1

The shell prompt string is stored in the shell variable **PS1** and you can customize it according to your preferences.

```
$ PS1="$LOGNAME@`hostname`$ "
username@host name
$
```

In this example, the prompt displays the login name of the user and the system's hostname.

The user name is read from the variable **LOGNAME**, and the host name comes from the output of the **hostname** command.

This prompt displays the correct information even when the user logs on to different hosts.

The second example shows how the value of another variable is used for prompt definition.

```
$ ILU="I Like UNIX"
$ PS1="$ILU > "
I Like UNIX > echo $ILU
I Like UNIX
I Like UNIX >
```

Note – To have the new shell prompt appear in every shell invoked, it must be included in the user's Korn shell initialization file, usually named **.kshrc**.

The PATH Variable

The PATH variable contains a list of directory path names, separated by colons. When you invoke a command from the command line, the shell searches these directories from left to right to locate the command to be executed.

The first command found is the one executed by the shell.

If the shell does not find the command in any of the listed directories, it displays the following error message:

```
ksh: command_name: not found
```

The command may, however reside in a directory not specified in the PATH variable. Since the shell restricts its search to only those directories specified in the PATH.

When the shell is unable to execute a command because it is not found, the user can type the command on the command line using the absolute path. For example:

```
$ /usr/bin/id
```

If the command is successfully executed by the shell, the user should then check the PATH variable to ensure the directory exists in the search list and, if so, it has been entered correctly. For example:

```
$ echo $PATH
/usr/openwin/bin:/bin:/usr/ucb:/usr/sbin:/usr/bin
$
```

Extending the PATH Variable

In the following example, the `PATH` variable is quickly extended to include the `home` directory of the user.

```
$ print $PATH
/usr/openwin/bin:/bin:/usr/bin:/usr/ucb:
$
$ PATH=$PATH:~
$ echo $PATH
/usr/openwin/bin:/usr/bin:/bin:/usr/ucb:/export/home/user1
$
```

The `PATH` variable is, by default, an environment variable and automatically passes the value to subshells.

Metacharacters of the Korn Shell

Metacharacters, which have special meaning to the shell, are a powerful feature of any shell. However, there are times when you need to instruct the shell to mask, or ignore, the special meaning of these metacharacters. This is accomplished through the process known as *quoting*.

The special characters used by the shell for quoting include the backslash (\), single quote ('), and the double quote (").

Quoting with the Backslash

Placing a backslash in front of a metacharacter prevents the shell from interpreting it as a metacharacter.

For example, a file was created with an asterisk (*) as its name. To delete this file, the asterisk (*) must be masked by a backslash (\) character.

```
$ ls
dat1  dat2  dat3  dir1  dir2  *
$ rm \*
```

Quoting with Single Quotes and Double Quotes

Placing special characters between quote characters prevents the shell from interpreting them as metacharacters. Unlike the backslash, which masks only one special character, quotes are used for masking more than one metacharacter at a time.

- Single quotes (') — Used to tell the shell to ignore all enclosed metacharacters.
- Double quotes (") — Used to tell the shell to ignore most of the enclosed metacharacters.

The following three metacharacters retain their special meaning to the shell whenever they are placed inside double quotes.

- Backslashes (\)

- Dollar signs (\$)
- Back quotes (' ')

Basic Examples of Quoting:

```
$ echo '$SHELL'
$SHELL
$ echo "$SHELL"
/bin/ksh
$ echo "\$SHELL"
$SHELL
$ echo '\ '
\
$ echo "\"
> ^C
```

Command Substitution

Any UNIX command that is placed between back quotes is executed by the shell and its output displayed. For example:

```
$ echo `date`
date
$ echo `date`
Tue May 2 14:10:05 MDT 2000
$
```

Using back quotes for command substitution is common among all the UNIX shells, however the Korn shell provides an alternative method for command substitution.

```
$ echo `pwd`
pwd
$ echo $(pwd)
/home/user1
```

The History Mechanism

The Korn shell keeps a history of recently entered commands. This history mechanism enables users to view, repeat, edit, or modify their previously executed commands.

The history command

The Korn shell stores history commands in a file specified by the HISTFILE variable. The default file is \$HOME/.sh_history.

You can specify the number of commands to be stored using the HISTSIZE variable. If the variable is not set, the most recent 128 commands are stored.

You can edit previously executed commands and re-run them using a Korn shell in-line editor, such as vi, by setting the environment variable EDITOR.

Set the EDITOR variable to an in-line editor, such as vi, for the history command.

The history command displays the last 16 commands to standard output, by default. For example:

```
$ history
...
157  date
158  cd /etc
159  touch dat1 dat2
160  ps -ef
161  history
```

The numbers on the left are command numbers, which you can use to instruct the shell to re-execute a particular command line.

To display the command history without line numbers, execute the following:

```
$ history -n
```


To display this command and the four commands preceding it, execute the following:

```
$ history -4
```

To display the history list in reverse order, execute the following:

```
$ history -r
```

To display the most recent `cd` command to the most recent `ls` command, execute the following:

```
$ history cd ls
```

The r command

The `r` command is a simple Korn shell command that allows you to repeat a command. For example:

```
$ cal  
May 2000  
  S  M Tu  W Th  F  S  
    1  2  3  4  5  6  
  7  8  9 10 11 12 13  
14 15 16 17 18 19 20  
21 22 23 24 25 26 27  
28 29 30 31
```

```
$ r  
May 2000  
  S  M Tu  W Th  F  S  
    1  2  3  4  5  6  
  7  8  9 10 11 12 13  
14 15 16 17 18 19 20  
21 22 23 24 25 26 27  
28 29 30 31
```

You can use the `r` command to re-execute a command by its number. For example, the shell re-runs 210 in the command history.

```
$ r 210
```

You can use the `r` command to re-execute a command beginning with a particular letter. In this example, the shell re-runs the most recent occurrence of a command that begins with the letter `c`.

```
$ r c
```

You can use the `r` command to repeat a previous command, perform a simple edit, and execute the modified command:

```
$ history
```

```
...
157    date
158    cd /etc
159    ps -ef
$
$ r c
cd /etc
$ r etc=tmp
cd /tmp
```

In this example, the `r` command repeated the most recent occurrence of a command beginning with the letter `c`. It then replaced *etc* with *tmp* and executed the modified command.

Using vi Commands to Edit a Previously Executed Command

You can access a command in the history, edit the command with the `vi` editor, and execute the modified command following these steps.

1. Verify that the default editor for the Korn shell is `vi`.

```
$ echo $EDITOR
/bin/vi
```

2. Type the `history` command to view the command history list.

```
$ history
```

3. Press the Escape key to access the command history list.

Use the following cursor movement keys to scroll up and down the command history.

- ▼ `k` = Move the cursor up one line at a time
- ▼ `j` = Move the cursor down one line at a time

Use the next set of cursor movement keys to scroll to the left and to the right on the command line.

▼ l = Move the cursor to the right

▼ h = Move the cursor to the left

Note – You cannot use the arrows keys to move the cursor. The h, j, k, and l keys are used for cursor movement within the command history.

4. Use vi commands to edit any previously executed command.
5. To execute a modified command, press the Return key.

The Korn Shell Alias Utility

An alias is a shorthand notation in the Korn shell to allow you to customize and abbreviate UNIX commands. An alias is defined by using the `alias` command.

Command Format

```
alias name=command_string
```

For example:

```
$ alias dir='ls -lF'
```

The shell keeps a list of aliases that it searches when a command is entered. If the first word on the command line is an alias, the shell replaces that word with the text of the alias.

When an alias is created, the following rules apply:

- There is no space on either side of the equal sign
- The command string must be quoted if it includes any options, metacharacters, or spaces
- Each command in a single alias is separated with a semicolon

Pre-defined Korn Shell Aliases

The Korn shell contains several predefined aliases, which you view using the `alias` command. Any new user-defined aliases are also be displayed.

```
$ alias
autoload='typeset -fu'
command='command '
functions='typeset -f'
history='fc -l'
integer='typeset -i'
local=typeset
nohup='nohup '
r='fc -e -'
stop='kill -STOP'
```

```
suspend='kill -STOP $$'
```

See Table 11-4 for a listing of alias definitions.

Table 11-4 Alias Definitions

Alias	Value	Definition
autoload	typeset -fu	Defines how to automatically load a function
command	command	Executes a simple command and suppress shell function lookup
functions	typeset -f	Displays a list of functions
history	fc -l	Lists commands from the history file
integer	typeset -i	Declares integer variables
local	typeset	Sets a local attribute for shell variables and functions
nohup	nohup	Keeps jobs running even if you log out
r	fc -e -	Executes the previous command again
stop	kill -STOP	Suspends a job
suspend	kill -STOP \$\$	Suspends a job

User-defined Aliases

Aliases are commonly used to abbreviate or customize frequently used commands. For example:

```
$ alias h=history
$
$ h
278 cat /etc/passwd
279 pwd
280 cp /etc/passwd /tmp
281 ls ~
282 alias h=history
```

283 h

Using the commands `rm`, `cp` and `mv` can inadvertently result in loss of data. As a precaution you can alias these commands with the interactive option. For example:

```
$ alias rm="rm -i"
$ rm dat1
rm: remove dat1: (yes/no)? no
```

By creating a `cp -i` and `mv -i` alias, the shell asks you before overwriting any existing file.

You can deactivate an alias temporarily by placing a backslash (\) in front of the alias on the command line. The backslash prevents the shell from looking in the alias list, causing it to execute the original `rm` command. For example:

```
$ rm file1
rm: remove file1 (yes/no)? no
$
$ \rm file1
$ ls file1
file1: No such file or directory
```

Command Sequences

You can group several commands together under one alias name. Individual commands are separated by semicolons. For example:

```
$ alias info='uname -a; id -a; date'
$ info
SunOS host1 5.8 Generic sun4u sparc SUNW,Ultra-5_10
uid=102(user2) gid=10(staff) groups=14(sysadmin)
Fri Jun 30 15:22:47 MST 2000
$
```

In the next example, an alias is created using a pipe (|) to direct the output of the `ls -l` command to the `more` command. When the new alias is invoked, you get a long directory listing, one screenful at a time. For example:

```
$ alias ll="ls -l|more"
$ cd /usr
$ ll
total 136
drwxrwxr-x   2 root    bin          1024 May 13 18:33
4lib
drwx-----   8 root    bin           512 May 13 18:14
aset
drwxrwxr-x   2 root    bin          7168 May 13 18:23 bin
drwxr-xr-x   4 bin     bin           512 May 13 18:13 ccs
drwxrwxr-x   5 root    bin           512 May 13 18:28
demo
--More--
```

Removing Aliases

The `unalias` command is used to remove aliases from the alias list.

Command Format

```
unalias name
```

For example:

```
$ unalias h
$ h
ksh: h:  not found
```

Note – To pass the new aliases to every shell invoked, it is included in your Korn shell initialization file, usually named `.kshrc`.

Korn Shell Functions

Functions are a powerful feature of shell programming used to construct customized commands.

A function is a group of UNIX commands organized as separate routines.

Using a function involves two steps:

- Defining the function
- Invoking the function

Defining a Function

A function is defined using the general format:

```
function name { command; . . . command; }
```

Some Function Examples

The following example creates a function called `num` to execute the `who` command, directing the output to the `wc` command to display the total number of users currently logged on the system.

```
$ function num { who | wc -l; }
$ num
    9
$
```

The following example creates a function called `list` to execute the `ls` command, directing the output to the `wc` command to display the total number of subdirectories and files in the current directory.

```
$ function list { ls -al | wc -l; }
$ list
   34
$
```

Note – If a command name is defined as a function and an alias, the alias takes precedence.

To view a list of all defined functions, execute the following command:

\$ typeset -f

Configuring of the Korn Shell Environment

You can customize you shell environments by placing two initialization files within the home directory.

- `~/.profile`
- `~/.kshrc`

The Korn shell looks in your home directory for an initialization file called `.profile`, and an environment file conventionally called `.kshrc`. After executing commands found in these files, the shell prompt (\$) appears on the screen and the Korn shell waits for commands.

The ~/.profile File

The `.profile` file is a user-defined initialization file that is executed once at login by the login shell and is found in your home directory. It gives you the ability to customize and modify your working environment.

Environment variables and terminal settings are normally set in this file, and if an application is to be initiated, it can be started here.

If the `.profile` file contains a special variable called `ENV`, the file name assigned to that variable is executed next.

Note – The `/etc/profile` is a system-wide file maintained by the system administrator to set up tasks executed by the Korn shell for every user who logs on.

The ~/.kshrc File

This `ENV` file is often called `.kshrc`, by convention. It contains Korn shell variables and aliases. The `ENV` file is executed every time a Korn shell subshell is started.

Note – You can choose and set any name for the ENV file.

The following items are normally set in the `~/.kshrc` file:

- Shell prompt definitions (PS1, PS2)
- Alias definitions
- Korn shell functions
- History environment variables
- List of Korn shell options (`set -o` option)

Reading Initialization Files

When you make changes to your individual initialization files, those changes take effect when you log in the next time.

However, if changes have been made and you want those changes to take effect immediately, the `.profile` and `.kshrc` files can be sourced using the dot (`.`) command.

```
$ . ~/.profile
$
$ . ~/.kshrc
```

Configuring of the CDE Environment

The .dtpfile File

If you work in the CDE environment, there is another initialization file called `.dtpfile`. It resides in your home directory and determines generic and customized settings for the CDE. The `.dtpfile` file means your preferences for variable settings can overwrite default settings.

There is a standard, system-wide `.dtpfile` file contained in the `/usr/dt/config` directory in the file called `.dtpfile`. The `dtsourceprofile=true` variable is located at the bottom of this file. This standard file is used by CDE to generate a `.dtpfile` file for placement in a user's home directory the first time the user logs in to CDE.

Whenever your log in to CDE, the `.dtpfile` is read, then the Korn shell reads the user's `.profile` file and the `.kshrc` file.

The `.profile` and `.kshrc` files are read again whenever your open a console window. The `.kshrc` file is always read whenever the user opens a terminal window in CDE.

Exercise: Modifying the Korn Shell



Exercise objective – In this exercise, you use the commands learned in this module to redirect standard output and standard error, create command aliases and functions, and use the `history` command for specific tasks.

Tasks

Complete the following steps and write the commands you would use to perform each task in the space provided.

1. Change to your `home` directory and list the contents of the current directory on the same command line. What commands are used?

2. List the contents of the current directory in a long format and direct the standard output (`stdout`) to a file named `dirlist`. Use the `cat` command to display the contents of `dirlist`.

3. Use `cat` to read the file `fruits` and append the standard output to a file called `itemlist`.

4. Turn `noclobber` on and verify you have turned it on with the `set` command.

5. Now list the contents of `dir1` and redirect the standard output to the file called `itemlist`. What message was displayed?

6. This time overwrite the file `itemlist` with the contents of the `dir1` directory. Use the `cat` command to display the contents of `dirlist`.

7. Execute the `date` command and append the standard output to the file `itemlist`.

8. Execute the command `who` and redirect the standard output to a file named `login.list`.

9. Use the `ps -ef` command and the `wc -l` command to display the total number of processes running on your system and append that information to the file `login.list`.

10. Execute the command `ls fruit fruit1 fruit2` and redirect standard output and standard error to the file named `check`. Use the `cat` command to display the contents of `check`.

11. Now execute that command in step 10 again, but this time redirect standard output to a file named `file.list` and redirect standard error to the file named `error.list`. Use the `cat` command to display the contents of both new files.

12. Display all predefined aliases.

13. Create an alias named `cls` that clears the terminal screen.

14. Create an alias named `dir` that displays a long listing of all the files in the current directory.

15. Create an alias named `h` that lists your command history.

16. Execute the following commands:

```
$ date
$ who
$ ls
```

17. Display the history list in reverse order.

18. Re-execute the `who` command.

19. Unalias the `history` command and the `clear` command.

20. Display all defined functions.

21. Create a function called `data` that clears the terminal screen, then displays the date and time, who is logged into your system, the path of the current working directory, and lists the current working directory in a long format.

22. Use the `vi` editor to create and edit the `.profile` in your home directory and add the following line entries:

```
$ vi ~/.profile
```

Add:

```
PATH=/bin:/usr/bin:/usr/ucb
ENV=~/.kshrc
PS1="$(uname -n) [!]"
export PATH ENV PS1
```

23. Use the `vi` editor to create and edit the `.kshrc` file in your home directory, and add the following line entries:

```
$ vi ~/.kshrc
```

Add:

```
set -o vi
alias h='history'
alias cls='clear'
alias lf='pwd ; ls -lF'
```

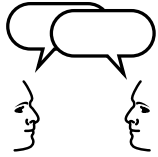
24. Execute the following commands:

```
$ . .profile
```

```
$ . .kshrc
```

25. Test your new aliases and functions.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

Complete the following steps and write the commands you would use to perform each task in the space provided.

1. Change to your home directory and list the contents of the current directory on the same command line. What commands are used?

```
$ cd ; ls
```

2. List the contents of the current directory in a long format and direct the standard output (stdout) to a file named `dirlist`. Use the `cat` command to display the contents of `dirlist`.

```
$ ls -l > dirlist  
$ cat dirlist
```

3. Use `cat` to read the file `fruits` and append the standard output to a file called `itemlist`.

```
$ cat fruit2 > itemlist
```

4. Turn `noclobber` on and verify you have turned it on with the `set` command.

```
$ set -o noclobber  
$ set -o | more
```

5. Now list the contents of `dir1` and redirect the standard output to the file called `itemlist`. What message was displayed?

```
$ ls dir1 > itemlist
```

6. This time overwrite the file `itemlist` with the contents of the `dir1` directory. Use the `cat` command to display the contents of `dirlist`.

```
$ ls dir1 >| itemlist
```

7. Execute the `date` command and append the standard output to the file `itemlist`.

```
$ date >> itemlist
```

8. Execute the command `who` and redirect the standard output to a file called `login.list`.

```
$ who > login.list
```

9. Use the `ps -ef` command and the `wc -l` command to display the total number of processes running on your system and append that information to the file `login.list`.

```
$ ps -ef | wc -l >> login.list
```

10. Execute the command `ls fruit fruit1 fruit2` and redirect standard output and standard error to the file named `check`. Use the `cat` command to display the contents of `check`.

```
$ ls -l fruit fruit1 fruit2 > check 2>&1
```

11. Now execute that command in step 10 again, but this time redirect standard output to a file named `file.list` and redirect standard error to the file named `error.list`. Use the `cat` command to display the contents of both new files.

```
$ ls -l fruit fruit1 fruit2 > file.list 2 > error.list
```

12. Display all predefined aliases.

```
$ alias
```

13. Create an alias called `cls` that clears the terminal screen.

```
$ alias cls=clear
```

14. Create an alias called `dir` that displays a long listing of all the files in the current directory.

```
$ dir="ls -l"
```

15. Create an alias called `h` that lists your command history.

```
$ alias h=history
```

16. Execute the following commands:

```
$ date
```

```
$ who
```

```
$ ls
```

17. Display the history list in reverse order.

```
$ history -r
```

or

```
$ h -r
```

18. Re-execute the `who` command.

```
$ history
$ r number_of_the_who_command
```

19. Unalias the `history` command and the `clear` command.

```
$ unalias h

$ unalias cls
```

20. Display all defined functions.

```
$ function typeset -f
```

21. Create a function called `data` that clears the terminal screen, then displays the date and time, who is logged into your system, the path of the current working directory, and lists the current working directory in a long format.

```
$ function data { clear; date; who; pwd; ls -l; }
```

22. Use the `vi` editor to create and edit the `.profile` in your home directory and add the following line entries:

```
$ vi ~/.profile
```

Add:

```
PATH=/bin:/usr/bin:/usr/ucb
ENV=~/.kshrc
PS1="$(uname -n) [!]"
export PATH ENV PS1
```

23. Use the `vi` editor to create and edit the `.kshrc` file in your home directory and add the following line entries:

```
$ vi ~/.kshrc
```

Add:

```
set -o vi
alias h='history'
alias cls='clear'
alias lf='pwd ; ls -lF'
```

24. Execute the following commands:

```
$ . .profile
```

```
$ . .kshrc
```

25. Test your new aliases and functions.

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Describe the functions of the Korn shell as a command interpreter
- ☐ Demonstrate the use of quoting to mask the special meaning of metacharacters by the Korn shell
- ☐ Define the terms standard input, standard output, and standard error
- ☐ Use metacharacters to redirect standard input, standard output and standard error
- ☐ Connect two or more commands together using the pipe feature
- ☐ Implement the Korn shell file name completion mechanism
- ☐ Describe the two types of variables used by the Korn shell
- ☐ Use commands to view, set, and unset shell variables
- ☐ Invoke the `history` mechanism to repeat or edit previously executed commands
- ☐ Use the `alias` utility to customize and abbreviate UNIX commands
- ☐ Create Korn shell functions to group UNIX commands into separate routines
- ☐ Define the Korn shell initialization files used to customize a user's environment

Objectives

Upon completion of this module, you should be able to:

- Use the stream editor (`sed`) to edit the contents of a text file from the command line and send the results to standard output
- Issue `sed` commands to delete lines, print lines containing a pattern, add text to lines, or change characters using regular expression metacharacters
- Use `awk` to scan text files or standard input to display specific data, change data format, and add text to existing data

The Stream Editor

The `sed` program (or stream editor) is used to edit data in files without opening them in an interactive editor, such as `vi`. It allows you to specify edits, or modifications, to a file from the command line and send the output to the screen by default. This enables you to do repetitive tasks quickly.

The `sed` editor does not change the content of the source file. To save the output, you need to redirect it to a new file.

This editor is best used to make the same changes across multiple files quickly. It is also used to read scripts that administrators need.

Command Format

```
sed [options] [address] command file(s) [>newfile]
```

Note – The `sed` command can also be used in a pipe, for example:
`ls -l | sed`

Options

Options are used to control the behavior of `sed`. The most commonly used option is:

- **-e** – Allows multiple edits on the same command line

Review of Regular Expressions

Similar to the `grep` command, `sed` uses a number of special metacharacters to control pattern searching.

Table 12-1 describes some useful regular expression metacharacters with `sed`.

Table 12-1 Regular Expression Metacharacters

Metacharacter	Purpose	Sample	Result
<code>^</code>	Beginning of line marker	<code>'^pattern'</code>	Matches all lines beginning with "pattern".
<code>\$</code>	End of line anchor	<code>'pattern\$'</code>	Matches all lines ending with "pattern".
<code>.</code>	Matches one character	<code>'p.....n'</code>	Matches lines containing a "p", followed by five characters, followed by an "n".
<code>*</code>	Matches proceeding item one or more times	<code>'[a-z]*'</code>	Matches lowercase alphanumeric characters.
<code>[]</code>	Matches one character in the pattern	<code>'[Pp]attern'</code>	Matches lines containing "Pattern" or "pattern".
<code>[^]</code>	Matches one character not in the pattern	<code>'[^a-m]attern'</code>	Matches lines not containing "a" through "m" followed by "attern".

Using the Stream Editor

Deleting Lines with the `d` Command

The following examples show how `sed` searches for lines containing a *pattern* and removes those lines.

- Search for a pattern within a file and remove all lines containing that pattern. Results display to the screen, and the file is untouched.

```
sed '/pattern/d' filename
```

For example to delete all lines containing the pattern *root* from the file `/etc/group`:

```
$ sed '/root/d' /etc/group
```

The next example would be used to delete all lines containing the pattern "3" in the output of the `ls` command:

```
$ ls -l | sed '/3/d'
```

- Search for specific lines within a file and remove those lines. Results display to the screen, and the file is untouched.

```
sed '#,#d' filename (removes line # to line #)
```

```
sed '#d' filename (removes only line #)
```

```
sed '#,$d' filename (removes line # to the last line)
```

```
sed '$d' filename (removes only the last line)
```

For example, direct the output of the `ls` command to `sed` and remove lines 5 to the last line in the output, placing the results in a new file.

```
$ ls -l | sed '5,$d' > new.file
$ cat new.file
```

Printing Lines with the p Command

The following example shows how `sed` prints all lines to standard output by default, duplicating lines containing *pattern* in addition to all the other lines in a file.,

```
$ sed '/Dante/p' dante
```

In this example `sed` prints all lines to standard output by default. If the pattern `Dante` is found, `sed` prints duplicate lines of output, in addition to all the other lines in the file.

Use the `-n` option to suppress the default action of `sed` when using the `p` command. For example:

```
$ sed -n '/Dante/p' dante
```

In this example, only the lines containing the pattern `Dante` are printed.

Placing Characters at the End of Each Line

The following illustrates how to append characters to the end of lines.

To append the character string `EOL` at the end of every line and display results to standard output, execute the following:

```
$ ls -l | sed 's$/ EOL/'
```

Changing Spaces to Colons in Data

To search for at least one or more spaces, and replace all spaces that are found with a single colon character, execute the following:

```
$ ls -l | sed 's/ */:/g'
```

Multiple Edits with sed

The following shows how `sed` can perform multiple edits on the same command line.

```
$ ls -l > file.3
$ cat file.3
$ sed -e 's/dante/DANTE/g' -e 's/poet/Poet/g' Dante
```

In the previous example, a new file is created, `sed` is then given two edit commands using the `-e` option. The first edit replaces all occurrences of `dante` (lowercase) to `DANTE` (uppercase). The second edit appends the word `poet` (lowercase) to `Poet` (initial capital) and displays results to standard output.

Text Processing Using the `awk` Command

The `awk` command is a flexible text processor used for manipulating columns data and generating reports. It scans a file (or input) line-by-line, from the first to the last line, searching for lines that match a specified pattern and perform selected actions on those lines.

Note – The `awk` command derived its name from the first initial of the last names of those who authored this command; Alfred Aho, Peter Weinberger, and Brian Kenighan.

Some basic uses of `awk` include making format changes to data, reordering columns, and adding to existing text.

Command Format

```
awk '{ action }' filename
```

Basic `awk` Command Format

The basic format of this command consists of the `awk` command, the instructions enclosed in quotes and curly braces, and the name of the input file. If an input file is not specified, then standard input is used, for example, the keyboard.

The following is a basic `awk` command. The output of the `ls -l` command is piped into `awk`. For each line received by `awk`, the `print` action is executed, which prints the output to the screen.

```
$ ls -l | awk '{print $0}'
```

The results of this command are exactly the same output as the `ls -l` command. See Figure 12-1.

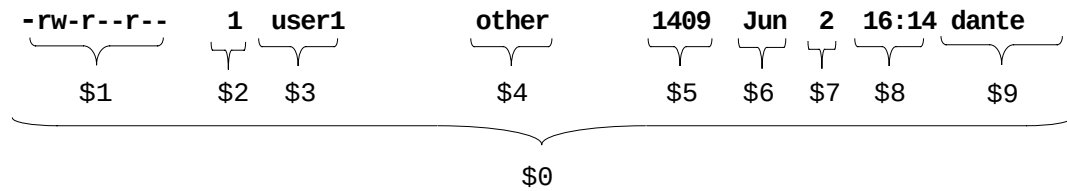


Figure 12-1 Results of the Basic `awk` Command

When `awk` reads in a line it automatically breaks the line into fields. Each field is assigned a variable name. Spaces or tabs are used as the default delimiter between fields.

The variable names assigned to fields are a dollar sign (\$) followed by the number of the field, counting from left to right. The variable name `$1` represent the contents of Field 1. The variable name `$2` represents the contents of Field 2, and so on. The entire line is represented by the variable name `$0`.

Using `awk` to Display Specific Data

To instruct `awk` to display specific data, for example the file owner, file size, and file name, the fields variable names are used with the action.

```
$ ls -l | awk '{print $3 $5 $9}'
user10beans.backup
user154120dante
user1368dante_1
user1176dat
user1512dir1
user1512dir2
user1512dir3
user1512dir4
user1592148file.1
user10file.txt
user1235file1
user1105file2
user1218file3
user1137file4
user135file5
user1715files.tar.Z
```

```
user156fruit
user157fruit2
user12483logfile
user10nuts
user1512practice
root512public_directory
user131script1
user140script2
user122script3
user156script6
user1512shared_directory
user111603text.file.Z
user128738tutor.vi
$
```

The previous output does not include a space between the three fields.

The following shows how to instruct `awk` to place spaces between the fields in the output.

```
$ ls -l | awk '{print $3, $5, $9}'
user1 0 beans.backup
user1 54120 dante
user1 368 dante_1
user1 176 dat
user1 512 dir1
user1 512 dir2
user1 512 dir3
user1 512 dir4
user1 592148 file.1
user1 0 file.txt
user1 235 file1
user1 105 file2
user1 218 file3
user1 137 file4
user1 35 file5
user1 715 files.tar.Z
user1 56 fruit
user1 57 fruit2
user1 2483 logfile
user1 0 nuts
user1 512 practice
root 512 public_directory
user1 31 script1
user1 40 script2
user1 22 script3
```

```
user1 56 script6
user1 512 shared_directory
user1 11603 text.file.Z
user1 28738 tutor.vi
$
```

This example places a space between each field; however, the fields are not aligned.

To provide for exact alignment between fields, press the Tab key to embed a tabbed space between the double quotes.

```
$ ls -l | awk '{print $3 " " $5 " " $9}'
user1 0 beans.backup
user1 54120 dante
user1 368 dante_1
user1 176 dat
user1 512 dir1
user1 512 dir2
user1 512 dir3
user1 512 dir4
user1 592148 file.1
user1 0 file.txt
user1 235 file1
user1 105 file2
user1 218 file3
user1 137 file4
user1 35 file5
user1 715 files.tar.Z
user1 56 fruit
user1 57 fruit2
user1 2483 logfile
user1 0 nuts
user1 512 practice
root 512 public_directory
user1 31 script1
user1 40 script2
user1 22 script3
user1 56 script6
user1 512 shared_directory
user1 11603 text.file.Z
user1 28738 tutor.vi
$
```


Using *awk* to Change the Format of Data

You can instruct the *awk* command to rearrange the fields for the purpose of changing the format of the data.

To format the data to display the file name first, then file size and then the file owner:

```
$ ls -l | awk '{print $9,$5,$3}'
beans.backup 0 user1
dante 54120 user1
dante_1 368 user1
dat 176 user1
dir1 512 user1
dir2 512 user1
dir3 512 user1
dir4 512 user1
file.1 592148 user1
file.txt 0 user1
file1 235 user1
file2 105 user1
file3 218 user1
file4 137 user1
file5 35 user1
files.tar.Z 715 user1
fruit 56 user1
fruit2 57 user1
logfile 2483 user1
nuts 0 user1
practice 512 user1
public_directory 512 root
script1 31 user1
script2 40 user1
script3 22 user1
script6 56 user1
shared_directory 512 user1
text.file.Z 11603 user1
tutor.vi 28738 user1
$
```

To format the data to display the file owner, file name, and file creation or modification date. (To provide exact alignment between these fields, press the Tab key between the double quote.)

```
$ ls -l | awk '{print $3 " " $9 " " $6,$7}'
user1  beans.backup    May 1
user1  dante    Apr 16
user1  dante_1  Mar 22
user1  dat      May 2
user1  dir1     May 1
user1  dir2     Mar 22
user1  dir3     Mar 22
user1  dir4     Mar 22
user1  file.1   May 1
user1  file.txt      Apr 12
user1  file1    May 1
user1  file2    Mar 22
user1  file3    Mar 22
user1  file4    Mar 22
user1  file5    Mar 22
user1  files.tar.Z    May 1
user1  fruit    Mar 22
user1  fruit2   Mar 22
user1  logfile  May 1
user1  nuts     May 1
user1  practice      Mar 22
root   public_directory    Mar 31
user1  script1  Apr 15
user1  script2  Apr 15
user1  script3  May 1
user1  script6  Mar 31
user1  shared_directory    Mar 31
user1  text.file.Z    May 1
user1  tutor.vi      Mar 22
$
```

Using awk to Add Text to Data

The awk command can be instructed to rearrange the fields as well as add new text between fields.

```
$ ls -l | awk '{print $9,"is using",$5,"bytes"}'
file1 is using 405 bytes
file2 is using 66 bytes
file3 is using 66 bytes
```

```
file4 is using 66 bytes  
file5 is using 46598 bytes  
file6 is using 1409 bytes  
$
```

When adding text on an `awk` command line, each text insert must be enclosed in double quotes, and all but the last text insert must be followed by a comma, as shown above.

Exercise: Using `sed` and `awk`



Exercise objective – In this lab you use `sed` to edit a file and the `awk` command to match patterns in files using regular expressions.

Tasks

Complete the following steps and write the commands you would use to perform each task in the space provided.

1. Use `sed` on the file `file1` and remove all lines containing the word “Achievers”. Let the output of `sed` display to the terminal screen.

2. Use the output of the `ls` command with `sed` to remove all lines containing the number 0. Redirect the output of `sed` to a new file called `new.file`.

3. Using `sed`, remove line number 2 to line number 7 from the file `fruit`.

4. Using `sed`, delete the last line of the file `fruit`.

5. Direct the output of the `ls` command to `sed` and remove lines 5 to the last line in the output. Place the results in a new file, called `results.file`.

6. Using `sed`, substitute all occurrences of the string “the” with the word “COOL” in the file `dante`.

-
7. Using `sed`, delete the first eight characters of each line in the file `file3`.

 8. Using `sed`, append an asterisk (*) character to the end of each line in the file `fruit2`.

 9. Append the word "COOL" to each occurrence of the string "the" in the file `dante`.

 10. Use `sed` to print all lines of `file1` to standard output, including those lines containing the pattern "Achievers".

 11. Using `sed`, print only the lines in `file1` that contain the pattern "Achievers".

 12. Direct the output of the `ls -l` command to `awk` to display the file owner, file size, date, time, and file name for each line.

 13. Re-enter the same command from step 11 and instruct `awk` to provide exact alignment between the five fields.

 14. Direct the output of the `ls -l` command to `awk` to rearrange fields to display the file size, file owner, and file name. Instruct `awk` to provide exact alignment between the fields.

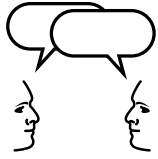
15. Direct the output of the `ls -l` command to `awk` to display the following three fields: file name, file owner and file size. At the same time, instruct `awk` to add the following text between the fields:

Add the text “belongs to” between the file name and the file owner fields.

Add the text “and it is” between the file owner and the file size fields.

Add the text “bytes in size” after the file size field.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

Complete the following steps and write the commands you would use to perform each task in the space provided.

1. Use `sed` on the file `file1` and remove all lines containing the word "Achievers". Let the output of `sed` display to the terminal screen.

```
$ sed '/Achievers/d' file1
```

2. Use the output of the `ls` command with `sed` to remove all lines containing the number 0. Redirect the output of `sed` to a new file called `new.file`.

```
$ ls -l | sed '/0/d' > new.file
```

3. Using `sed`, remove line number 2 to line number 7 from the file `fruit`.

```
$ sed '2,7d' fruit
```

4. Using `sed`, delete only the last line of the file `fruit`.

```
$ sed '$d' fruit
```

5. Direct the output of the `ls` command to `sed` and remove lines 5 to the last line in the output. Place the results in a new file, called `results.file`.

```
$ ls -l | sed '5,$d' > results.file
```

6. Using `sed`, substitute all occurrences of the string "the" with the word "COOL" in the file `dante`.

```
$ sed 's/the/COOL/g' dante
```

7. Using `sed`, delete the first eight characters of each line in the file `file3`.

```
$ sed 's/^.....//' file3
```

8. Using `sed`, append an asterisk (*) character to the end of each line in the file `fruit2`.

```
$ sed 's/$/ */' fruit2
```


9. Append the word “COOL” to each occurrence of the string “the” in the file `dante`.

```
$ sed 's/the/COOL/g' dante
```

10. Use `sed` to print all lines of `file1` to standard output including those lines containing the pattern “Achievers”.

```
$ sed '/Achievers/p' file1
```

11. Using `sed` print only the lines in `file1` that contain the pattern “Achievers”.

```
$ sed -n '/Achievers/p' file1
```

12. Direct the output of the `ls -l` command to `awk` to display the file owner, file size, date, time, and file name for each line.

```
$ ls -l | awk '{print $3 $5 $6 $7 $9}'
```

13. Re-enter the same command from step 11 and instruct `awk` to provide exact alignment between the five fields.

```
$ ls -l | awk '{print $3 " " $5 " " $6 " " $7 " " $9}'
```

14. Direct the output of the `ls -l` command to `awk` to rearrange fields to display the file size, file owner, and file name. Instruct `awk` to provide exact alignment between the fields.

```
$ ls -l | awk '{print $5 " " $3 " " $9}'
```

15. Direct the output of the `ls -l` command to `awk` to display the following three fields: filename, file owner and file size. At the same time instruct `awk` to add the following text between the fields.

Add the text “belongs to” between the filename and the file owner fields.

Add the text “and it is” between the file owner and the file size fields.

Add the text “bytes in size” after the file size field.

```
$ ls -l | awk '{print $9,"belongs to",$3,"and it is",$5,"bytes in size."}'
```

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Use the stream editor (`sed`) to edit the contents of a text file from the command line and send the results to standard output
- ☐ Issue `sed` commands to delete lines, print lines containing a pattern, add text to lines, or change characters using regular expression metacharacters
- ☐ Use `awk` to scan text files or standard input to display specific data, change data format, and add text to existing data

Objectives

Upon completion of this module, you should be able to:

- Identify which shell program interprets the lines of a script
- Explain how command line arguments are passed to a shell script with special variables called positional parameters
- Demonstrate the use of three conditional commands: `if`, `test`, and `case`
- Interpret the contents of a simple Bourne shell administration script

Additional Resources



Common Desktop Environment: Desktop Korn Shell User's Guide,
Part Number 802-6495

The Basics of Shell Scripts

A shell script is simply an ASCII file that contains a sequence of commands and comments.

Comments are text used to document what the script does and describe what the lines within the script are supposed to do when it is executed. Comments are preceded by a hash (#) symbol and can be on a line by themselves or on the same line following a command.

Determining the Type of Shell Script

The top line of a script identifies the shell program that executes the lines in the script. For example, for a Bourne shell script, the first line would read:

```
#!/bin/sh
```

The #! is used by the kernel to identify the program that interprets the lines in the script.

Note – The first line in a Korn shell script would read `#!/bin/ksh`. The first line in a C shell script would read `#!/bin/csh`.

However, not all shell scripts use the top line to identify the shell program.

Some shell scripts might just have a regular text comment on the top line. In this case, the same shell as the parent shell would be used to execute the script.

Other shell scripts might have no comment at all on the top line. If this is the case, the default shell is used to execute the script.

Creating a Basic Shell Script

The following steps demonstrate how to build a basic program.

1. With the `vi` editor, create a file called `my.script`, and enter each command followed by a Tab and hash (#) symbol to define it as a comment.
2. Make the file executable, and run the new script by typing its name on the command line.

```
$ vi my.script
who # To view who is logged on the system.
date # To view the current date and time
ls -l # To view files in current directory

:wq

$ chmod 755 my.script

$ ./my.script
```



Caution – When naming shell scripts, do not use the words `script` or `test`. These are names of actual commands in the Solaris Operating Environment.

Bourne Shell Programming

The standard administration scripts used to manage the Solaris Operating Environment are Bourne shell scripts.

To successfully control or modify the behavior of the operating environment, system administrators must be able to read, modify, and customize the contents of these various shell script files.

Note – Currently there are two comprehensive Sun Educational Services courses on shell programming available: Shell Programming for Programmers and Shell Programming for System Administrators.

In general, all Bourne shell scripts consists of a combination of UNIX commands, Bourne shell built-in commands, programming constructs, and comments.

Bourne Shell Scripts

To be able to read and, more importantly, comprehend the contents of a basic shell script, you must be able to:

- Understand how arguments are passed to a script, through the use of special built-in variables called positional parameters
- Identify and analyze simple conditional constructs and flow control

Special Shell Built-in Variables

Positional Parameters

Special shell built-in variables, called *positional parameters*, are used to pass arguments from the command line into a script.

On the command line, each word, separated by a space, that follows the script name is called an argument. These arguments are referenced in the shell script with positional parameters.

Command Format

```
scriptname argument1 argument2 argument3 ...
```

When the script is executed, the shell automatically stores the first argument on the command line in the positional parameter `$1`, the second argument in the positional parameter `$2`, the third argument in `$3`, and so on.

Resetting Positional Parameters

The `set` command is used to reset the positional parameters based on the arguments.

The following example illustrates how the `set` command and positional parameters work within a script.

First execute the `who` command with the `-m` option and view the output.

```
$ who -m
user1      pts/5          Mar 13 11:43    (host1)
$
```

Now create a shell script with the `vi` command and name it `my.script3`.

```
$ vi my.script3
#!/bin/sh
set 'who -m'
echo Here are the positional variables that have been set:
echo The first is: $1
echo The second is: $2
echo The third is: $3
echo The fourth is: $4
echo The fifth is: $5
echo The sixth is: $6
echo This script is: $0
$ ^d
```

Apply the `chmod` command to make the script executable.

```
$ chmod 755 my.script3
$
```

Execute the new script.

```
$ my.script3
Here are the positional variables that have been set:
The first is: user1
The second is: pts/5
The third is: Mar
The fourth is: 13
The fifth is: 11:43
The sixth is: (host1)
This script is: my.script3
$
```

The Bourne shell allows up to nine positional parameters, \$1 through \$9, and the script name is explicitly referenced in the \$0 variable.

Table 13-1 Positional Parameters

Positional Parameter	Meaning
\$0	References the name of the current shell script
\$1 - \$9	References up to nine positional parameters

Two other useful positional parameters are \$# and \$@.

- **\$#** — Returns the total number of command line arguments stored in positional parameters \$1 through \$9
- **\$@** — Expands the positional parameters to display each stored argument

The following example demonstrates how these two shell built-in variables function within a script.

Create the shell script named `my.script4`, and make it executable.

```
$ vi my.script4
#!/bin/sh
echo The script name is: $0
echo
echo The first argument passed is: $1
echo The second argument passed is: $2
echo
echo The highest numbered parameter is: $#
echo The parameters passed to the script are: $@

$ chmod 755 my.script4
$
```

Execute the new script with the two arguments: stop and start.

```
$ my.script4 start stop
```

```
The script name is: my.script4
```

```
The first argument passed is: start
```

```
The second argument passed is: stop
```

```
The highest numbered parameter is: 2
```

```
The parameters passed to the script are: start stop
```

```
$
```

Conditional Constructs and Flow Control

Conditional Commands

Conditional commands allow you to perform some tasks based on whether a condition succeeds or fails.

The simplest form of a conditional is the `if` command.

The `if` command enables you to test a condition and then change the flow of a script's execution based on the results of the test.

Command Format

```
if  command
then
    command
    command
fi
```

Exit Status

Whenever a program completes execution, within a script or on the command line, it returns an exit status back to the shell. An exit status is an integer between 0 and 255.

Following the generic command format above, the command following the `if` construct is executed and its *exit status* is returned.

If the exit status is 0 (zero) the command succeeded, and the command(s) that follow between the `then` and `fi` constructs are executed. The `fi` terminates the `if` block.

If, however, the first command is executed and the exit status is non-zero, it means the command failed in some way. The statements after the `then` keyword are ignored and the control goes to the line directly after the `fi` statement.

Failures can be caused by invalid arguments passed to the program or by an error condition that is detected.

The shell variable `$?` is automatically set by the shell to the exit status of the last command executed. The `echo` command displays its value.

This simple example demonstrates how to view the exit status of a successful and unsuccessful command execution, from the command line.

```
$ mkdir newdir
$
$ echo $?
0
$
$ mkdir
mkdir: Failed to make directory "newdir"; File exists
$ echo $?
127
$
```

The test Command

The built-in command `test` is often used as the command following the `if` command.

The `test` command evaluates an expression, and if the result is true, it returns an exit status of zero. Otherwise the result is false, and it returns a nonzero exit status.

Command Format

```
if test expression
then
command
fi
```

In the sample below, `test` evaluates the variable `$name` to determine if it has been pre-set to the value `user2`.

```
if test "$name" = "user2"
then
echo "matches"
fi
```

The equal (=) operator is used to test if two values are identical. It tests to see if the contents of the shell variable `$name` is identical to the characters `user2`. If it is, `test` returns an exit status of zero; otherwise it returns a non-zero.

An Alternate Format for test

The `test` command is used so often in shell scripts that an alternative format of the command is recognized by the shell. The bracket (`[`) is actually the name of the `test` command.

The previous statement could be written as follows:

```
if [ "$name" = "user2" ]  
.  
. . .
```

The shell still executes the `test` command, only in this format, `test` expects to see a closing bracket at the end of the expression. Spaces must appear after the first bracket and before the closing bracket.

Command Format

```
if [ expression ]  
then  
    command  
    command  
fi
```

The following example shows `test` evaluating an expression. If the result is true, an exit status of zero is returned. If the result is false, it returns a nonzero exit status.

Create the shell script called `script5`, make it executable and run it.

```
$ vi my.script5
#!/bin/sh
if
name=user5
[ "$name" = "user5" ]
echo $?

then
    name=user7
    [ "$name" = "user5" ]
    echo $?
fi
$
$ chmod 755 my.script5
$ my.script5
0
1
$
```

The test Command Operators

Almost every shell script deals with one or more files, so the `test` command has a wide assortment of file operators. These provide you with the tools necessary to ask questions about files when building a script.

The most commonly used file operators are listed in the following table.

Table 13-2 The test File Operators

Operator	Returns TRUE (Exit Status of 0) if
<i>-d file</i>	<i>file</i> is a directory
<i>-f file</i>	<i>file</i> is an ordinary file
<i>-r file</i>	<i>file</i> is readable by the process
<i>-s file</i>	<i>file</i> has nonzero length
<i>-w file</i>	<i>file</i> is writable by the process
<i>-x file</i>	<i>file</i> is executable

This example shows the use of a file operator to test if a file exists.

Create the script called `script6`, make it executable, and run the program. As a last step, check the results of the exit status.

```
$ vi my.script6
#!/bin/sh
if [ -d $HOME ]
then
    echo "found it!"
fi
$ chmod 755 my.script6
$ my.script6
found it!
$ echo $?
0
$
```

The case Command

The `case` command is used when there are many conditions to test.

Command Format

```
case value in
value1 )    command
            command ;;

value2 )    command
            command ;;

value3 )    command
            command ;;

* )         command ;;

esac
```

The Value of case

The value of a `case` variable is matched against *value1*, *value2* and so on, until a match is found.

When a value matches the **case** variable, the commands following that value are executed until double semicolons (;;) are reached. Then control goes to the line directly after the **esac** statement.

If the value of a **case** variable is not matched, the program executes the commands after the default value *****) until double brackets or **esac** is reached.

Note – The **case** values allow the use of shell wildcards and the pipe symbol for OR-ing two values.

The following example illustrates the use of the **case** command.

```
$ vi my.script7
#!/bin/sh

hour='date | cut -c12-13'
echo $hour

case "$hour" in
    0? | 1[01])
        echo "Good Morning." ;;
    1[2-7])
        echo "Good Afternoon." ;;
    *)
        echo "Good Evening." ;;
esac
$ chmod 755 my.script7
$ my.script7
16
Good Afternoon.
$
```

The **case** command evaluates the variable **\$hour**. The value stored in this case variable is the current time. This is determined when the **date** command piped its output to the **cut** command, which "cut out" the hour by selecting characters 12 and 13 from the output.

The value of the case variable `$hour` is matched against each value in the script until a match is found. Then the result is echoed to the screen.

- The first value tested is to determine if the value of the case variable matches an hour of the morning, (for example, 1:00 am to 11:00 am): `0? | 1[01]`)

`0?` is the number zero and the wildcard character `?` used to match a single character. So the possible values tested for would include: 01, 02, 03, 04, 05, 06, 07, 08, 09.

`|` is an OR statement

`1[01]` is the number one and the wildcard characters `[ ]` used to match a single character. Here the possible values tested for would include 10 and 11.

- The second value tested is to determine if the value of the case variable matches an hour of the afternoon, (for example, 12:00 pm to 5:00 pm): `1[2-7]`)

The possible values tested for include 12, 13, 14, 15, 16 and 17.

- The third value tested is to determine if the value of the case variable matches an hour of the evening, (for example, 6:00 pm to 12:00 am): `*`)

The possible values would include: 18, 19, 20, 21, 22, 23, 24.

The exit Command

The built-in shell command called `exit` gives the user the ability to immediately terminate execution of a shell script.

The `exit` command is frequently used as a convenient method to end, or breakout of, a script and return to the command line.

Command Format

`exit n`

The argument given to the `exit` command is an integer ranging from 0 to 255.

One or more `exit` commands can be placed in a script as a way to instruct the shell to exit the script if a particular condition occurs, prior to the program's normal completion.

In large scripts it is useful to have a wide range of numbers as an argument to each `exit` command to pinpoint where in the script the particular condition occurred.

If the script exits with a zero as an argument, the program exited with success. A non-zero argument (1-255) would indicate some kind of failure.

The user can quickly identify the exit result using the command:

\$ echo \$?

Reading a Sample Solaris Administration Script

The following Bourne shell script is one of many administration scripts maintained in the `/etc/init.d` directory.

These scripts can be used in system startup. Also, these scripts can be used in system shutdown.

A system administrator can manually start and stop system daemons and services by executing an administration script from the command line using the argument `stop` or `start`.

Note – The Bourne shell script is located in `/sbin/sh`.

For example, to manually stop or start the audit program, the root user would invoke one of the following commands.

```
# /etc/init.d/audit stop
```

or

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

Note – If one of these arguments is not included on the command line the following error message will display on the screen: **Usage:**
`/etc/init.d/audit { start | stop }`

Using the newly acquired shell scripting skills, analyze each line in the following script to determine what is happening as the program is executed.

The /etc/init.d/audit Script

```
1 #!/sbin/sh
2 #
3 # Copyright (c) 1997 by Sun Microsystems, Inc.
4 # All rights reserved.
5 #
6 #ident "@(#)audit      1.5      97/12/08 SMI"
7
8 case "$1" in
9 'start')
10     if [ -f /etc/security/audit_startup ]; then
11         echo 'starting audit daemon'
12         /etc/security/audit_startup
13         /usr/sbin/auditd &
14     fi
15     ;;
16
17 'stop')
18     if [ -f /etc/security/audit_startup ]; then
19         /usr/sbin/audit -T
20     fi
21     ;;
22
23 *)
24     echo "Usage: $0 { start | stop }"
25     exit 1
26     ;;
27 esac
28 exit 0
```

Interpreting the audit Administration Script

The following is a line-by-line description of what happens when the audit script is executed.

Line 1: Identifies the shell that is used to execute this program, (for example, the Bourne shell).

Lines 2 - 6: Comments, as each line is preceded by a hash symbol (#).

Line 7: Blank line

Line 8: Begins a case statement. The variable given to case is the positional parameter \$1. This variable passes the first argument on the command line into the script.

Line 9: Identifies the first value to be matched with the variable given to case, (for example, start).

If the two values match, the result is true, returning an exit status of 0 (zero) and control goes down to the next line, Line 10.

If the two values do not match, the result is false, returning an exit status of nonzero and control goes to the line directly after the ;; statement, (for example, Line 16).

Line 10: Begins a conditional if statement with the test command evaluating an expression.

If the file /etc/security/audit_startup exists, the result is true and control goes to the next line, Line 11.

If the file /etc/security/audit_startup does not exist the statements after the then keyword are ignored and control goes to the line directly after the fi statement, (for example, Line 15).

Line 11: Executes the echo command to display the following text to the screen.

starting audit daemon

Line 12: Executes the audit_startup script to initialize the security audit subsystem prior to the audit daemon actually being started.

Line 13: Starts the audit daemon. This program controls the generation and location of audit trail files.

Line 14: Terminates the `if` block statement.

Line 15: Sends control to the last line directly after the `esac` statement. Lines 16 through 27 are skipped (or ignored).

Line 16: Blank line.

Line 17: Identifies the second value to be matched with the case variable, (for example, `stop`).

If the two values match, the result is true, returning an exit status of 0 (zero) and control goes down to the next line, Line 18.

If the two values do not match, the result is false, returning an exit status of nonzero and control goes to the line directly after the `;;` statement, (for example, Line 22).

Line 18: Begins a conditional `if` statement with the `test` command evaluating an expression.

If the file `/etc/security/audit_startup` exists, the result is true and control goes to the next line, Line 19.

If the file `/etc/security/audit_startup` does not exist, the statements after the `then` keyword are ignored and the control goes to the line directly after the `;;` statement, (for example, Line 21).

Line 19: Executes the `/usr/sbin/audit -T` function to signal the audit daemon to close the current audit trail file, disable auditing, and die.

Line 20: Terminates the `if` block statement.

Line 21: Sends control to the last line directly after the `esac` statement. Lines 22 through 27 are skipped (or ignored).

Line 22: Blank line.

Line 23: If the `case` variable did not match the values in either line 9 or line 17 the shell executes the commands on line 24 and line 25.

Line 24: Executes the `echo` command to display the following text string to the screen:

Usage: `/etc/init.d/audit { start | stop }`

Line 25: Executes the `exit` command to immediately terminate execution of the script and return control to the command line. Indicates that some type of failure occurred in the program.

Line 26: Sends control to the last line directly after the `esac` statement.

Line 27: Terminates the `case` block statement.

Line 28: Executes the `exit` command to terminate execution of the script and return control to the command line. Indicates the program exited with success.

Exercise: Introduction to Scripting



Exercise objective – In this exercise you will practice reading the contents of several different shell scripts using the key points covered in the module.

Tasks

Reading Shell Scripts

Examine this particular shell script and then answer the questions that follow.

```
$ cat /etc/init.d/spc
PATH=/bin:/usr/bin:/usr/sbin export PATH

state=$1

set `who -r`
case $state in

'start')
if [ $9 = "2" -o $9 = "3" ]
then
exit
fi

        rm -f /var/spool/print/tf* >/dev/null 2>&1
# clean up after crash
        [ -f /usr/lib/print/printd ] && \
/usr/lib/print/printd

'stop')
        ;;
esac
;;
```

1. What must the value of the first positional parameter (\$1) be for the first branch of the case statement to be executed?

2. After entering the start branch of the case statement, under what condition(s) will this script exit?

3. Commands typically report their execution errors using stderr (standard error). If the rm command in this script experienced an error during its execution, where would the error be displayed?

Below is a segment of a shell script called /etc/init.d/MOUNTFSYS.

Examine this script and then answer the questions that follow.

```
if [ ! -d /usr/sbin ]
then
    echo "/usr sub-tree is not present - changing to single user
mode"
    /etc/init S
fi

# The following counts the number of filesystems with quotas enabled
NQQUOTA=`cut -f 4 /etc/mnttab | egrep -c "^quota|,quota"`

if [ $NQQUOTA -gt 0 ]
then
    echo "Checking UFS quotas: \c"
    /usr/sbin/quotacheck -a -p
    echo "done."
    /usr/sbin/quotachk -a
fi
```

4. Under what condition will the then branch of the first if construct be executed?
-

5. If the value of the variable NQUOTA were zero, would the then branch of the second if construct be executed?
-

Here is a segment of a shell script called /etc/init.d/rpc. Examine this script and then answer the questions that follow.

```
#
# Start the key service.
#
if [ -x /usr/sbin/keyserv ] ; then
    /usr/sbin/keyserv > /dev/console 2>&1
    echo " keyserv\c"
fi
```

6. Under what condition(s) will the then branch of the if construct be executed?
-

7. When /usr/sbin/keyserv is executed, where will the standard output be displayed? Where will standard error output be displayed?
-

Below is a segment of a shell script called /etc/init.d/sysid.sys.

Examine this script and then answer the questions that follow.

```
pkginfo -q SUNWsibi
status=$?

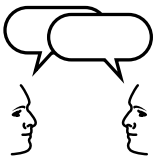
if [ "$status" = "1" -a "$NeedReboot" = "yes" ]
then
    echo "\nrebooting system due to change(s) in \
/etc/default/init \n"
    Net=`need_net_reboot`
    /usr/sbin/reboot -l $Net
fi
```

8. Assume that the `pkginfo` command executes without error. What value will be assigned to the variable called `status`?

9. What is the purpose of the special reserved variable `$_` ?

10. Under what condition(s) will the commands inside the `if` construct be executed?

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

Reading Shell Scripts

1. What must the value of the first positional parameter (\$1) be for the first branch of the `case` statement to be executed?

\$1 must have a value of `start` in order for the first branch of the `case` statement to execute.

2. After entering the `start` branch of the `case` statement, under what condition(s) will this script exit?

The script will exit if the ninth positional parameter (\$9) in the output of the `who -r` command is either a 2 or a 3.

3. Commands typically report their execution errors using `stderr` (standard error). If the `rm` command in this script experienced an error during its execution, where would the error be displayed?

Errors would not be displayed at all since they are being redirected to the “bit bucket” (`/dev/null`).

4. Under what condition will the `then` branch of the first `if` construct be executed?

The `then` branch will execute if the directory `/usr/sbin` does not exist.

5. If the value of the variable `NQUOTA` were zero, would the `then` branch of the second `if` construct be executed?

The value of `NQUOTA` must be greater than zero for the `then` branch of the second `if` construct to be executed. (The “greater or equal to” operator is `-ge`.)

6. Under what condition(s) will the `then` branch of the `if` construct be executed?

The `then` branch will execute if the file `/usr/sbin/keyserv` is executable.

7. When `/usr/sbin/keyserv` is executed, where will the standard output be displayed? Where will standard error output be displayed?

Standard output and standard error output will be displayed on the console (/dev/console).

8. Assume that the `pkginfo` command executes without error. What value will be assigned to the variable called `status`?

The value zero will be assigned to the variable `status`.

9. What is the purpose of the special reserved variable, `$?` ?

`$?` is the variable which always contains the exit status (success or failure) of the most recently executed command.

10. Under what condition(s) will the commands inside the `if` construct be executed?

The commands inside the `if` construct will be executed if the `pkginfo` command does not execute correctly *and* (`-a`) the value of the variable called `NeedReboot` is `yes`.

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Identify which shell program interprets the lines of a script
- ☐ Explain how command line arguments are passed to a shell script with special variables called positional parameters
- ☐ Demonstrate the use of three conditional commands: `if`, `test`, and `case`
- ☐ Interpret the contents of a simple Bourne shell administration script

Index

A

- absolute pathname 3-4
- account
 - root 2-3
 - user 2-3
- alias 11-28
- argument
 - definition 2-15
- awk 12-7

B

- Bash shell 1-9
- bg 10-14
- Bourne shell 1-8, 1-9
- BSD UNIX 1-2

C

- C shell 1-8, 1-10
- cal 2-14
- cancel 4-34
- case 13-14
- cat 4-4
- cd 3-7
- CDE 1-6
- central processing unit 1-3, 1-4
- child process 10-2
- chmod 6-10
- cmp 5-5
- command
 - definition 2-15

- command input/output
 - stderr 11-5
 - stdin 11-5
 - stdout 11-5
- command-line login 2-7
- commands
 - alias 11-28
 - awk 12-7
 - bg 10-14
 - cal 2-14
 - cancel 4-34
 - case 13-14
 - cat 4-4
 - cd 3-7
 - chmod 6-10
 - cmp 5-5
 - compress 8-2, 8-7, 8-8
 - cp 4-20
 - cpio 8-2, 8-15
 - date 2-13
 - diff 5-5
 - egrep 5-17
 - eject 8-22
 - exit 13-16
 - fg 10-13
 - fgrep 5-19
 - file 4-2
 - find 5-2
 - ftp 9-8
 - function 11-32
 - grep 5-12
 - gunzip 8-2
 - gzcat 8-2

gzip 8-2, 8-13
head 4-11
history 11-24
id 2-22
jar 8-2, 8-10
jobs 10-13
kill 10-9
lcd 9-8
lp 4-29
lpstat 4-31
ls 3-10
man 2-17
mkdir 4-18
more 4-8
mv 4-23
passwd 2-10
pg 4-10, 4-13
pgrep 10-5
pkill 10-10
pr 4-35
ps 10-3
pwd 3-8
r 11-25
rlogin 9-4
rm 4-26
rmdir 4-27
rsh 9-6, 9-7
sort 5-8
stop 10-14
strings 4-6
tail 4-12
tar 8-2, 8-3, 8-6
tee 4-16
telnet 9-3
test 13-11
touch 4-15
unalias 11-31
uname 2-13
uncompress 8-2
unzip 8-2
vi 7-5
volcheck 8-19
who 2-20
who am i 2-21
zcat 8-2, 8-9
zip 8-2, 8-12

comment 2-4
Common Desktop
 Environment 1-6
components 1-7
compress 8-2, 8-7, 8-8
computer 1-3
 main components 1-5
conditional commands
 if 13-10
control characters
 Control-c 2-16
 Control-d 2-16
 Control-q 2-16
 Control-s 2-16
 Control-u 2-16
 Control-w 2-16
 definition 2-16
cp 4-20
cpio 8-2, 8-15
CPU 1-3, 1-4

D

date 2-13
delimiter 3-3
delimiters 3-4
diff 5-5
directory
 home 3-3
 login 2-4
 root 3-3
directory tree 3-2

E

egrep 5-17
eject 8-22
environment variables 11-14
execute 6-9
exit 13-16

F

Failsafe session 2-6
fg 10-13
fgrep 5-19
file 4-2

File Transfer Protocol 9-8

files

 /etc/passwd 2-3

find 5-2

ftp 9-8

full pathname 3-4

function 11-32

G

GID 2-4, 6-7

grep 5-12

group identification number 2-4,
 6-7

gunzip 8-2

gzcat 8-2

gzip 8-2, 8-13

H

hard disk 1-4

head 4-11

history 11-24

home directory 3-3

I

I/O 1-4

id command 2-22

if 13-10

input/output 1-4

J

jar 8-2, 8-10

jobs 10-13

K

kernel 1-7, 11-2

kill 10-9

Korn shell 1-8, 1-10

L

LAN 9-2

language 2-6

lcd 9-8

listing

 recursive 3-13

local area network 9-2

local variables 11-14

login

 command line 2-7

 process 2-5

 remote 2-7

 reset 2-7

login directory 2-4

login ID 2-3

Login Manager 2-5

lp 4-29

lpstat 4-31

ls 3-10

M

main components 1-3

man 2-17

menu

 options 2-6

metacharacters

 asterisk 3-16

 dash 3-16

 pipe 4-7

 question mark 3-17

 redirect 4-7

 square brackets 3-18

 tilde 3-15

mkdir 4-18

more 4-8

mv 4-23

O

octal mode 6-10

ONC+ 1-6

Open Network Computing 1-6

Open Windows 1-6

option

 definition 2-15

options menu 2-6

P

- parent process 10-2
- passwd command 2-10
- password
 - requirements 2-9
- PATH variable 11-3, 11-20, 11-21
- pathname 3-3
 - abbreviations 3-9
 - absolute 3-4
 - delimiter 3-3
 - delimiters 3-4
 - full 3-4
 - relative 3-4
- pg 4-10, 4-13
- pgrep 10-5
- pkill 10-10
- placeholder 2-4
- pr 4-35
- process
 - child 10-2
 - parent 10-2
- ps 10-3
- PS1 variable 11-19
- pwd 3-8

R

- r 11-25
- RAM
 - random access memory 1-3
- read 6-9
- recursive listing 3-13
- relative pathname 3-4
- remote login 2-7
- reset login 2-7
- rlogin 9-4
- rm 4-26
- rmdir 4-27
- root account 2-3
- root directory 3-3
- rsh 9-6, 9-7

S

- sed 12-2
- semicolon 2-23

session

- exiting 2-11
- Failsafe 2-6
- shell 1-8, 11-2
 - Bash 1-9
 - Bourne 1-8, 1-9
 - C 1-8, 1-10
 - default 2-4
 - Korn 1-8, 1-10
 - TC 1-9
 - Z 1-9
- shell variable
 - PATH 11-3
 - TERM 11-3
- Solaris
 - directory tree 3-2
- Solaris Operating Environment
 - CDE 1-6
 - components 1-6
- sort 5-8
- stop 10-14
- stream editor
 - sed 12-2
- strings 4-6
- SunoOS
 - history 1-2
- SunOS 1-7
 - kernel 1-7
 - operating system 1-6
 - shell 1-8
- symbolic mode 6-10
- syntax 2-15

T

- tail 4-12
- tar 8-2, 8-3, 8-6
- tar file 8-3
- TC shell 1-9
- tee 4-16
- telnet 9-3
- TERM variable 11-3
- test 13-11
- touch 4-15

U

- UID 2-4, 6-7
- unalias 11-31
- uname 2-13
- uncompress 8-2
- UNIX 1-2
 - AIX 1-2
 - history 1-2
 - man pages 2-17
 - reference manuals 2-17
 - System V 1-2
 - XENIX 1-2
- unzip 8-2
- user accounts 2-3
- user identification number 2-4, 6-7

V

- variables
 - environment 11-14
 - local 11-14
 - PATH 11-20, 11-21
 - PS1 11-19
- vi 7-5
- vi editor
 - command mode 7-3
 - edit mode 7-3
 - last line mode 7-3
- volcheck 8-19

W

- WAN 9-2
- who am i command 2-21
- who command 2-20
- wide area network 9-2
- write 6-9

Z

- Z shell 1-9
- zcat 8-2, 8-9
- zip 8-2, 8-12

Copyright 2000 Sun Microsystems Inc., 901 San Antonio Road, Palo Alto, California 94303, Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées du système Berkeley 4.3 BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd.

Sun, Sun Microsystems, le logo Sun, Solaris Operating Environment, Common Desktop Environment (CDE), admintool, SPARC, et SunOS sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Toutes les marques SPARC sont utilisées sous licence sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays.

Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

UNIX est une marques déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

L'accord du gouvernement américain est requis avant l'exportation du produit.

Le système X Window est un produit de X Consortium, Inc.

LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.



Please
Recycle



Adobe PostScript

