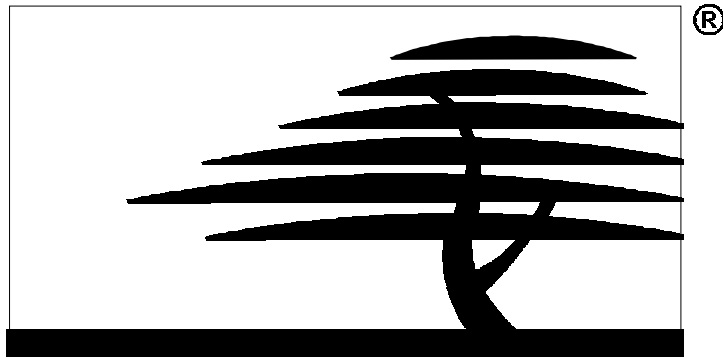




SCO

Introduction to SCO® UNIX® Systems

Student Guide

Course Release: 4.0.2

For Software: UnixWare® 7 and
SCO OpenServer™ Release 5

© 1987–1999 The Santa Cruz Operation, Inc. All rights reserved.

This publication is protected under copyright laws and international treaties.

Information in this document is subject to change without notice and does not represent a commitment on the part of The Santa Cruz Operation, Inc.

SCO, The Santa Cruz Operation, the SCO logos, VP/ix, SCO OpenServer, UnixWare, dbXtra, Panner, SCO Global Access, SCO MPX, SCO Doctor, SCO Doctor for Networks, SCO Doctor Lite, SCO Premier Motif, SCO Visual Tcl, SCO VisionFS, SCO ToolWare, SCO TermLite, SCO Vision 97, SCO CIFS Bridge, SCO WebTop, The Internet Way of Computing, and IWoC are trademarks or registered trademarks of The Santa Cruz Operation, Inc. in the USA and other countries. IXI, X.desktop, Deskworks, Wintif, IXI Panorama, Deskterm, and IXI Desktop are trademarks or registered trademarks of IXI Limited, a subsidiary of The Santa Cruz Operation, Inc. Visionware, Devkit.Vision, Esprit, Kodon, Super.Vision, Visionware Super.Vision, Term.Vision, Visionware Direction, the Visionware logo, Visionware SQL-Retriever, and XVision are registered trademarks of Visionware Limited, a subsidiary of The Santa Cruz Operation, Inc. SuperVision, TermVision, X-Visionware, SQL-Retriever, and Vision Builder are trademarks of Visionware Limited. UNIX is a registered trademark of The Open Group in the United States and other countries. Cheyenne and ARCServe are registered trademarks of Cheyenne Software, Inc. Netscape, Netscape Navigator, Netscape Communications Server, Netscape Commerce Server, Netscape Proxy Server, Netscape FastTrack Server, Netscape Enterprise Server, Netscape SuiteSpot, Netscape Catalog Server, Netscape News Server, Netscape Mail Server, and Netscape Navigator Gold are trademarks of Netscape Communications Corporation. NFS was developed by Legent Corporation (formerly Lachman Associates, Inc.) based on LACHMAN SYSTEM V NFS. LACHMAN is a trademark of Legent Corporation. NFS is a trademark of Sun Microsystems, Inc. TCP/IP was developed by Legent Corporation (formerly Lachman Associates, Inc.) based on LACHMAN SYSTEM V STREAMS TCP, a joint development of Lachman Associates and Convergent Technologies. MPX was developed by Corollary, Inc. VP/ix is a product developed and licensed by Phoenix Technologies, Ltd/INTERACTIVE Systems Corporation. XRemote is a registered trademark of Network Computing Devices, Inc. Oracle is a registered trademark of Oracle Corporation, Redwood City, California. Java is a trademark of Sun Microsystems, Inc. All other brand and product names are or may be trademarks of, and are used to identify products or services of, their respective owners.

The SCO software that accompanies this publication is commercial computer software and, together with any related documentation, is subject to the restrictions on US Government use as set forth below. If this procurement is for a DOD agency, the following DFAR Restricted Rights Legend applies:

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the US Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013. Contractor/Manufacturer is The Santa Cruz Operation, Inc., 400 Encinal Street, Santa Cruz, CA 95060.

If this procurement is for a civilian government agency, this FAR Restricted Rights Legend applies:

RESTRICTED RIGHTS LEGEND: This computer software is submitted with restricted rights under Government Contract No. _____ (and Subcontract No. _____, if appropriate). It may not be used, reproduced, or disclosed by the Government except as provided in paragraph (g)(3)(i) of FAR Clause 52.227-14 alt III or as otherwise expressly stated in the contract. Contractor/Manufacturer is The Santa Cruz Operation, Inc., 400 Encinal Street, Santa Cruz, CA 95060.

The copyrighted software that accompanies this publication is licensed to the End User only for use in strict accordance with the End User License Agreement, which should be read carefully before commencing use of the software.



The SCO® Authorized Education class you are about to take may form part of the recommended preparation for those seeking the SCO ACE™ certification.

The SCO ACE certification program uses a series of exams to measure your ability to support and administer SCO installations with the SCO recommended skills that maximize system performance. As a certified SCO ACE graduate, you earn all of the respect that the industry accords to each certified member of the extended SCO support team. When professionals in the computer industry see your SCO ACE certification, they know that you have passed demanding SCO testing requirements. This certification is the foundation for mastering real-world UNIX® systems, including those from other vendors.

Important!

When registering for any SCO ACE exam at a Sylvan Prometric™ testing center, please quote the course release number on the front of the ***Student Guide*** that the exam covers. This number corresponds to the version number of the SCO ACE exam for that course.

Ask your SCO Authorized Instructor, or call SCO for more information about SCO Certification.

Table of Contents

Lesson 1: Course Introduction	1-1
Introduction.....	1-3
Course goal and purpose.....	1-3
Target audience	1-3
Student Materials	1-4
The <i>Student Guide</i>	1-4
Job aids	1-4
SCO documentation.....	1-4
Typographical Conventions	1-7
Lesson 2: Overview of SCO UNIX Systems	2-1
Computer Concepts and Terminology	2-5
Hardware terms.....	2-5
Software	2-7
Operating system	2-7
System software	2-9
Application software.....	2-9
Functions of an Operating System.....	2-11
Hardware control	2-11
Resource management	2-11
User interfaces	2-13
Input/Output processing.....	2-17
System monitoring.....	2-18
Communications	2-19
Features of SCO UNIX Systems	2-21
Security	2-21
Scalability	2-22
Open technology	2-22
Interoperability.....	2-23
Components of the UNIX Operating System	2-25
System commands	2-25
The shell.....	2-26
The kernel	2-29
Lesson Summary.....	2-30
Answers to Review Questions	2-31
Lesson 3: Accessing the SCO UNIX Operating System	3-1
System Security	3-5
Logging in.....	3-5
Identification	3-6
Changing your password	3-7

Good security habits	3-9
Command-Line Interfaces	3-11
Using Commands.....	3-13
Command syntax	3-13
Entering commands	3-13
Basic commands	3-19
Getting Help.....	3-22
Error messages	3-22
Online man pages.....	3-25
System administrator's services.....	3-26
Lesson Summary	3-27
Labs: Accessing the SCO UNIX Operating System.....	3-28
Lab 1: Logging in	3-28
Lab 2: Using commands	3-28
Relevant Commands and Files	3-31
Answers to Review Questions	3-32

Lesson 4: Files and Directories

4-1

The File and Directory System	4-5
The structure of a file.....	4-9
Types of files.....	4-13
Determining file types	4-15
Navigating Around the Filesystem	4-16
The home directory	4-16
Viewing files and directories	4-17
Pathnames	4-25
Changing directories	4-28
Viewing Ordinary Files.....	4-31
Viewing text files	4-31
Viewing data files	4-34
Organizing Files.....	4-35
Creating directories.....	4-35
Copying files	4-37
Moving and renaming files	4-38
Linking files	4-41
Filename expansion characters (wildcards)	4-45
Removing files and directories	4-49
Working with Files.....	4-53
Finding files and directories	4-53
Finding text within files.....	4-55
Comparing data between files.....	4-57
Lesson Summary	4-59
Review questions	4-59
Lab: Using the Filesystem	4-61
Lab 1: Viewing the filesystem	4-61

Lab 2: Managing files and directories	4-62
Answers to Review Questions	4-66
Lesson 5: File Security	5-1
Introduction	5-3
Maintaining Security	5-5
Ownership	5-5
Access permissions	5-7
Default permissions	5-17
Lesson Summary	5-19
Review questions	5-19
Lab: Securing files	5-20
Relevant Commands	5-21
Answers to Review Questions	5-22
Lesson 6: Text Editing	6-1
Overview of Text Editors	6-4
Uses of a text editor	6-4
Overview of vi	6-5
Using vi	6-7
Invoking vi	6-7
Saving and quitting	6-9
Commands for moving the cursor	6-11
Editing text	6-15
Entering text	6-17
Changing text	6-18
Copying and moving text	6-21
Special commands	6-22
Customizing vi	6-24
Setting options interactively	6-25
Setting options automatically	6-27
The set options	6-27
Checking Your Spelling	6-29
Lesson Summary	6-30
Relevant Commands and Files	6-32
Answers to Review Questions	6-33
Lesson 7: Printing	7-1
Printer Overview	7-5
Spooled printing	7-5
Printing Files with lp	7-7
Using the print spooler	7-7
Options to the lp command	7-8
Monitoring print status	7-11

Canceling print jobs	7-15
Preparing Files for Printing.....	7-16
Printer formatting features	7-16
Printing data files	7-17
Lesson Summary.....	7-18
Lab: Printing	7-19
Overview	7-19
Procedure	7-19
Relevant Commands	7-20
Answers to Review Questions	7-21

Lesson 8: Input and Output	8-1
-----------------------------------	------------

Overview	8-5
Standard input	8-5
Standard output	8-6
Standard error	8-7
File Redirection.....	8-9
Redirecting output.....	8-9
Redirecting input.....	8-13
Pipes, Tees, and Filters.....	8-17
Pipes.....	8-17
Filters	8-18
The tee command.....	8-18
More Filter Commands	8-19
The wc command.....	8-19
The sort command.....	8-21
The tr command	8-21
The pr command	8-22
Lesson Summary.....	8-23
Lab: Input and Output Redirection	8-24
Relevant Commands and Syntax	8-26
Answers to Review Questions	8-27

Lesson 9: Process Control	9-1
----------------------------------	------------

Overview of Processes.....	9-5
Definition of a process	9-6
The Life Cycle of a Process	9-9
Spawning processes	9-9
The life of a process.....	9-10
Terminating processes.....	9-11
Viewing Process Status	9-13
Process Control	9-15
Background processes.....	9-15
Job control.....	9-16
Sending Signals.....	9-19

Signals.....	9-19
The nohup command.....	9-21
Sending signals	9-23
Process Scheduling	9-25
The nice command.....	9-25
The at command	9-25
Lesson Summary	9-27
Lab: Working with Processes.....	9-28
Relevant Commands, Files, and Korn Shell Metacharacters.....	9-30
Answers to Review Questions	9-31

Lesson 10: The Shell Environment	10-1
---	-------------

Shell Review	10-5
Command-line interfaces.....	10-6
Graphical interfaces	10-7
Features of UNIX System Shells	10-9
Advanced Korn shell features.....	10-10
The Shell Environment	10-15
Environment variables	10-15
Environment files.....	10-19
Configuration commands.....	10-20
Lesson Summary	10-21
Lab: Configuring Your Shell Environment.....	10-22
Relevant Commands and Files	10-24
Answers to Review Questions	10-26

Appendix A: Electronic Mail	A-1
------------------------------------	------------

Electronic Mail Overview.....	A-5
The UNIX Mail System.....	A-7
Mail user agents	A-7
Mail transport agents	A-7
Addresses and mailboxes.....	A-8
Using the SCO UNIX mail Utility	A-11
Creating and sending mail	A-11
Reading Mail.....	A-17
Managing Mail.....	A-19
Responding to mail	A-27
Using Aliases	A-28
Customizing Your Mail Environment.....	A-31
Setting options interactively	A-31
Setting options automatically.....	A-33
Adding a signature	A-34
Relevant Commands and Files	A-35

Appendix B: Transferring Files Between Systems	B-1
Overview	B-4
Using tar	B-5
Using cpio	B-13
Using DOS Floppies	B-18
DOS file conversion.....	B-19
Using TCP/IP	B-21
Transferring files with ftp	B-21
Remote login (telnet)	B-25
Finding system names.....	B-26
Relevant Commands, Files, and Syntax	B-27

1

Course Introduction

Lesson 1

Course Introduction

1.1 Introduction

This *Introduction to SCO UNIX Systems* course is designed for users and system administrators who are new to UNIX operating systems.

1.1.1 Course goal and purpose

At the conclusion of this course, students should be able to accomplish basic tasks such as creating, organizing, and removing files, using text editors, printing, and monitoring their processes. Students learn to accomplish these tasks through a command-line interface. By the end of the course, the student will be able to:

- Describe the primary functions, key features, and basic components of SCO UNIX operating systems
- Access the command line interface to navigate file systems and manipulate files and directories
- Use UNIX commands, utilities, and shell capabilities to accomplish common tasks such as editing and printing files
- View, monitor, and control UNIX processes

1.1.2 Target audience

This is an introductory level course, which provides pre-requisite skill training for the SCO operating system administration courses. The course is aimed at new UNIX system administrators as well as end users who work with SCO UNIX systems on a daily basis. There is no course pre-requisite.

1.2 Student Materials

The contents of the box containing your student materials are yours to keep and use for taking notes. Your instructor may have other reference materials in the classroom for your use while attending this class. Please do not take or write in these materials as they will be needed again for later classes.

1.2.1 The *Student Guide*

The *Student Guide* contains:

- Outline of presentation
- Note-taking space
- Copies of all transparencies

1.2.2 Job aids

Job aids are designed to assist students in performing common tasks and as references to other helpful materials. The job aid provided with this course is the ***vi Quick Reference Guide***. This card provides a list of the most commonly used **vi** commands and their functions.

1.2.3 SCO documentation

References to SCO UNIX documentation are made throughout the Introduction to SCO UNIX Systems course.

The SCO OpenServer Release 5 and UnixWare 7 operating systems include extensive online documentation available through the SCOhelp graphical interface. With the exception of installation-related books, SCO operating systems products are not shipped with printed documentation. Hardcopy documentation is available as an optional purchase.

1.2.3.1 OpenServer documentation

The SCO OpenServer documentation includes:

- ***SCO OpenServer Handbook***

This book introduces the product and covers installation and some basic use of the system, including the Desktop, online help, email, starting and stopping the system, configuring hardware, and installing software. This book is available in both online and printed versions.

- ***SCO OpenServer Operating System Administration Guide***

This book describes the configuration and maintenance of the base operating system, including management of accounts, filesystems, printers, backups, security, UUCP, and virtual disks.

- ***SCO OpenServer Operating System Administrator's Reference***

This book contains the manual pages for these sections:

ADM	System administration commands and utilities
F	System file formats
HW	Hardware-specific information
M	Miscellaneous command
TCL	TCL commands

- ***SCO OpenServer Operating System User's Guide***

This book provides an introduction to UNIX command-line utilities, as well as these topics:

- Using SCO Shell utilities
- Working with files and directories
- Editing files with the **vi(C)** text editor
- Transferring files to disk and tape
- Using DOS disks and files in the UNIX environment
- Managing processes
- Shell programming
- Using regular expressions, **awk(C)**, and **sed(C)**

- ***SCO OpenServer Operating System User's Reference***

This book contains the manual pages for ordinary-user (i.e., non-**root** user) operating system commands and utilities (the (C) section).

1.2.3.2 UnixWare documentation

UnixWare 7 documentation is provided online through the graphical **SCOhelp** interface and reference manual pages. In addition to the online **SCOhelp** topics and manual pages, the following printed books are provided with UnixWare 7:

- ***UnixWare 7 System Handbook***

This book covers many commonly-used system administration tasks, such as managing users, starting and stopping the system, working with filesystems, and configuring printers.

- ***UnixWare 7 Getting Started Guide and Release Notes***

This is a printed book that provides the information needed to get the UnixWare 7 system up and running.

1.3 Typographical Conventions

This publication presents commands, filenames, keystrokes, and other special elements as shown here:

lp	Commands, device drivers, programs, and utilities (names, icons, or windows)
<i>/new/client.list</i>	Files, directories, desktops (names, icons, or windows), system, network, and printer names
root	User names
<i>arg</i> or <i>arg</i>	Placeholders (replace with appropriate name or value)
[<i>arg</i>]	Surrounds an <i>option</i> or <i>cmdarg</i> that is not mandatory
<i>arg</i> ...	Indicates multiple occurrences of the <i>option</i> or <i>cmdarg</i>
\$	Command-line prompt for commands executable from the Bourne shell by any user
#	Command-line prompt for commands only executable by the root user from the root account
F2	Keyboard keys
CTRL SPACE	Press CTRL and SPACE keys simultaneously
CTRL ALT DEL	Press all three keys, CTRL , ALT , and DEL , simultaneously
CTRL d	Press CTRL key. With CTRL still depressed, press D .
ALT 0240	Press ALT key. With ALT still depressed, press and release in sequence 0 , 2 , 4 , and 0 .
CTRL Q + i	Press CTRL key. With CTRL still depressed, press Q . Release both keys and press I .
CTRL Q + I	Press CTRL key. With CTRL still depressed, press Q . Release both keys and press SHIFT I .Exit program?System output (prompts, messages)
tar tv	User input, such as on a command line

“Description”	Field names or column headings (on screen or in database)
Cancel	Button names
Edit	Menu names
Copy	Menu items
File ⇨ Find ⇨ Text	Sequences of menus and menu items
open	Library routines, system calls, kernel functions, C keywords
\$HOME	Environment or shell variables
SIGHUP	Named constants, signals, and kernel parameters
“adm3a”	Data values
<i>employees</i>	Database names and tables
buf	C program structures
b_b.errno	Structure members

2

Overview of SCO UNIX Systems

AV 2-1

Overview of SCO UNIX Systems

- Describe basic operating system terminology
- List the primary functions of an operating system
- Identify key features of SCO UNIX systems
- List the basic components of SCO UNIX systems

ntro0201 © The Santa Cruz Operation, Inc.

Lesson 2

Overview of SCO UNIX Systems

Introduction

The UNIX[®] operating system is a multiuser, multitasking system that provides advanced features and attractive solutions in a wide variety of computing environments.

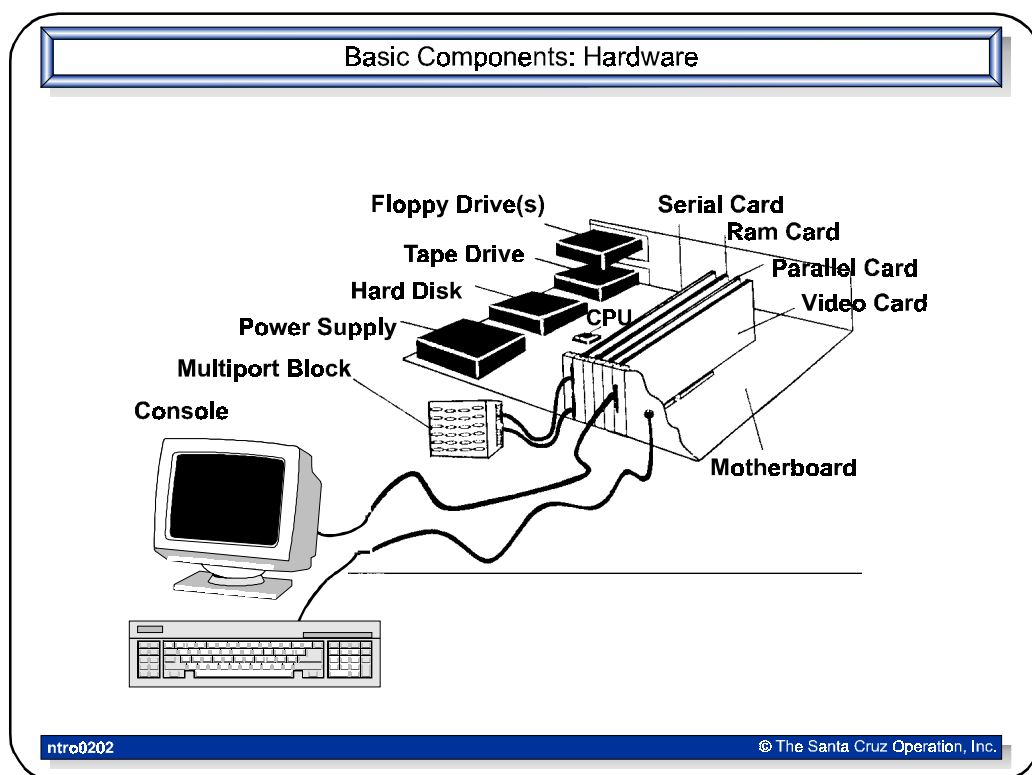
In order to take full advantage of the flexibility and power of this operating system, you should become familiar with the concepts and terminology used to describe how it is put together and what it can do. This lesson describes the capabilities and basic structure of the UNIX operating system.

Objectives

This lesson introduces you to the features of the SCO UNIX operating system. By the end of this lesson, you will be able to:

- Describe basic computer terminology relating to operating systems
- List the primary functions of an operating system
- Identify the key features of the SCO UNIX operating system
- List the basic components of the SCO UNIX operating system

AV 2-2



2.1 Computer Concepts and Terminology

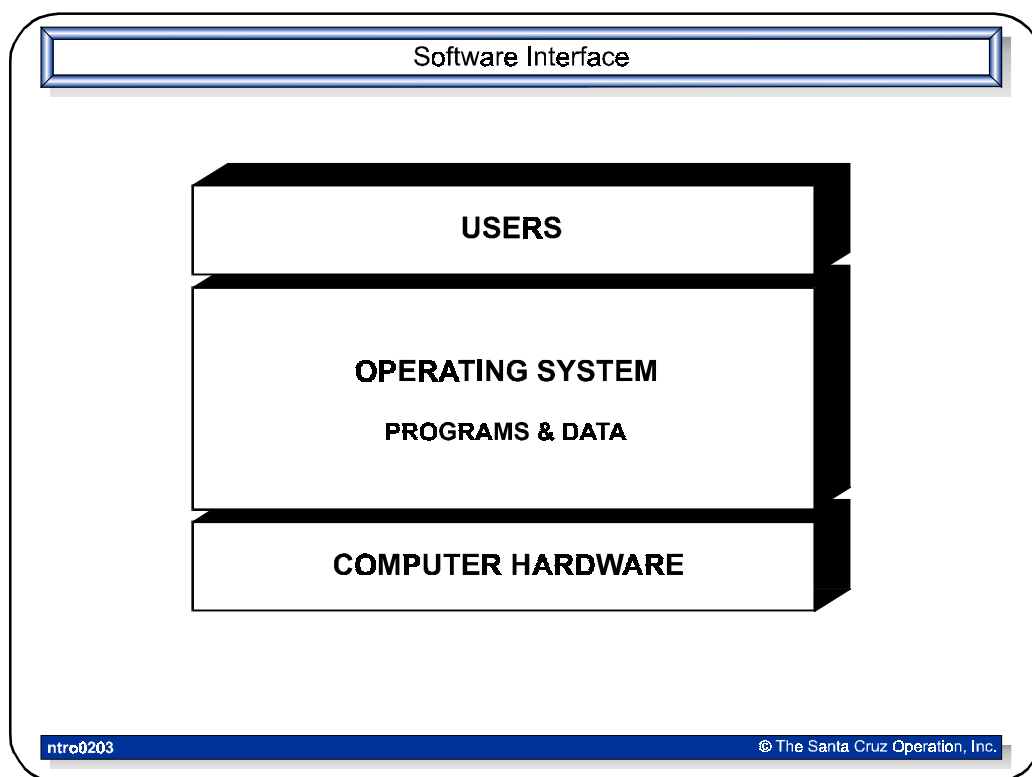
Computers are highly integrated combinations of hardware and software. Understanding computer concepts and terminology is the first step in turning the computer into a tool that will increase your daily productivity.

2.1.1 Hardware terms

The term *hardware* refers to the physical components of a computer that determine its power and capabilities. The hardware resources that most affect a system's performance include:

CPU	The central processing unit—the “brains” of the computer—processes instructions and manipulates data. SCO UNIX runs on Intel's 80x86 family, which includes the i486 and Pentium processors.
RAM	Random access memory stores data and instructions used by the CPU. The contents of this memory can be changed by writing to it.
ROM	Read-only memory also stores data and instructions used by the CPU. The contents of this memory <i>cannot</i> be changed.
Storage devices	Peripheral devices such as hard disks, CD-ROMs, floppy disks, and magnetic tapes store much larger amounts of data and instructions at a lower cost than RAM, but access speed is much slower.
I/O devices	Input and output devices such as consoles, terminals, mice, keyboards, modems, and other peripheral devices provide a means for people to communicate with the computer.

AV 2-3



2.1.2 Software

The term *software* refers to the computer *programs* and associated *data* that tell the CPU what to do.

Programs contain the specific code (i.e., instructions) needed to perform a task. Programs may be very specific commands to the computer or general applications that perform a wide variety of tasks, including word processing and database management.

Data consists of different types of information used by or generated by programs. Data can be used by programs for both input and output. Examples of data are letters, reports, documents, databases, spreadsheets, and graphic images.

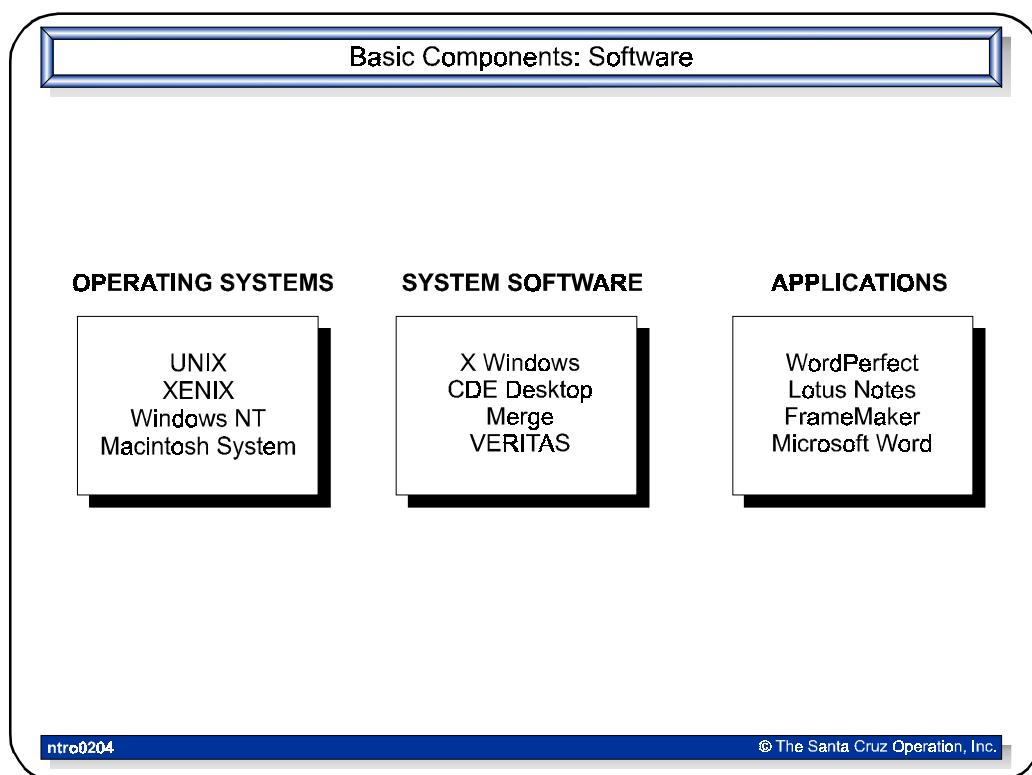
2.1.3 Operating system

An operating system is a software package that provides users with a flexible means of operating and controlling hardware and software resources of a computer. The operating system provides useful services for users and can include:

- General hardware control, such as memory management, file access on peripheral storage devices, print services, and graphical display capabilities
- Task management allows many users to execute multiple applications concurrently
- Communications, including networks and modems on serial lines
- Security, which enables only authorized individuals access to the system or files stored on the system
- User applications, such as word processors, spreadsheets, and database managers

Examples of operating systems include UNIX, XENIX[®], NetWare[®], IBM OS/2[®], Microsoft Windows[®], and MS-DOS[®].

AV 2-4



2.1.4 System software

Layers of software are built upon each other so that complex systems can be developed to provide a wide range of technologies to users. These software layers that are closely tied to the operating system are referred to as *system software*.

System layers commonly include:

Graphical layer	This draws high resolution images on a screen to represent menus, pushbuttons, sliders, and other objects. A pointing device such as a mouse is supported for selecting, moving, and activating a wide variety of graphic objects. Examples include the X server and X Windows.
Network layer	This provides a communication interface between operating systems on different machines. Networks allow hardware and software resources to be shared across different computer architectures and different operating systems, for example, UNIX and Windows. Example network software include TCP/IP and UUCP.
Development system	A development system includes computer languages and a programming interface for creating computer software programs. Example languages include C, C++, Java, Fortran, and Pascal.

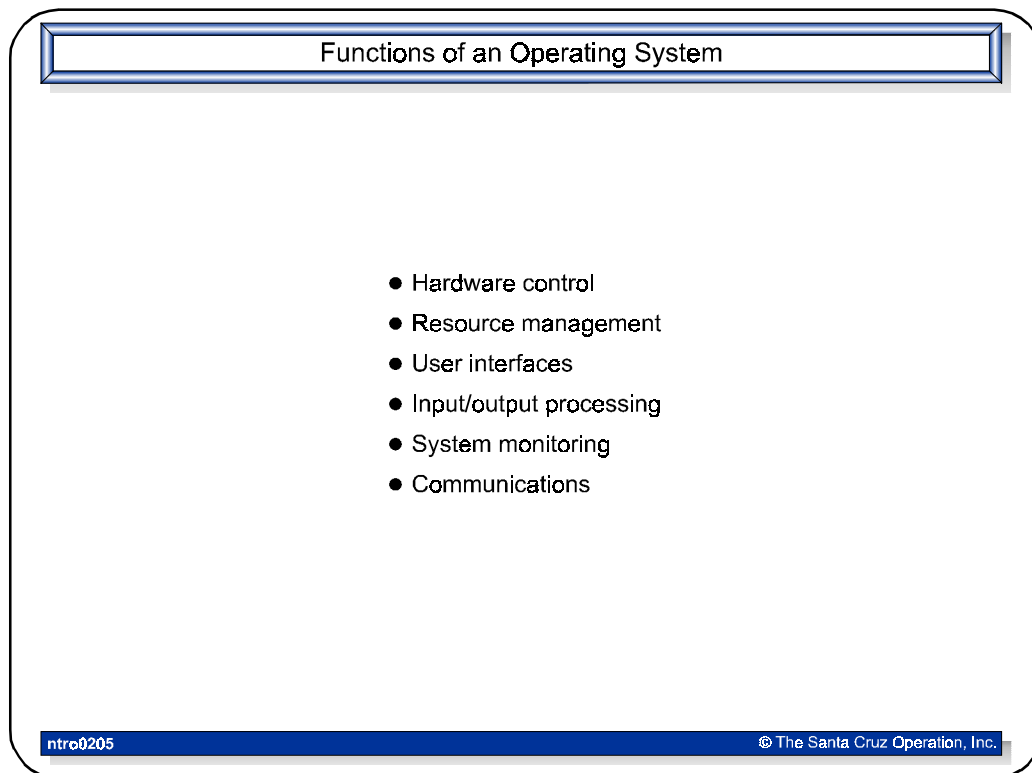
2.1.5 Application software

Application software is tailored directly to user needs and provides direct interaction with the user. Application software utilizes system software to bring the power of the computer to the user in an easy-to-use way.

Applications are large, complex programs or groups of programs that provide services to people using the computer. They are generally created and sold separately from computer systems by third-party vendors.

Examples include word processors, spreadsheets, databases, graphics, desktop publishing, and presentation packages.

AV 2-5



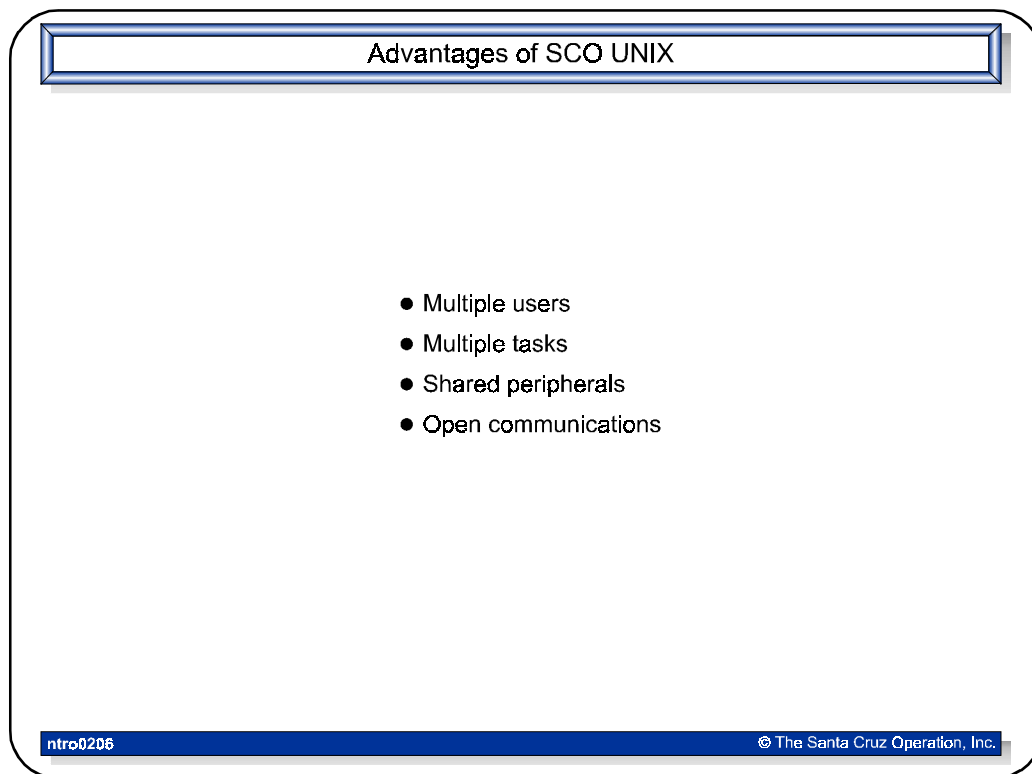
A presentation slide titled "Functions of an Operating System". The title is in a blue-bordered box at the top. The slide contains a bulleted list of six functions: Hardware control, Resource management, User interfaces, Input/output processing, System monitoring, and Communications. At the bottom, there is a blue bar with the text "ntro0205" on the left and "© The Santa Cruz Operation, Inc." on the right.

Functions of an Operating System

- Hardware control
- Resource management
- User interfaces
- Input/output processing
- System monitoring
- Communications

ntro0205 © The Santa Cruz Operation, Inc.

AV 2-6



A presentation slide titled "Advantages of SCO UNIX". The title is in a blue-bordered box at the top. The slide contains a bulleted list of four advantages: Multiple users, Multiple tasks, Shared peripherals, and Open communications. At the bottom, there is a blue bar with the text "ntro0206" on the left and "© The Santa Cruz Operation, Inc." on the right.

Advantages of SCO UNIX

- Multiple users
- Multiple tasks
- Shared peripherals
- Open communications

ntro0206 © The Santa Cruz Operation, Inc.

2.2 Functions of an Operating System

Knowing what an operating system does will help you better utilize its capabilities. SCO UNIX systems interact with the hardware and software resources of a computer by performing these functions:

- Hardware control
- Resource management
- User interfaces
- Input/output processing
- System monitoring
- Communication

2.2.1 Hardware control

The operating system controls and communicates with the computer hardware. It enables applications and other programs to access peripheral devices without having to know the specifications of the hardware. This device independence is important for software to be portable, that is, usable on a wide variety of system configurations.

2.2.2 Resource management

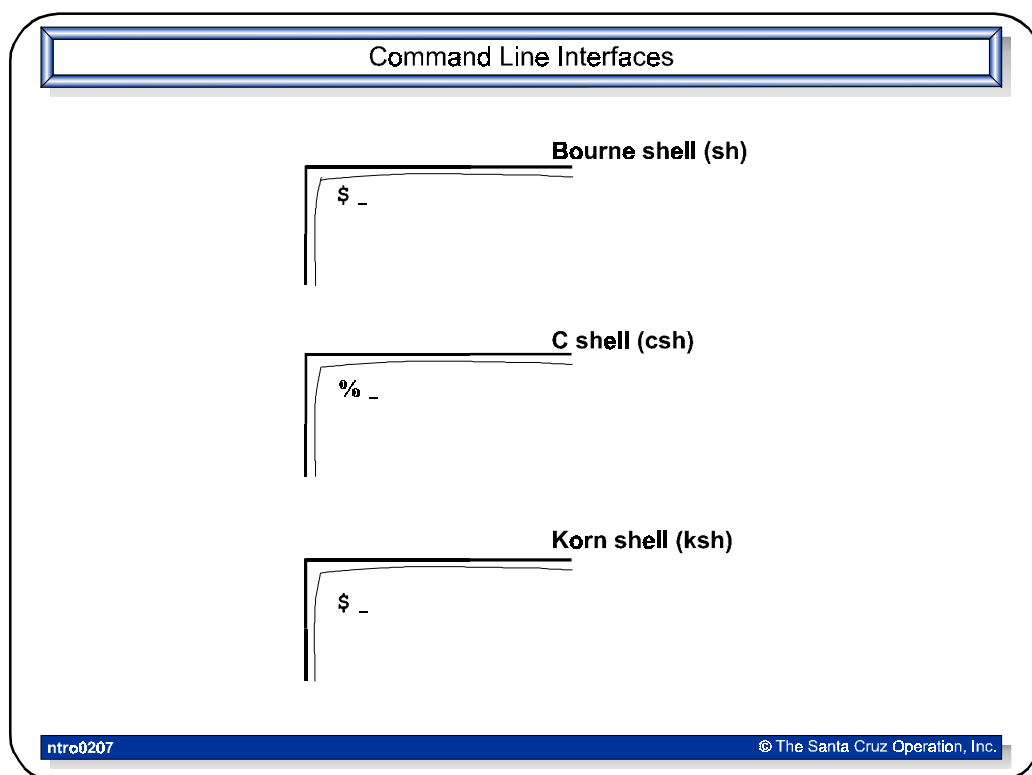
SCO UNIX systems must share the computer's resources among many users performing many different tasks. To accomplish this, the operating system allocates memory to each *program*. A program that is executing in memory is called a *process*. Note the following distinctions in these terms:

- A *program* is a file that contains instructions and data to perform a specific task.
- A *process* is an instance of a program that is in a state of execution. They are the active, working entities of the operating system.

SCO UNIX systems are able to share the computer's memory and other resources because of its *multiuser*, *multitasking* capabilities.

Multitasking refers to the ability of the system to execute several tasks or *processes* concurrently. The CPU and memory resources are shared among several processes, and this can improve the efficiency and productivity of users.

AV 2-7



Multiuser support permits several people to use the computer simultaneously, allowing users to share a computer's resources and expensive peripheral devices such as printers and plotters. One computer supporting many users and sharing expensive resources can be more cost-effective and easier to maintain than a network of single-user machines.

2.2.3 User interfaces

User interfaces define the way people interact with computers. An individual's preferred style of interacting with a computer depends on his or her background and experience level. SCO UNIX operating systems offer several different user interfaces.

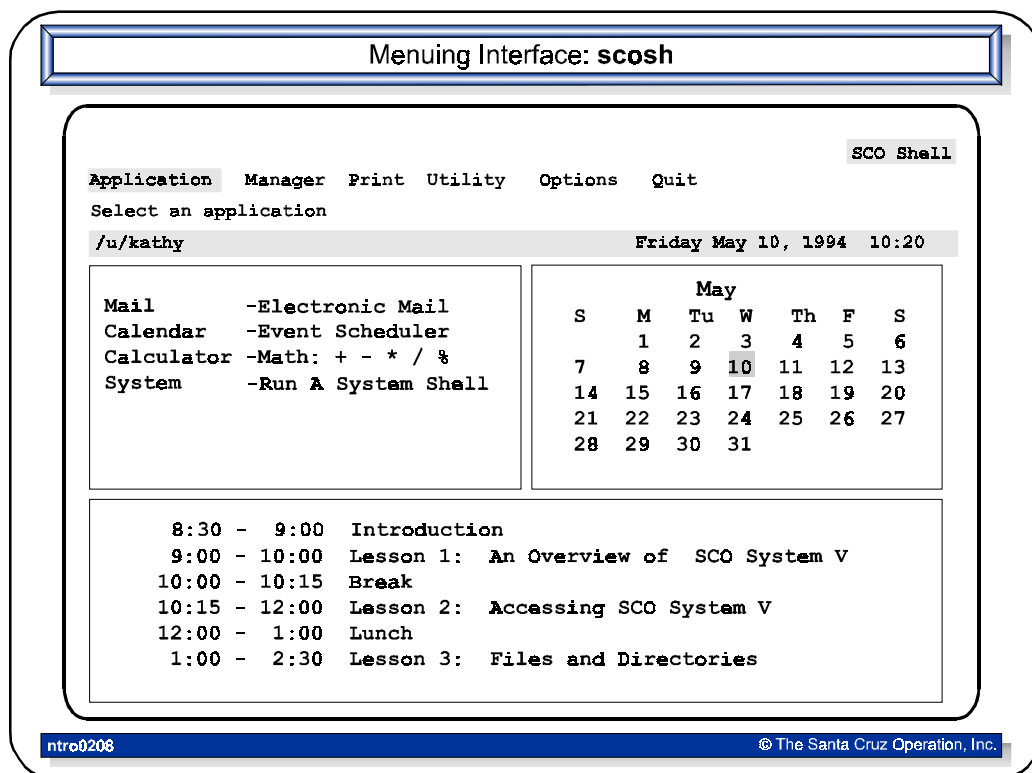
2.2.3.1 Command-line interfaces

Command-line interfaces are designed for users who are experienced with the operating system, and know the commands available and layout of the system. The characteristics of command-line interfaces are:

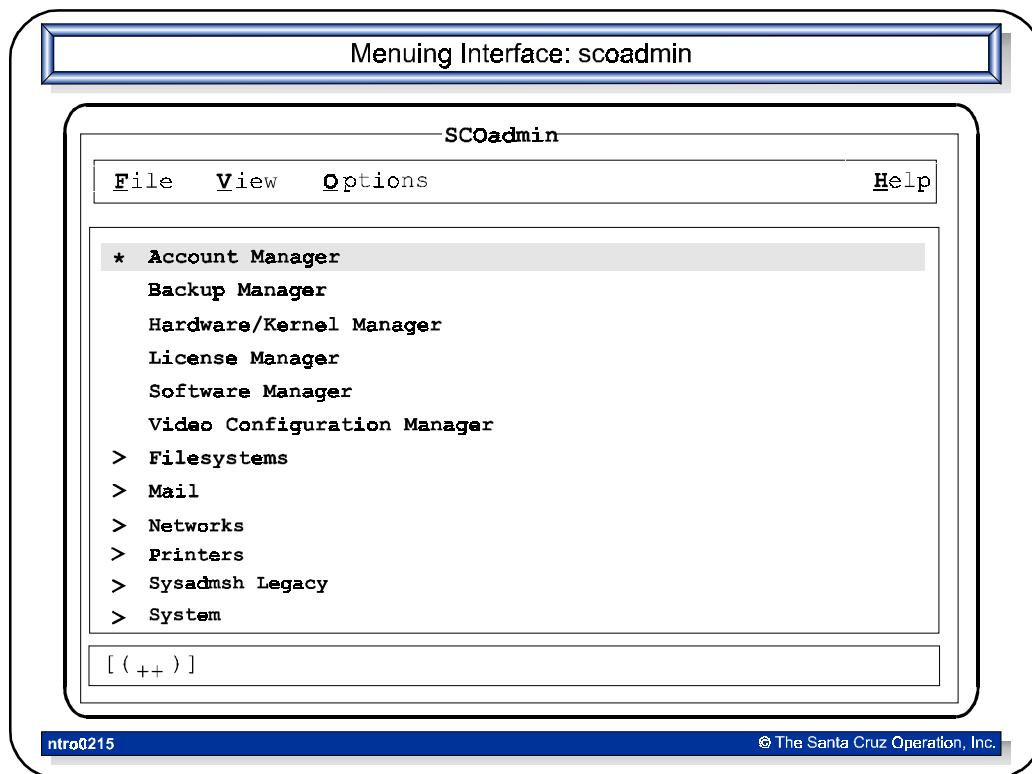
- Prompting the user to enter commands one line at a time, using the keyboard
- Allowing access to all system commands
- Offering the quickest, most direct access to the operating system

Several SCO UNIX programs provide this interface; they are the Bourne shell (**sh**), the C shell (**csh**), and the Korn shell (**ksh**)

AV 2-8



AV 2-9



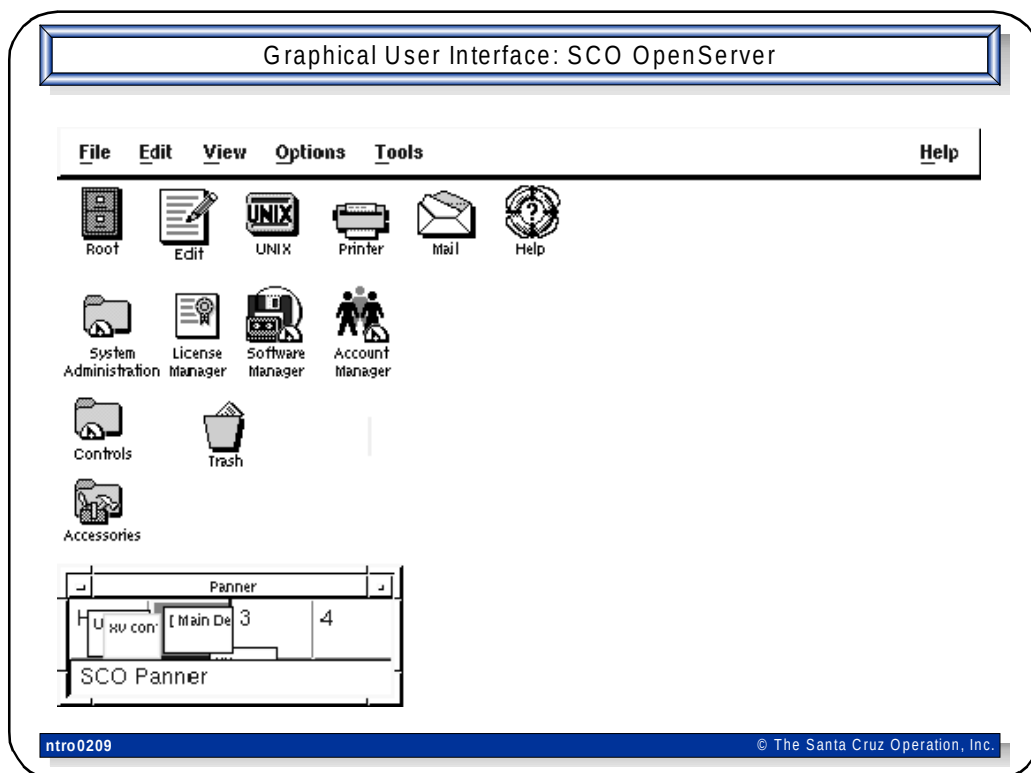
2.2.3.2 Menuing interfaces

Menuing systems are designed for novice users or users who need to interact with only a subset of the operating system. Menuing interfaces offer some guidance to the user. The main characteristics of menuing interfaces are:

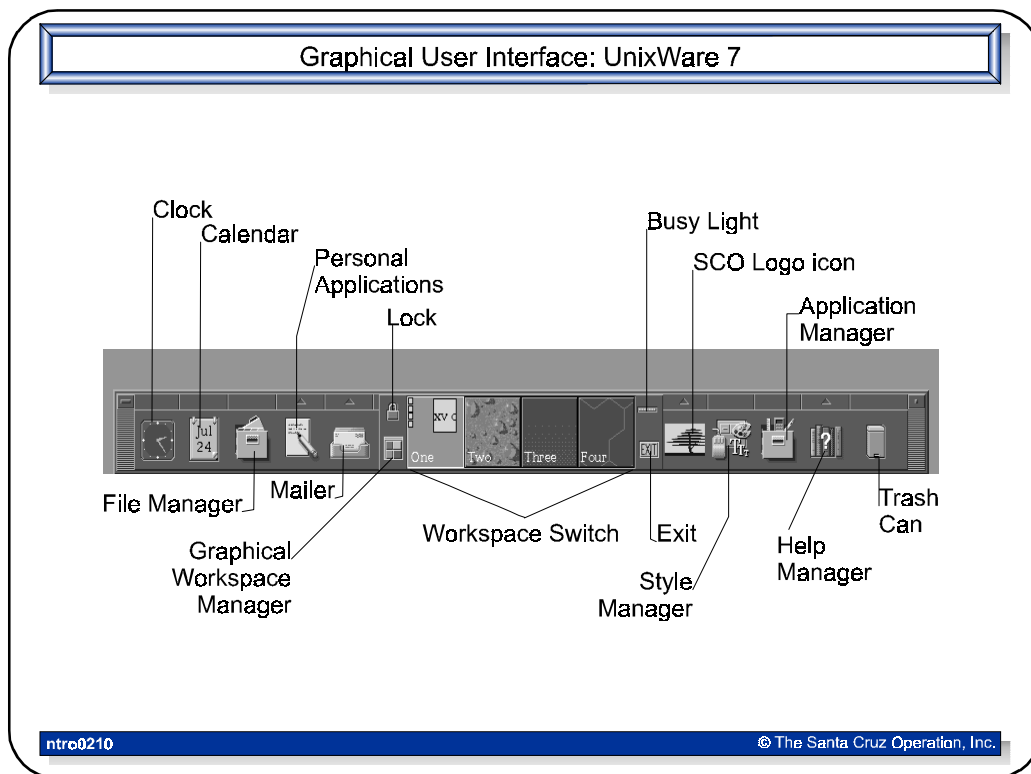
- They allow users to issue a limited number of commands by selecting from menus that list possible actions
- Menus are usually organized in multiple levels through which users narrow their choice
- Users traverse the menus and make their selections using the keyboard, with the addition of keypad and function keys for accelerated access
- Menuing interfaces are quick and easy to use, but limited in scope

Examples of programs that use this interface are SCO Shell in SCO OpenServer and the character-based version of the SCO administration utility (**scoadmin**) in both SCO OpenServer and UnixWare 7.

AV 2-10



AV 2-11



2.2.3.3 Graphical user interfaces

Graphical user interfaces (GUIs) are used by both novice and experienced users. In addition to providing several styles of menus, the visual layout and behavior of graphical user interfaces can be easily configured.

Graphical user interfaces allow several ways of interacting with a computer:

- Viewing the system through graphic images that resemble familiar objects
- Issuing commands through the direct manipulation of images on the screen using a mouse or tracking ball
- Windowing environments, often associated with graphical systems, allow multiple views of the system and the ability to work on several tasks simultaneously

SCO OpenServer systems provide **X.desktop** as a graphical user interface for use on special X-terminals or on the console running under the X Window system.

UnixWare 7 systems provide several graphical user interfaces:

CDE Desktop	UnixWare 7 default desktop based on the industry-standard Common Desktop Environment interface
Panorama	An option with UnixWare 7 that provides a graphical window manager

2.2.4 Input/Output processing

The programs that you run on a system typically require input and produce output. Input may come from a keyboard, a mouse, or a modem line; output may go to the console, a terminal screen, a printer, or modem.

The operating system processes user input into a form meaningful to programs and converts program output into a form meaningful to users.

2.2.5 System monitoring

As a computer system is used, resources are constantly being used and freed up again. Sometimes multiple requests are made to use the same resource at the same time. The operating system must monitor this activity, resolve conflicts and assure that resources are eventually made available to the system again. SCO UNIX systems do this by:

- Ensuring data integrity through record and file locking
- Scheduling processes and auditing system usage
- Providing diagnostic error messages when something goes wrong
- Terminating misbehaving processes and in worst case situations, halting the system

2.2.6 Communications

In modern computing environments, users must be able to communicate with one another. An operating system must provide such communication software and/or support for software that will allow it.

- SCO UNIX systems provide UUCP (UNIX-to-UNIX copy) software which supports modem devices and allows data to be accessed through the phone lines. Electronic mail and facsimile (FAX) transmission are supported in this way.
- SCO UNIX systems support network systems that allow resource sharing with other computers, for example, NFS (Network File System)

AV 2-12

The slide is titled "Features of SCO UNIX Systems" in a blue-bordered box at the top. Below the title, there is a bulleted list of four features: Security, Scalability, Open technology, and Interoperability. At the bottom of the slide, there is a blue footer bar containing the text "ntro0211" on the left and "© The Santa Cruz Operation, Inc." on the right.

Features of SCO UNIX Systems

- Security
- Scalability
- Open technology
- Interoperability

ntro0211 © The Santa Cruz Operation, Inc.

2.3 Features of SCO UNIX Systems

SCO OpenServer Release 5 and UnixWare 7 are advanced operating systems that provide and support many services and features. This section highlights those services and features.

2.3.1 Security

SCO UNIX systems provide secure computing environments. Among the features included are auditing, accountability, authorization, and protected subsystems.

SCO UNIX systems can be installed at one of four different security profiles: *low*, *traditional*, *improved*, and *high*.

2.3.2 Scalability

Both personal computer hardware and SCO UNIX systems are modular. That is, they are built from components that are removable, replaceable, interchangeable and extendable. This allows a system to be scaled up or down so that system resources stay compatible with system usage. This makes SCO UNIX systems very cost-effective.

2.3.3 Open technology

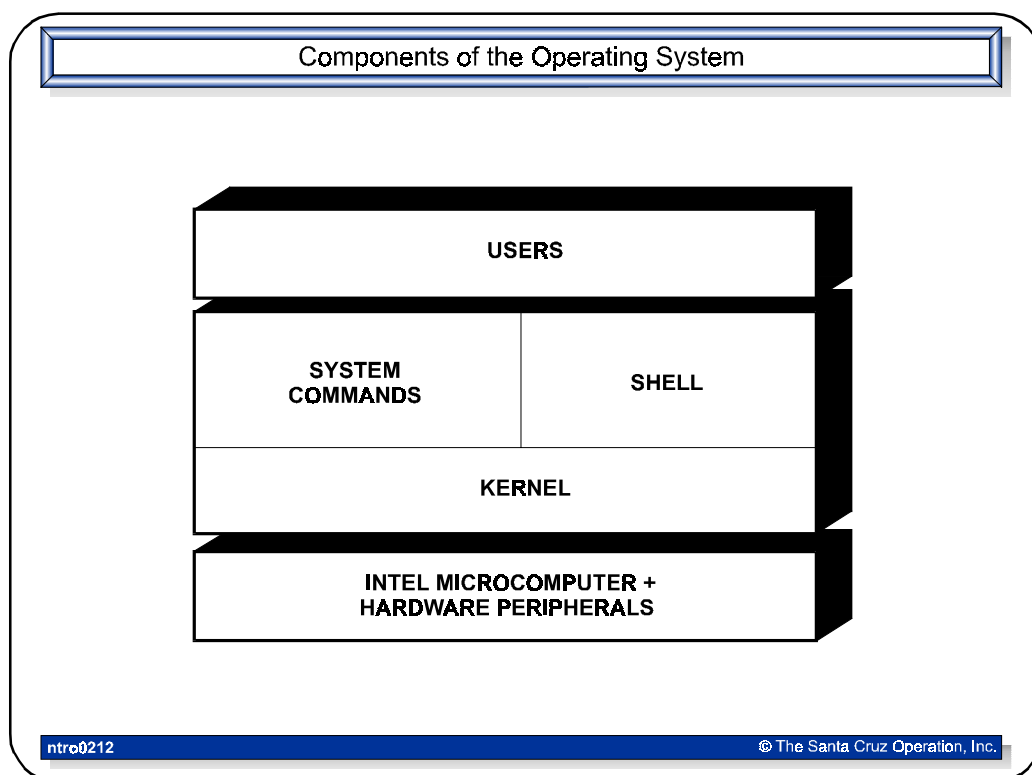
SCO UNIX software is written following well-defined industry standards. These standards define the interfaces through which software modules communicate and interact. By conforming to standards, SCO software remains compatible with future system enhancements and is more easily moved to new environments.

2.3.4 Interoperability

The SCO UNIX operating systems interoperate with non-SCO systems in a variety of different ways and on different levels:

Internet	An extensive set of Internet utilities and services allows SCO UNIX systems to access the full range of Internet resources
Client-Server	SCO UNIX systems provide file storage, printer, terminal, and backup services, as well as business-critical applications for MS-DOS, OS/2 and Windows clients
Workstations	Computing needs can be distributed throughout a network of diverse computer systems running compatible network protocols
Emulation	SCO UNIX systems provide utilities that emulate MS-DOS and Windows platforms and allow DOS/Windows programs to run on SCO systems

AV 2-13



2.4 Components of the UNIX Operating System

Recall that an operating system is a package of software that provides an interface between users and computer hardware. Being a large package of software, it is itself composed of smaller software components that interact through well-defined interfaces. UNIX systems contain three major components: system commands, a shell, and the kernel.

2.4.1 System commands

SCO UNIX operating systems include more than 400 programs that perform such tasks as organizing and processing data and maintaining your working environment.

- Commands are simple programs that direct the CPU to take specific actions. They require a minimal amount of input and run for a limited period of time.
- There are more than 200 commands available to general users of SCO UNIX systems. You will learn approximately 50 of these commands in this course.
- More than 100 additional commands are available to system administrators
- Some of the more complex commands that require a user's interaction to control them are sometimes called utilities
- Examples of utilities introduced in this course include the **vi** text editor and the electronic mailer, **mail**

2.4.2 The shell

The shell is any interactive computer program that provides a user interface to the operating system kernel. The shell that is assigned to your account becomes the active program when you log into the system. It remains active until you exit from it when you log out.

A shell performs the following functions:

- Provides a user interface through which users interact with the system
- Prompts users for commands to be executed and then executes them on behalf of the user
- Provides a configurable environment allowing users to modify commands, prompts, and other system behavior

2.4.2.1 Types of shells

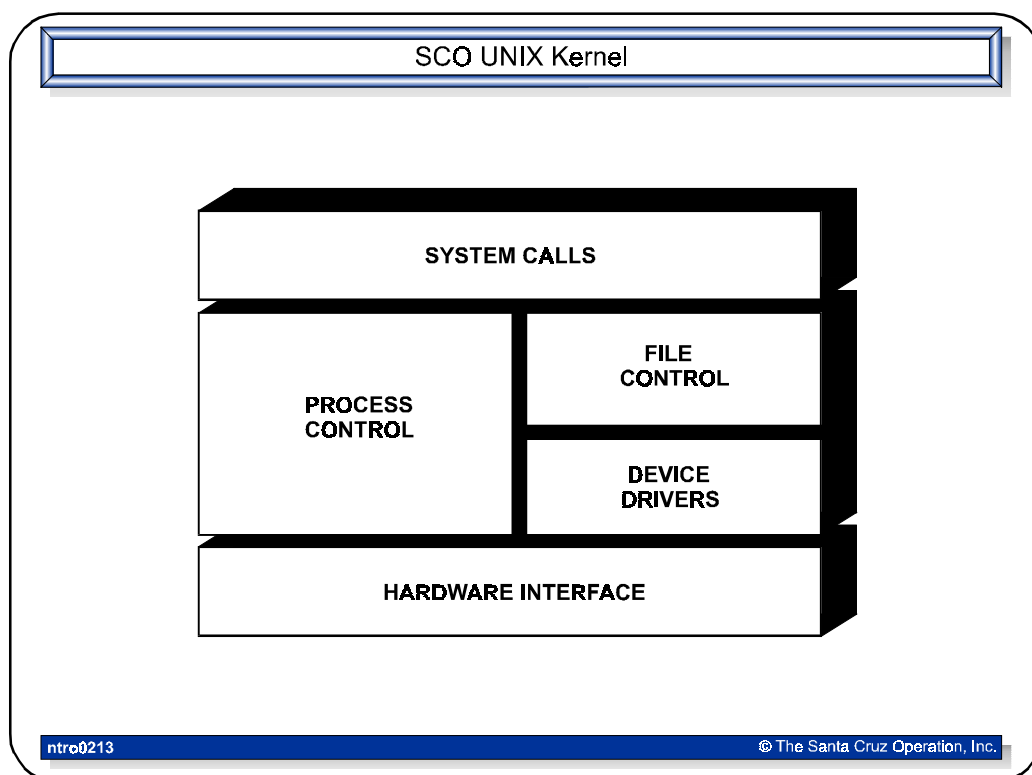
Command-line interfaces (UNIX system shells) include:

Bourne shell (sh)	The standard shell found on UNIX systems
Korn shell (ksh)	An extended version of the Bourne shell
C shell (cs)	The shell popularized by versions of UNIX developed at Berkeley (BSD UNIX)
Restricted shell (rsh , rksh)	Subsets of the Bourne and Korn shells designed for accounts with limited access to the system

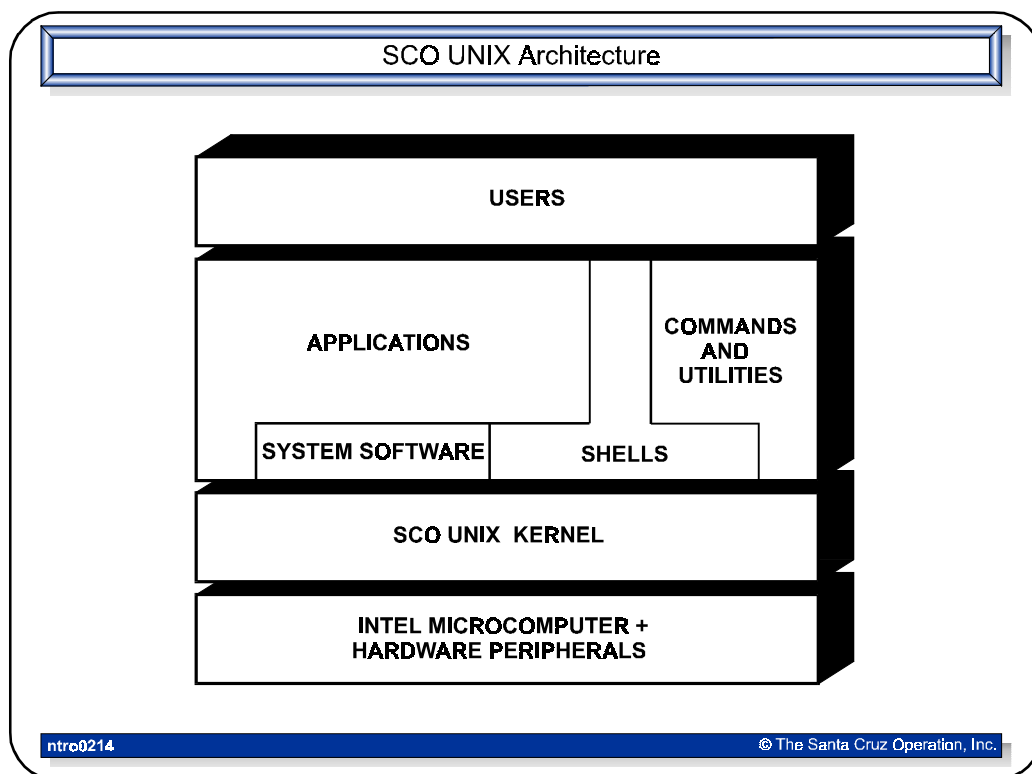
Graphical interfaces include:

SCO Shell (scosh)	A menu-driven program for novice users (available only in SCO OpenServer)
Administration shell (scoadmin)	Graphical and menu-driven versions of a shell designed specifically for system administrators
X.desktop (xdt3)	A graphical desktop for SCO OpenServer
CDE desktop	A graphical desktop for UnixWare 7 systems

AV 2-14



AV 2-15



2.4.3 The kernel

The kernel represents the core of the operating system and is responsible for the computer's basic behavior. Its functions include:

- System call interface—the kernel receives requests through a well-defined set of system calls. These calls are made by users from a shell or some higher level application.
- Process control—the kernel controls the SCO UNIX multitasking environment. It creates the processes in which commands execute. These processes are allowed to run for a small amount of time (called a time slice), and are then preempted, giving another process a chance to use the CPU. Eventually, each process gets enough time slices to finish its job.
- Memory management—the kernel allocates and frees up memory so that processes have the memory they require to execute
- File system management—the kernel manages the storage environment by maintaining the organization, integrity, and security of user data
- Device drivers—the kernel controls and communicates directly with the underlying computer hardware and makes important resources available for user processes

2.5 Lesson Summary

Both computer hardware and software are built from components that fit together and interact through well-defined interfaces. The SCO UNIX operating system provides an interface between users and Intel based computers. It also serves as an interface for additional system and application software.

SCO UNIX is a multiuser, multitasking operating system which makes it a versatile, cost-effective way to meet users' computing needs, whether it is used as a stand-alone system supporting many terminals, a network server providing resources to remote clients, or as a graphical workstation engaged in distributed computing on a network of different computers.

SCO UNIX is itself composed of a kernel, a shell and a collection of commands and utilities.

Review questions

1. Using the terms and concepts discussed in this lesson, describe what an operating system is and does.
2. What are six functions of an operating system?
3. What are the similarities and differences between a standalone multiuser system and a network of single-user machines?
4. List four advanced features of SCO UNIX systems.
5. Describe the major role of each of the three components of SCO UNIX.

2.6 Answers to Review Questions

1. An operating system is a software interface between a computer and the users of that computer which allows you to access both the hardware and software resources that comprise the system..
2. Six functions of an operating system are: 1) controlling hardware, 2) managing resources, 3) providing a user interface, 4) processing input and output, 5) monitoring the system, and 6) providing for communications.
3. The similarities of a standalone multiuser system and a network of single-user machines are that multiple users are supported and expensive resources can be shared by many users. The differences are that users on a multiuser system are sharing the same CPU and memory, where on a network, each user has his or her own dedicated machine. On a network of single-user machines, each machine must be configured and administered, as opposed to everything being managed from one machine on a multiuser, stand-alone system.
4. Four advanced features are: 1) its variable levels of security offered (C2 certified), 2) its scalable nature offering modularity and extensibility, 3) its open technology conforming to industry-wide standards, and 4) its interoperability with other versions of UNIX as well as other operating systems, particularly Windows.
5. The kernel runs the hardware and controls CPU and filesystem access; the shell is the command interpreter, and the system commands and utilities are the programs which perform system tasks.

3

Accessing the SCO UNIX Operating System

AV 3-1

Accessing the SCO UNIX Operating Systems

- Identify secure practices of computer use
- Access the command-line interface
- Perform simple tasks by entering commands
- Identify ways to get help

ntro0301 © The Santa Cruz Operation, Inc.

Lesson 3

Accessing the SCO UNIX Operating System

Introduction

The SCO UNIX operating system is designed so that only authorized users have access to its many commands. Valuable information on its commands may be found both on the system and within printed documentation.

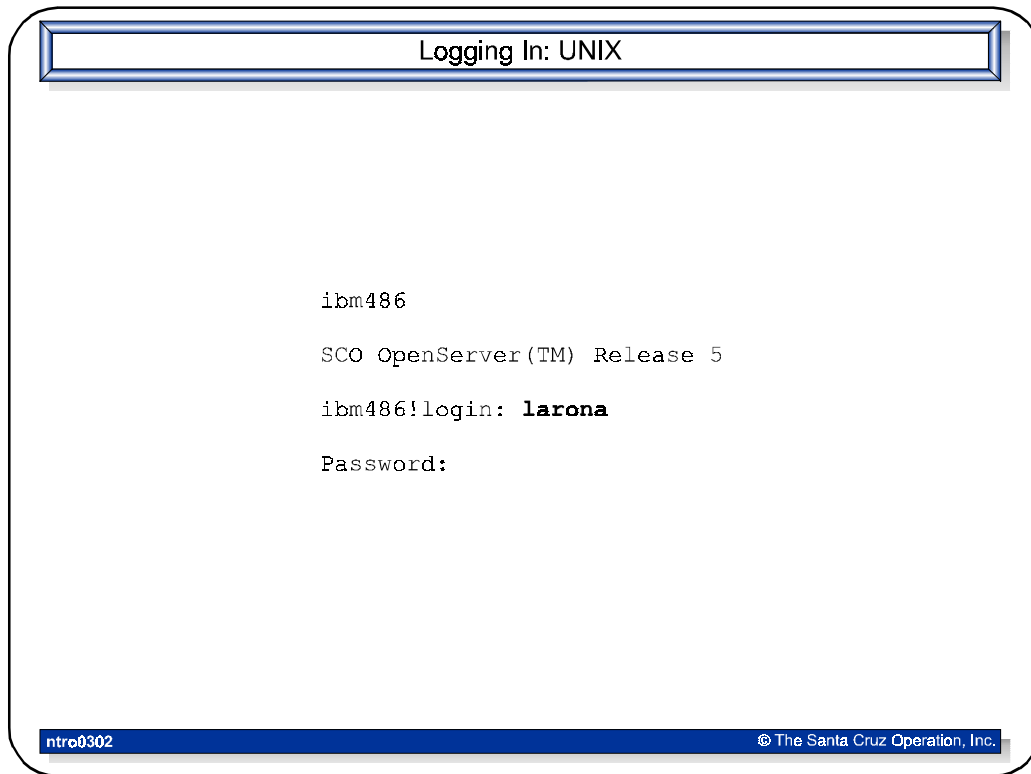
Being able to access this information is important for using the computer as a productive tool. This lesson covers how to access the SCO UNIX operating system in a secure and productive manner.

Objectives

Using an SCO UNIX system, you will learn how to execute system commands in a secure environment. By the end of this lesson, you will be able to:

- Identify secure practices of computer use
- Access the command-line interface of your choice
- Perform simple tasks by entering valid commands
- Identify ways to get help when confronted with problems

AV 3-2



3.1 System Security

As a multiuser and a multitasking operating system, UNIX must share not only hardware resources but software as well. Since information is often of a confidential nature and the users vary in their needs and use of the system, SCO UNIX systems have a procedure for identifying and authenticating users attempting to access the system. That process is called *logging in*.

3.1.1 Logging in

Only authorized users can log into a UNIX system. You become authorized when the system administrator gives you an account, which consists of a name, a password, and a place for your work to reside.

When you log in, you will provide the following information:

Login name

The computer is prepared to accept your login when it displays the following prompt (the administrator can configure the exact wording):

`login:`

Identify yourself in response to the prompt by typing your *login name*, sometimes referred to as your *user name*, and press .

Password

Type your password, in response to the following prompt:

`Password:`

Note that the characters you type do not appear on the screen, nor does the cursor move. This prevents others from learning your password.

Terminal type

Depending on how your account has been configured, you may or may not be prompted to enter the type of terminal you are using:

`TERM=(ansi)`

Within the parentheses is the default terminal type. Press  to accept that default, or type in the correct terminal name. Your system administrator will let you know your terminal type, if necessary.

3.1.2 Identification

When the system administrator creates your account, your login name is associated with two items of identification: your user ID (UID) and your group ID (GID). These values correspond to your user name (login name) and your group name.

3.1.2.1 User name

Your user name is uniquely associated with your user ID number. The system uses this identification to implement various security features of UNIX, such as file ownership and permissions to run various commands. It also uses your UID to audit what you do on the system.

3.1.2.2 Group name

When your account was made, the system administrator gave you a group ID and name that assigned you to a particular group of users. These group names generally include users with common needs, such as all members of an accounting department, and they also help to enforce secure use of the system.

Although you are assigned one group ID that identifies your login group, you may also be a member of other defined groups. If you work on a project that involved several groups of users, you might need to belong to more than one group so you can share information with all of these users.

3.1.2.3 Superuser

Every SCO UNIX system has one special account that has a user name of **root** and a UID of zero. This special value gives all system privileges to this user.

The system displays your user name, UID, group name, GID, and any other groups to which you belong when you enter the following at the system prompt:

```
id 
```

Exercise: Logging in

Log into your system and note any messages that are displayed.
Use the **id** command to see how you are identified to the system.

3.1.3 Changing your password

Your system's security profile determines the control you have over your password. It decides whether you can change your password from the one assigned to you, with what frequency you must change it, and what possible strings of characters are allowable as acceptable passwords.

3.1.3.1 Selecting a password

- A password should be eight or more characters in length and include letters and numbers or punctuation marks. For example:

gr8X*ashn

- A password should not be a name, nickname, proper noun, or a word found in the dictionary spelled forwards or backwards
- Use different passwords on different machines, but don't make the password reflect the names of those machines
- Do not reuse old passwords; you may have revealed one to another user
- Change your password if you think another user knows it
- Keep your password secret. Do not put a copy in your desk, send it over electronic mail, or share it with another person.
- Pick a password that you will remember. There is no way to retrieve your password if you forget it, but the system administrator can change it to a known value.

AV 3-3

Changing Your Password: UNIX

```
$ passwd
Setting password for user: joesco
Old password:
Last successful password change for joesco: Mon May 29 14:40:06 1995
Last unsuccessful password change for joesco: Fri May 26 12:21:01 1995

      Choose Password

You can choose whether you pick a password,
or have the system create one for you.

      1. Pick a password
      2. Pronouncable password will be generated for you

Enter choice (default is 1): 1
Please enter new password (at least 5 characters):
New password:
Re-enter password:
$ _
```

ntro0303

© The Santa Cruz Operation, Inc.

3.1.3.2 Invoking passwd

- Step 1. Change your password to a more secure string by invoking the **passwd** command:

```
passwd  
Setting password for user: login  
Old password:
```

- Step 2. Enter your current password; just as when you logged in, the characters are not echoed to the screen as you type.

If you type your current password in correctly, the date and time of your last change of password is displayed; otherwise, you will have to start the process over again. If these messages do not reflect your last attempts to change your password, tell your system administrator immediately.

- Step 3. Select the option to pick your own password (on OpenServer systems)

- Step 4. Enter your new password when prompted.

You may see a message asking for a password that conforms to the system-defined password guidelines. Try different passwords until one is accepted.

- Step 5. Re-enter your password.

If you cannot correctly duplicate your new password, start the procedure over again by pressing the **DEL** key.

When you re-enter your password correctly, you will return to the shell prompt and your new password will take effect immediately.

Exercise: Changing your password

Change your password following the steps in this section.

3.1.4 Good security habits

In general, do not leave your station unattended, even for a short break.

The operating system measures your idle time, which is time spent not processing any input or output. It may be configured to log you out if your idle time exceeds a predefined limit.

3.1.4.1 Logging out

If you are going to leave your terminal for more than a few minutes, you should log out of the system; thereby eliminating your liability. To log out, use the **exit** command:

exit

The **exit** command terminates your shell and causes the system to run the program that prompts for a new login.

Exercise: Logging out

Log out of your system and log in again using your new password.

3.2 Command-Line Interfaces

When you log into a computer, you are presented with a user interface, which provides a way of interacting with the computer. The command-line interface is the simplest, most powerful user interface. From your system shell, any of the other shell types may be invoked, offering a variety of different user interfaces.

The Korn shell (**ksh**) is the command-line interface used in this course. It acts as your current shell. The default prompt that it provides is the dollar sign (\$). From this prompt, you may invoke any one of the other command-line interfaces provided by SCO UNIX systems by entering its program name at the shell prompt. For example, to run the Bourne shell, enter:

sh

The Bourne shell (**sh**) was the first shell to be developed for UNIX. It provides a basic command language that is a subset of the Korn shell and also uses the dollar sign (\$) prompt.

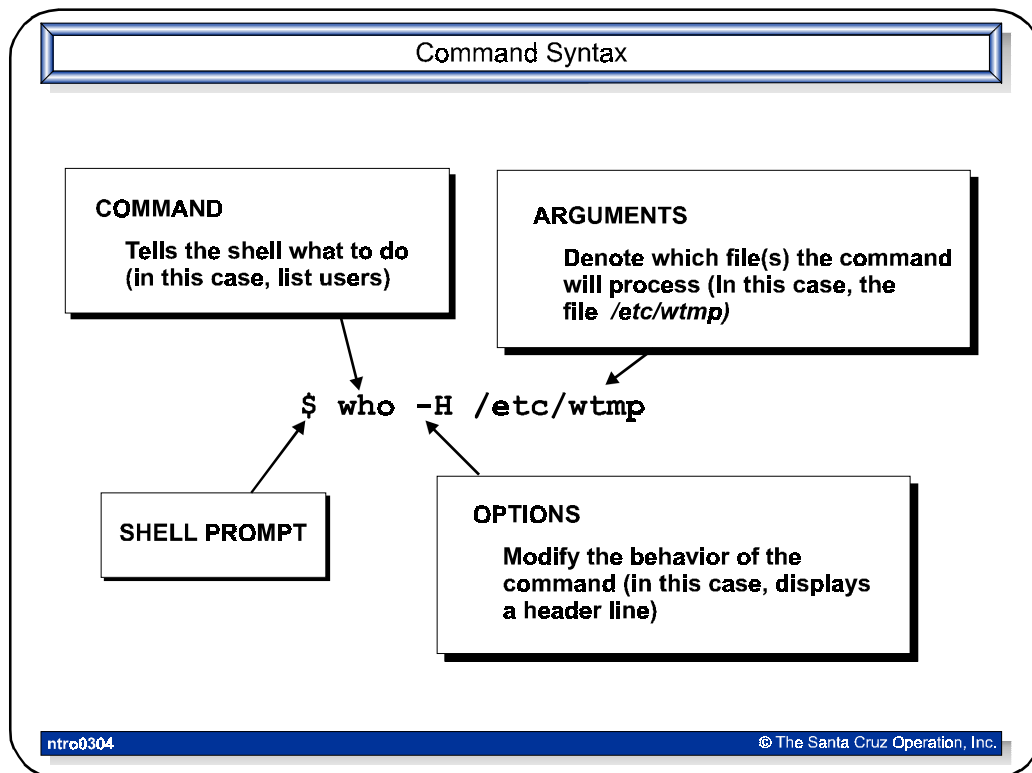
The C shell uses a syntax (similar to the C programming language) that is different from the Bourne/Korn shell family. It uses the percent sign (%) as its default prompt, but as we will see later in the course, you can set the prompt to any desired string of characters.

The **root** account usually uses a pound symbol (#) as its prompt, regardless of which shell it uses.

To find out which of the UNIX command-line interfaces you are running, use the **ps** command.

A command-line interface program will continue running until you explicitly exit by typing the **exit** command.

AV 3-4



3.3 Using Commands

Commands are the fundamental means of communicating with the operating system. They tell the system what to do. Although there are hundreds of UNIX commands, you need to learn only a few of the basic ones to begin using the system.

3.3.1 Command syntax

Just as natural languages have syntax that define the way sentences are constructed, computer programs require commands to be composed in a particular format.

- All SCO UNIX commands use a common syntax:

command [*options*] [*arguments*]

- The use of brackets ([]) means that these portions of the command line are not required and do not necessarily have to be specified with a particular command
- The first part of the command line is the command name itself. It tells the system what action to perform.
- The option(s), sometimes called switches, are specific to the command and modify the behavior of the command. Options are usually represented by single characters preceded by a dash (-). Some commands allow multiple options to be combined into a single string of characters, again usually preceded by a dash.
- The argument(s) either provide information for the command to operate on, or they are names of files for the command to use. Typically, arguments are filenames telling the command where to get its input or where to send the output.
- If a command has no arguments, it usually gets its input, if any, from the standard input (usually the keyboard) and sends its output to the standard output (usually the screen)

3.3.2 Entering commands

- Commands are entered after the shell prompt, and must be terminated with the  key. The shell then reads and executes that command, and displays a new prompt when finished.
- SCO UNIX recognizes the difference between upper and lower case letters. Commands are case sensitive—**DATE**, **Date**, and **date** are interpreted as three different commands, only one of which is an actual UNIX command.

Exercise: Case sensitivity

What happens when you enter the following commands? Why?

DATE
Date
date

- When a command is not found, the shell responds with a “Command not found” message. Check the spelling and case of the command name.
- A command line that is too long to fit on a single line may continue onto additional lines by terminating the first line with a backslash (\) character and immediately pressing the  key (do not press  before ). The shell will respond with a *secondary prompt*, which is usually a greater-than sign (>). Continue your command line on this new line. For example:

```
$ echo This line is too long to fit \  
> on one line  
This line is too long to fit on one line
```

- A space character or tab is required to separate commands from options and arguments. Consecutive spaces are reduced to a single space by the shell.

Exercise: White space

What happens when you enter the following command using multiple spaces between the words:

echo one two three

3.3.2.1 Correcting errors

Although you can use extra spaces, you must use accurate spelling and command syntax. If you mistype a word on the command line, there are several ways to correct the mistake before sending it to the system with the  key.

- Errors may be corrected using the `⌫` key to backspace past the error and then retyping the command correctly (NOTE: Use the “backspace” key—not the “left arrow” key)
- If you wish to discard the entire current line of input, press `CTRL u`. If this is used with the line continuation mechanism (i.e., backslash followed by `ENTER`), only the current input line is discarded—not previous lines which also comprise the command line.
- If you wish to discard the entire command line, press the `DEL` key. If this is used with the line continuation mechanism (i.e., backslash followed by `ENTER`), *all* lines that comprise the command line are discarded. The shell will display a new prompt.

NOTE: The `DEL` key not only discards the line, but it also signals a command. This will usually halt the operation of most commands.

Exercise: Correcting errors

Enter the command:

banner Helo There

and correct the spelling of “Helo” before pressing `ENTER`.

3.3.2.2 Special characters

The shell provides several control and special characters for making command line entering more flexible. Control characters are characters that are typed while also pressing the `CTRL` key.

- `CTRL h` is the default erase key which has the same function as the `⌫` (backspace) key described previously
- `CTRL u` discards an entire line of input which allows you to start over
- `CTRL j` behaves the same as the `ENTER` key. It causes the shell to execute the command line. At times, a program is aborted prematurely, and the `ENTER` key will not work. In such cases `CTRL j` will.
- A pair of single quote (') characters causes the shell to treat all enclosed text exactly as typed without further interpretation. This includes the `SPACE` and `ENTER` key.

- The backslash (\) character implements another quoting mechanism that is similar to the single quotes. It tells the shell to treat the *one* character which follows exactly as typed without further interpretation. The backslash is also used to implement *line continuation* when that one character is the  key.
- A semicolon (;) separates two commands, allowing you to enter more than one command on a single line. The commands are executed in the order given.

Exercise: Metacharacters

Note the shell's secondary prompt character by entering the following command. Press the  key after the word "and":

```
$ echo red 'white      and  
> blue'
```

Enter the following, pressing  immediately after the backslash (\):

```
$ echo This could be a very long command that \  
> needs two lines to complete
```

Enter the command line:

```
date; who
```

Note which output is displayed first. Also note that spaces around the semicolon are optional.

AV 3-5

Basic Commands	
<code>banner [text]</code>	Displays text in a large format
<code>cal</code>	Displays a calendar
<code>clear</code>	Clears the screen
<code>date</code>	Displays the current time and day
<code>echo [text]</code>	Displays the text you type
<code>finger</code>	Displays information on users
<code>who</code>	Displays the users who are currently on the system

ntro0305

© The Santa Cruz Operation, Inc.

3.3.3 Basic commands

The following commands perform a variety of basic tasks and show the use of command line arguments and options.

3.3.3.1 clear

You can use the **clear** command to clear the screen:

```
clear
```

3.3.3.2 date

You can print the current date and time on the screen with the **date** command:

```
date
```

3.3.3.3 cal

The **cal** command displays a calendar for a specified month or year. It takes zero, one, or two arguments. Months can be expressed either numerically or with its full name or abbreviation. The syntax is:

```
cal [month] [year]
```

The following are examples of valid **calendar** command lines:

```
cal  
cal 1752  
cal 9 1995  
cal september 1995  
cal se 1995  
cal sept
```

3.3.3.4 echo

The **echo** command echoes its command line arguments to the standard output (usually the screen). This is useful for displaying messages from shell programs and for displaying the value of shell variables.

```
$ echo Press Enter to continue
Press Enter to continue
$ echo $TERM
ansi
```

3.3.3.5 banner

The **banner** command displays its command line arguments as large words across the screen, one argument per line. Each argument is limited to 10 characters in length.

```
$ banner Hello
#      #
#      # ##### #      #      #####
#      # #      #      #      #      #
##### ##### #      #      #      #
#      # #      #      #      #      #
#      # #      #      #      #      #
#      # ##### ##### ##### #####
```

3.3.3.6 who

The **who** command lists the users on the system. It accepts several command line options:

```
who [-aAbdHlmpqrstTux]
```

When options and arguments are both present, the options must precede the arguments. You can display who is on the system with a header line using the **-H** option:

```
who -H
```

You can list all users who have logged on to the system over time:

```
who -H /etc/wtmp
```

3.3.3.7 finger

The **finger** command provides information on the users of a system:

```
finger [-bfilpqsw] [user1 [user2...]]
```


Display a list of two users with their idle times using the command:

```
finger -i brian andy
```

3.3.3.8 uname

The **uname** command displays information about the operating system. The syntax is:

```
uname [-aAmnrsvX]
```

For example, display all information by using the command:

```
$ uname -a  
SCO Sys V  
ibm486
```

3.4 Getting Help

The SCO UNIX operating system communicates with you in the form of prompts and error messages to help you accomplish your tasks. Additional help is provided both on-line and in the printed documentation.

3.4.1 Error messages

The SCO UNIX operating system is an interactive system. If it does not recognize the information you give it, or if some system limitation prevents it from completing a task, it usually lets you know by displaying an error message.

- Error messages may come from the operating system, a command, or an application that you are currently running
- Error messages from graphical applications pop up in their own window. They must be dismissed by selecting an **OK** button, indicating that you have acknowledged the error.
- Many error messages give suggestions for correcting the error condition
- If you make a syntax error in a command you have entered, the system often responds with a usage message that explains the proper command syntax
- Whenever the system issues an error or warning message, read it carefully. You may need the information to correct a problem.

Exercise: Displaying a usage message

Enter the **who** command with an invalid option:

who -g

What happens? Why?

AV 3-6

Accessing Online Manual Pages

<code>man</code>	Displays a usage message for <code>man</code>
<code>man command</code>	Displays the manual page for the specified command
<code>man -k keyword</code>	Locates commands by keyword (same as the command, apropos)

ntro0306
© The Santa Cruz Operation, Inc.

AV 3-7

The **banner** Manual Page

```

$ man banner
BANNER(C)
Name
    banner - prints large letters
Syntax
    banner strings
Description
    banner prints its arguments (each up to 10 characters long)
    in large letters on the standard output. This is useful
    for printing names at the front of printouts.
See Also
    echo(C)
Standards Conference
    banner is in conformance with:
    AT&T SVID Issue 2, Select Code 307-127
    and the X/Open Portability Guide II of January 1987
(printed 2/15/90)
    BANNER(C)
        
```

ntro0307
© The Santa Cruz Operation, Inc.

3.4.2 Online man pages

Online manual pages are available to all SCO UNIX users. Each command has a corresponding **man** page that contains the following information:

- The *Name* section presents the name and usage of the command
- The *Syntax* section displays the syntax. Anything presented between the brackets ([]) is optional.
- The *Description* section describes more specific uses of the command and the function of each of the available options
- The *Examples* section presents ways the command can be properly executed
- The *See also* section references commands that present additional aspects or explanations of the command
- The *Warning* section describes any special problems that may arise when using the command

3.4.2.1 Using the man command

The **man** command retrieves helpful information when used as follows:

- For information on a specific command, use:

`man command`

- To display a usage message for the **man** command, use **man** with no arguments
- Use **man -k** or the **apropos** command to locate commands by keyword lookup, as if you were looking through an index. The syntax is:

`man -k keyword or apropos keyword`

Exercise: Using the man pages

Use **man** to read the manual page on the **who** command.

Find any commands or references to the keyword *copy* using the **apropos** command. What did you find?

3.4.3 System administrator's services

The system administrator of your SCO UNIX system is another source of help to which you may turn when other resources have not provided a solution to your problem.

- The system administrator has **root** privileges, allowing access to all files
- The system administrator can be called upon to solve a wide variety of problems, since he or she has a knowledge of the reference material and other resources such as technical support
- The system administrator is responsible for keeping the system running properly, creating and removing user accounts, maintaining system security, and helping users

3.5 Lesson Summary

The SCO UNIX operating system limits access to authorized users who have unique name/password combinations that allow them to log into the computer. Once on the system, the power of UNIX is accessed through system commands. Printed documentation and online help facilities are available to help you learn the commands and use the system more effectively. Although an SCO UNIX system has many security features, ultimate security depends on the safe, secure practices of its users.

Review questions

1. What are the security precautions to consider when choosing a password?
2. What should you do with your login session before you leave your station unattended?
3. Name three user interfaces available on an SCO UNIX system.
4. What is a command? An option? An argument?
5. Where should you look for help on using a command?
6. When should you ask the system administrator for help?

3.6 Labs: Accessing the SCO UNIX Operating System

3.6.1 Lab 1: Logging in

Overview

By experimenting with login procedures, you will discover how an SCO UNIX system tracks your computer usage.

Procedure

If you have not logged off the system, do so now by typing **exit** .

1. Enter your login name with a misspelling. Does the system complain about your identity?
2. Enter your password and press . What message is displayed?
3. Enter your correct login name, but mistype your password. How does the message you get this time compare to the previous message?
4. Enter your login name, and at the password prompt, type just the first three letters of your password—do not press .

Now press the  (backspace) key 5 times. Retype your entire password, this time pressing . What happened?

5. With the help of your neighbor, determine if you can log into the system through a different terminal than the one you have been using. (If on the console, use a multiscreen.) Can you log in on one terminal without logging off the other?
6. Try logging in by typing your login name in uppercase. What happens?

3.6.2 Lab 2: Using commands

This lab uses the online manual pages as a reference for learning about commands.

Overview

This lab takes a closer look at invoking various commands that demonstrate the syntax and variety of functions available from SCO UNIX commands.

Procedure

1. Use the **apropos** command to find out all the commands that tell you about who is on the system. Hint: use *who* as the keyword.

Which of the commands listed could you use to find out which shell you are currently running?

What is the difference between the **who** and **w** commands?

What is the name of the terminal port you are using?

How many users are currently logged into your system? Who has been on the system longest? Who has the longest idle time? Use that person's login name as an argument to the **finger** command.

2. Enter the following commands lines:

```
who -H  
who am i
```

Use the reference manual to determine what these options show you.

3. Invoke one of the system shells different from the one you are currently using by typing its name. How does the prompt differ? Exit from the shell you just invoked.
4. Find the prompts of each of the shells introduced in Lesson 2. Be sure you exit from each shell. How can you tell if you have done so?
5. Read the man page on **man** and find the abbreviations for all the section names.
6. Run the **date** command. What time is it? Can you change the time?
7. Run the **cal** command. What day of the week was July 4, 1776? Explain the difference between the following two commands: **cal 12** and **cal December**.
8. Use the **banner** command to output **Hi There!** in one (enlarged) line on your screen. How many arguments must the command have?

9. Use the semicolon metacharacter to issue one command that clears the screen and logs you off the system.

3.7 Relevant Commands and Files

Relevant commands

apropos	Locates commands by keyword lookup
banner	Prints large letters
cal	Prints a calendar
clear	Clears a terminal screen
csch	Berkeley C shell
date	Prints and sets the date
echo	Echoes arguments to the screen
exit	Logs you out of a shell
finger	Displays information about users
id	Prints user and group IDs and names
ksh	Korn shell
lock	Locks a user's terminal
man	Displays the reference manual pages
passwd	Changes login password
sh	Bourne shell
who	Lists who is on the system
whodo	Prints who is doing what

Relevant files

<i>/etc/wtmp</i> and <i>/etc/utmp</i>	Files containing user login entries for use by the who command
---------------------------------------	---

3.8 Answers to Exercises, Labs, and Review Questions

This section provides suggested answers to the exercises, labs, and review questions.

Exercise: Logging in

Use the account name and password provided by the instructor to log into the system. Upon a successful login attempt, the operating system displays a message that **user_name** is logged in and a shell prompt (\$ or %) is displayed.

Exercise: Changing your password

When prompted by **passwd**, enter the password used to log into the system. When the *old* password has been validated and you are prompted, enter and confirm the new password. If no error messages are displayed and the shell prompt returns, you have successfully changed your password.

Exercise: Case sensitivity

Since UNIX is case sensitive, it recognizes only the **date** command. There are no program (executable) files named **DATE** or **Date**; therefore, UNIX displays an error message that the files were not found.

Exercise: White space

The shell parses the command line and collapses multiple spaces and tabs down to one space character.

Exercise: Correcting errors

Use the backspace key to correct the word “Helo”. The backspace key deletes characters as it moves to the left, so the line must be retyped from the point of correction.

Exercise: Metacharacters

The shell provides a continuation line when an unmatched quote character is entered on the command line. When the closing quote is detected and the  key is pressed, the shell executes the command. The \ character also indicates to the shell that the line is continued.

Exercise: Displaying a usage message

The **who** command has not been programmed to recognize a **g** option, so it responds with a “usage” message that informs the user what options are available with this command.

Exercise: Using the man pages

The manual pages contain several topics for the keyword, *copy*: the **copy** command for copying files; and various commands that contain *copy* in their titles or descriptions, for example: **volcopy**, **dcopy**, and **cp**.

Answers to review questions

1. In choosing a password, you should select a sequence of letters and numbers or punctuation symbols that do not represent any known words. It should also be something that you will remember.
2. Before leaving your terminal unattended, you should log off the system. SCO OpenServer systems provide a lock command which can be used to lock your screen to prevent access by someone else.
3. The Korn shell is an example of a command-line interface; the SCO Shell is an example of a menuing interface, and SCO's X.desktop is an example of a graphical desktop interface.
4. A command performs a specific task on the system; an option modifies the behavior of the command in some way; an argument provides information or the names of files on which the command operates.
5. When using a new command you can often get some help from the command itself in the form of a usage message which can usually be obtained by typing the command name with a **-?** option. If that does not work, or provides insufficient help, the online manual pages are available through the **man** command. Additional information is usually available in the documentation. As another resort, you can turn to the system administrator or a knowledgeable friend.
6. You should consult your system administrator when you have forgotten your password, when your system is not running properly, when you need to create or remove a user account, or when you cannot find a solution to your problems in the documentation.

Lab: Accessing the SCO UNIX operating system

This section provides suggested answers to the lab exercises.

Lab 1: Logging in

It is important to log out when you are finished with your work to ensure that someone else does not gain access to the computer, especially where he or she would be using the system with your user name.

Lab 2: Using commands

The **w** command shows what process is running on each terminal (or console screen).

4

Files and Directories

AV 4-1

The slide is titled "Files and Directories" in a blue-bordered box at the top. Below the title is a bulleted list of six items. At the bottom of the slide, there is a dark blue footer bar containing the text "ntro0401" on the left and "© The Santa Cruz Operation, Inc." on the right.

Files and Directories

- Describe the heirarchical directory structure
- Identify the different file types supported by SCO UNIX
- Navigate around the filesystem using cd
- Manage your home directory
- Use filename expansion characters to access groups of files
- Use appropriate commands for finding and comparing files

ntro0401 © The Santa Cruz Operation, Inc.

Lesson 4

Files and Directories

Introduction

When you access SCO UNIX system commands, you access information stored in a structured environment. All of this information is stored in a hierarchical structure that allows you to maintain your data in an orderly and accessible way. It is important that you learn not only how to access this data, but to control its access.

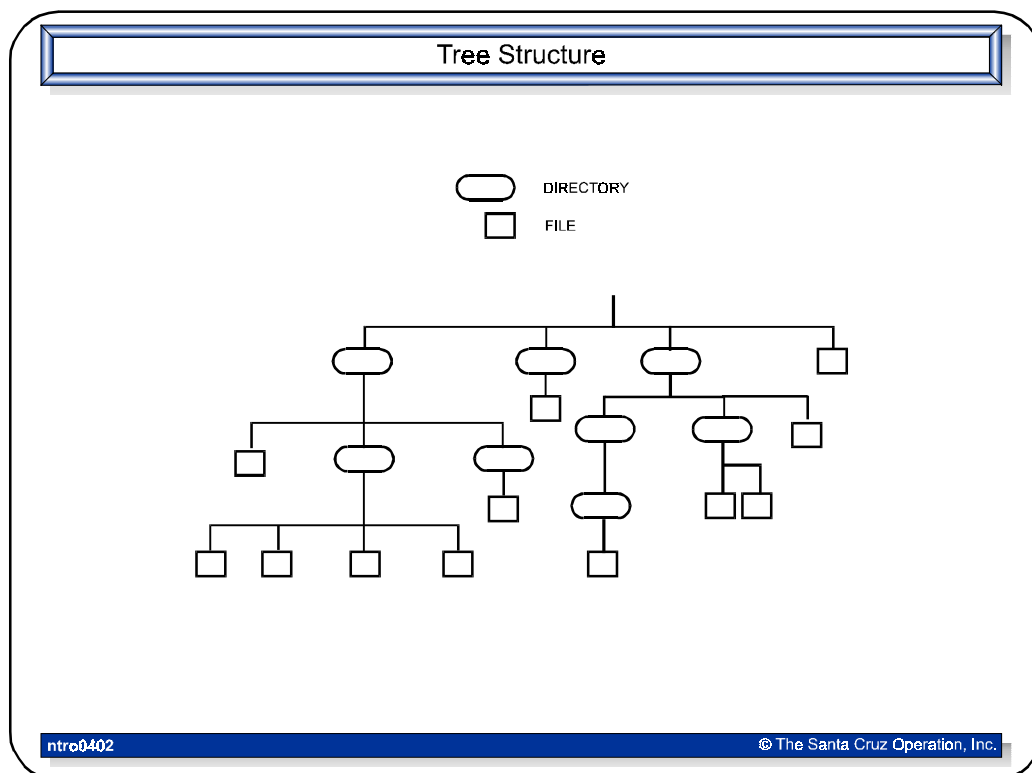
In this lesson you will learn how to use SCO UNIX commands to organize and manipulate files and directories as well as how to manage and maintain your personal *home* directory.

Objectives

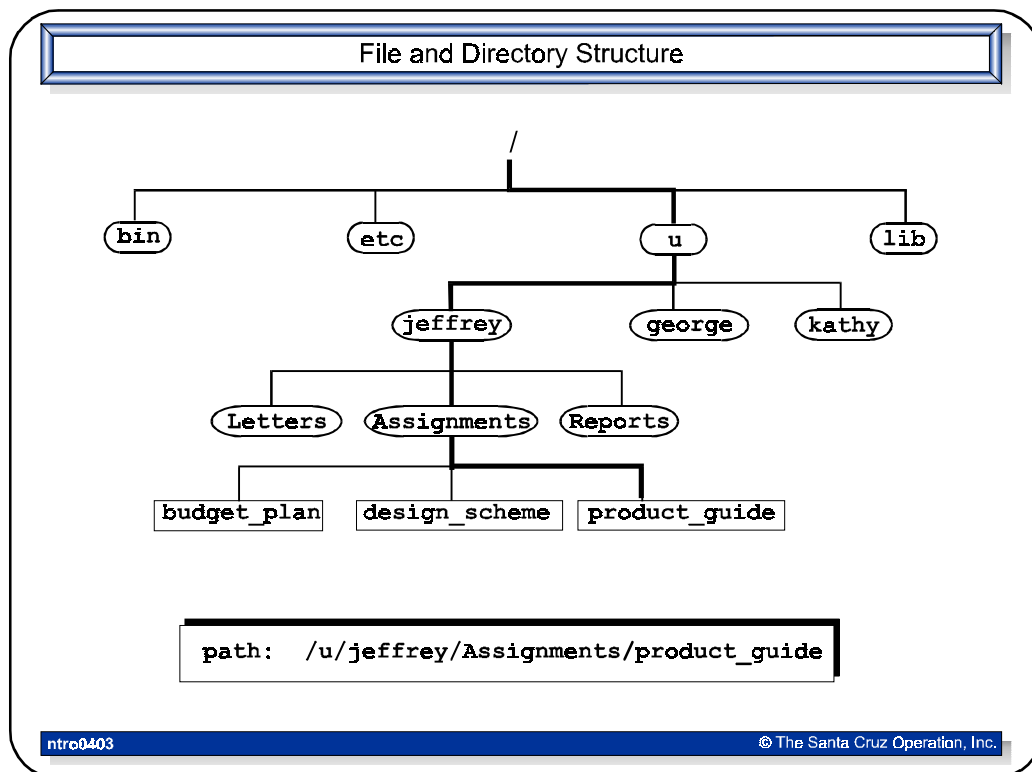
This lesson explains the relationship between files and directories and how to work with them using SCO UNIX commands. By the end of this lesson, you will be able to:

- Describe the hierarchical directory structure using the terms: *root*, *directory*, *subdirectory*, *filename*, and *pathname*
- Identify the different file types supported by SCO UNIX systems
- Navigate around the filesystem using the **cd** command using absolute and relative pathnames
- Manage your home directory by creating, viewing, copying, renaming, and removing files and directories
- Use filename expansion characters to access groups of files and facilitate entering and executing file management commands
- Use appropriate commands for finding and comparing files and directories on the disk

AV 4-2



AV 4-3



4.1 The File and Directory System

The system of files on a disk is structured in a hierarchy of directories and subdirectories, with the top of the hierarchy referred to as the *root* (/) directory. Common terms used to describe the elements of this hierarchy are defined below:

<i>Filesystem</i>	An area of a disk that has been specially formatted to allow the operating system to easily address and access the disk blocks contained within it
<i>File</i>	A named object within a filesystem that stores data. A file can be empty (i.e., contain no user data), but still provide other information for the operating system.
<i>Directory</i>	A special file whose entries list the files contained within it. Directories support the hierarchical structure of the filesystem. Every file on a filesystem is listed in one (or more) directories.
<i>Subdirectory</i>	A directory contained within another directory. The directory that contains a subdirectory is called the parent directory. All directories are subdirectories and have one parent, except for the root directory which acts as its own parent.
<i>Filename</i>	A string of characters that is stored in a directory file entry to identify a file
<i>Pathname</i>	A set of one or more filenames, combined together with slash (/) characters. A pathname specifies a file's location within the hierarchical tree structure (i.e., within the filesystem).

4.1.0.1 Viewing the filesystem

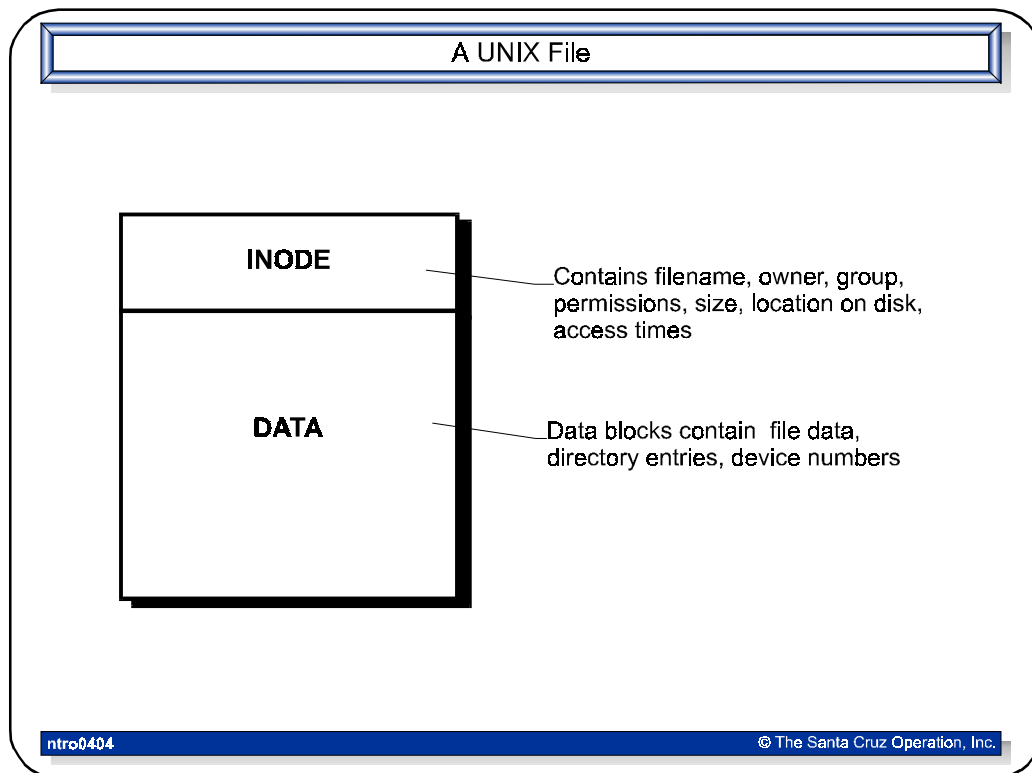
Your view of the filesystem uses a reference point directory; that directory is referred to as your *current working directory*.

- The **ls** command lists the names of files and subdirectories contained within your current working directory by default
- The filenames are listed in ASCII order: names beginning with numbers are first, uppercase letters are next, and lowercase letters are last

Exercise: Viewing the filesystem

Use the **ls** command to list the files in your current directory.

AV 4-4



4.1.1 The structure of a file

A file is the fundamental unit of the SCO UNIX operating system. All software is organized into files.

4.1.1.1 Elements of a file

Whether a file is a program, a document, a database, or a directory, the operating system gives it the same structure:

Inode A record (index node) in the filesystem structure that contains information about that file, such as file permissions, ownership, and size

Data The actual contents of the file, which may be empty or very large in size with its own structure applied to it

4.1.1.2 Filenames

The name of a file is stored within a directory file. UNIX filenames can contain nearly any combination of ASCII characters, up to 255 characters in length (some older filesystem types limited the length to 14 characters).

There are conventions that will make your management of files much easier.

- Keep the filename simple. It should reflect the file's purpose. Filenames rarely need to be longer than 14 characters.
- A filename may contain any ASCII character except for a slash (/) or the null character (ASCII character 0). This is because these are specially interpreted by the kernel for pathnames.
- When naming files, convention allows the use of the underscore (_) and period (.) characters to categorize file types and make the names more readable, but avoid using the following characters, as they have special meanings to system shells:

; | < > ` " ' \$! % & * ? \ () []

Filenames should avoid blank spaces, tabs, or other control characters

- Use common prefixes or extensions for similar files
- SCO UNIX systems are case sensitive and distinguish between upper and lower case characters. A file called *letter* is not the same as a file called *Letter*.
- Filenames beginning with a dot (.) are *hidden* files that are not displayed by default by the **ls** command. Shell wildcards metacharacters likewise do not match these filenames by default.

AV 4-5

File Types	
Ordinary	① r w - r w - - - -
Directory	② r w x r w x - - -
Symbolic link	③ r w x r w x r w x
Device files	④ r w - r w - - - -
	⑤ r w - r w - - - -

ntro0405 © The Santa Cruz Operation, Inc.

4.1.2 Types of files

The SCO UNIX operating system supports several types of files, including:

- Ordinary files
- Directory files
- Special device files
- Symbolic links

4.1.2.1 Ordinary files

Ordinary files (also called regular files) contain a variable length string of bytes. The kernel imposes no structure on the data; it simply presents it to applications as an ordered sequence of bytes. The application organizes and interprets the data itself, generally into one of these categories:

- Text files contain ASCII characters. Examples: letters, reports, and command text files called scripts that are used by shells.
- Data files contain numeric and text data from an application. Examples: spreadsheets, databases, and word processing documents.
- Executable binary programs contain machine instructions and data. Examples are the system commands you've learned so far. Command text files may also be made executable.

4.1.2.2 Directories

A directory is a special file that helps organize the hierarchical tree structure of the filesystem. Directory files contain data about the files and subdirectories located within the directory, including inode numbers and filenames.

- The first entry of every directory represents the directory itself and is given the name “dot” (.)
- The second entry of every directory has the name “dot-dot” (. .) and represents the parent directory
- No two files/directories in a given directory may have the same name

Recall that filenames beginning with a dot (`.`) are hidden files, but can be listed with the `-a` option to the `ls` command.

Exercise: Displaying the contents of a directory

Display the contents of your current directory using the command:

```
ls -ai
```

Note the first two entries listed. They represent the current directory and the parent directory respectively. Are there any other hidden files?

4.1.2.3 Device files

Device files are special files that contain no data beyond what is stored in the file inode. They are used by the system to identify the specific device driver that the kernel will use to communicate with hardware devices.

There are two subcategories of special device files that correspond to the two different kinds of device drivers:

- Character device—the most generic type of device driver that allows I/O transfers of any size, depending on the capabilities of the device itself. Devices that use this interface include terminals, printers, and mice.
- Block device—a device driver that uses the automatic buffering facilities of the kernel buffer cache, where I/O transfers are always 1 Kbyte in size. Devices that use this interface include hard disks, floppy disks, and RAM disks.

An example of a device file is the file that names the terminal channel you are using. The `tty` command displays a terminal device file such as:

```
$ tty  
/dev/tty1a
```

Device files conventionally reside in the `/dev` directory.

4.1.2.4 Symbolic links

Symbolic links are special files that reference another file. The data they store is the path to the file within the filesystem. The kernel automatically accesses this stored path when the symbolic link file is used.

4.1.3 Determining file types

Using the **file** command to display the file type will help you determine the commands that can be used with a file. It is important to avoid using commands intended for text files (such as **cat**, **more**, and **pg**) with files containing data. This may cause the screen to become garbled or locked.

The **file** command takes any number of filenames (or pathnames) as arguments and displays the file type:

```
$ file /etc/motd
```

```
/etc/motd: ascii
```

Exercise: Displaying file types

Using the **file** command, determine the type of the files *bigfile* and *what_am_i* in your home directory.

Use the **file** command with the argument *bin/treed*. What type of file is *treed*?

Try the **file** command with *Poems* and *Miscellaneous*

4.2 Navigating Around the Filesystem

When you log into an SCO UNIX operating system, you are given a process (namely, the shell) with which you issue commands to the system. You are also given an initial working directory called your home directory.

4.2.1 The home directory

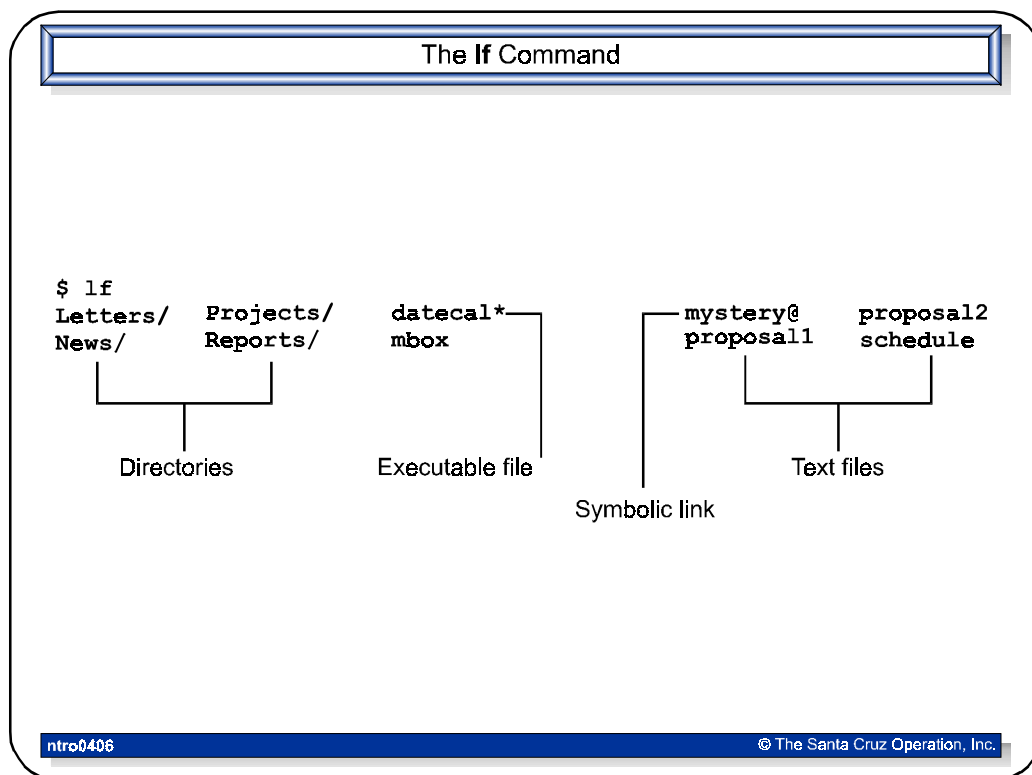
- Your home directory is your current working directory when you log into the system
- Your home directory generally contains subdirectories, data files, and configuration files for your login environment
- Home directories are often located in either the */u* or */usr* directories and are named after your login name (for example, */u/jason*)

4.2.2 Viewing files and directories

The **ls** command displays the contents of a directory file, one file per line.

- By default **ls** displays the current directory
- Specify one or more directory names as command-line arguments to view other directories
- Command-line options modify how files are listed
- Additional directory listing commands are provided which automatically apply the commonly-used options

AV 4-6



4.2.2.1 Filename list commands

Each of the listing commands can take any number of directory or file names as arguments. The syntax is:

```
ls [ filename ... ]
```

where *filename* can be the name of a file or a directory.

Two often-used variants of the **ls** command are **lf** and **l**. The **lf** command lists filenames in columns and distinguishes file types by appending a character next to their names:

- A slash (/) is appended to each directory name
- An asterisk (*) is appended to each executable filename
- An at sign (@) is appended to files that are symbolic links

AV 4-7

The l Command

Disk blocks

```

$ l
total 44
drwx----- 2 jon train 68 Aug 13 10:23 Letters
drwxrwxrwx 2 jon train 41 Jul 23 14:12 News
drwxrwx--- 2 jon train 113 Aug 02 09:15 Reports
lrwxrwxrwx 1 jon train 13 Sep 09 08:10 project -> /Rpts/proj
-rw-r--r-- 1 jon train 2095 Aug 26 11:45 schedule
-rwxr-x--- 1 jon train 762 Jun 02 09:53 timecal

```

Permissions	Number of links	Owner	Group	File size	Date modified	Filename
-------------	--------------------	-------	-------	-----------	---------------	----------

ntro0407
© The Santa Cruz Operation, Inc.

The **l** command displays additional information about files, including:

- file types, file permissions, owner IDs, and group IDs
- the size of each file and the number of links
- the date and time when each file was last modified

Other variants of the list commands include:

lc Lists files in columns sorted down the screen

lf Lists files in columns with markings indicating file types:

 *programs
 @symbolic links
 /directory

lr Lists recursively through all subdirectories

lx Lists files in columns, sorted across the screen

These list commands equivalent to using **ls** with a combination of options.

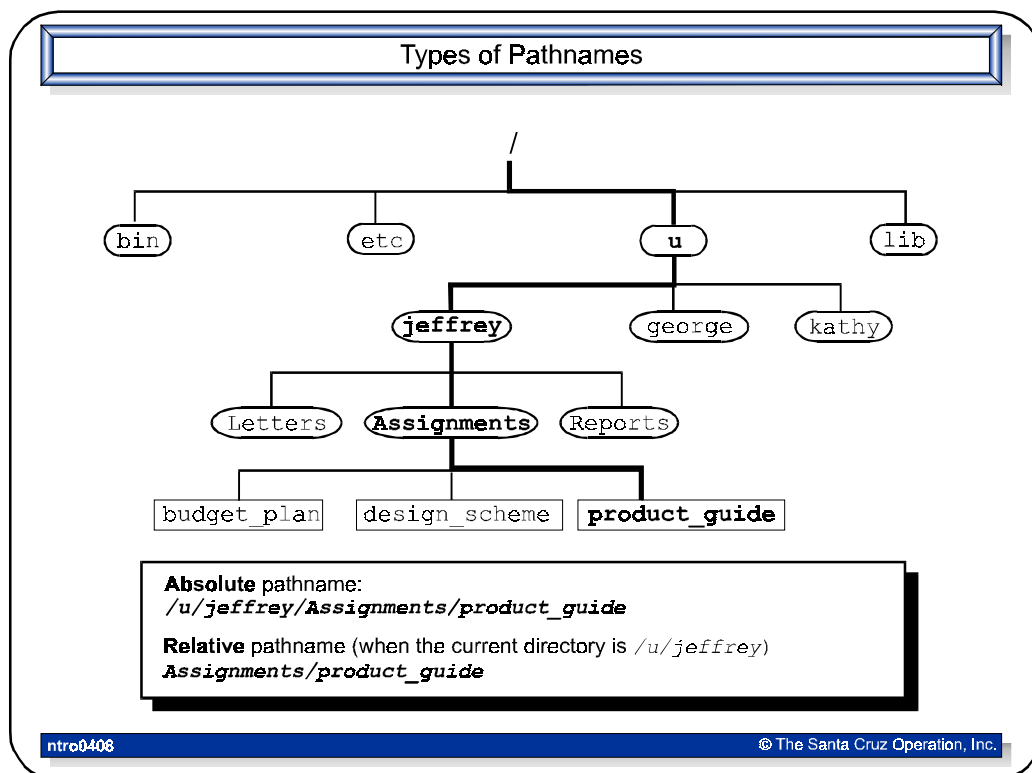
Exercise: Listing files and directories

From your home directory, run each of the list commands and compare the results.

Use the **lf** command to list a subdirectory in your current directory. Use the **l** command to list an ordinary file. What happens if you attempt to list a file that doesn't exist?

The family of list commands you have just seen shares many of the same options. Run the **lc**, **lf** and **l** commands with the **-a** option to see the hidden files.

AV 4-8



4.2.3 Pathnames

Until now, you have looked at files only within your home directory. You can use pathnames to access files and directories anywhere within the hierarchical filesystem.

- A pathname describes the route to any file in the filesystem
- There are two kinds of pathnames: *absolute* and *relative*
- When specifying a pathname for a command, you may specify either kind of path, no matter how long or complex

4.2.3.1 Absolute pathnames

Every file on a UNIX operating system has a unique absolute pathname which is made by concatenating all the directory names down the hierarchy, starting from the root directory to the file. These directory names are separated by the slash character (/). For example:

/usr/lib/mail/charlie

- An absolute pathname always begins with (/) which represents the root directory
- Using absolute pathnames is often the easiest way to access files that are above your current working directory
- Another term for *absolute* pathname is *full* pathname
- The **pwd** command displays the absolute pathname of your current working directory:

```
$ pwd
/u/charlie
```

Exercise: Absolute pathnames

Use the **pwd** command to display the absolute pathname of your current working directory.

4.2.3.2 Relative pathnames

A relative pathname specifies a file with a path that is relative to your current working directory

- To access files in your current working directory or any of its subdirectories, you can use relative pathnames. For example, if you are in the directory `/u/matthew`, to list a file in the directory `/u/matthew/Personnel/Letters`, use:

```
ls -l Personnel/Letters/file
```

- A relative pathname does not begin with a slash (/)
- You can use the special directory names dot (.) and dot-dot (..) in relative pathnames to access files within and above your current working directory. For example, if you are in the directory `/usr/donna` and you want to list the `/usr/lib/mail` directory, enter the command:

```
ls ../lib/mail
```

Recall that dot-dot is in every directory and is a reference to that directory's parent. In the above example, the parent directory of `/usr/donna` is `/usr`.

- References to parent directories using `../` can be made sequentially all the way up to the root directory, so every file on the system can be named with a relative pathname

4.2.3.3 Absolute vs. relative pathnames

To decide whether to use relative or absolute pathnames, evaluate which method specifies fewer directories (which often results in less typing!).

- For example, if you are in the */etc/conf/cf.d* directory, to access the system password file:
 - Use the absolute path */etc/passwd* or the relative path *../../passwd*
 - */etc/passwd* specifies fewer directories (2) than *../../passwd* (3) and is more efficient
- But if you are in the */usr/spool* directory, to access **root**'s mailbox file:
 - Use the absolute path */usr/spool/mail/root* or the relative path *mail/root*
 - *mail/root* specifies fewer directories (2) than */usr/spool/mail/root* (4) and is more efficient

If you do not know your current directory in relation to other directories, you can still use an absolute pathname.

4.2.4 Changing directories

If you are going to access several files in a different directory, such as one of your subdirectories, it is often easier to change your current working directory to that directory and then access the files from there. Note that you can change to directories that are outside of your personal subdirectories.

- To change your current working directory, use the **cd** command with the pathname (relative or absolute) of a destination directory as its only argument. For example, to change to the subdirectory named *Projects*, enter:

cd Projects

- To move two directories above your current directory, use dot-dot (**..**):

cd ../..

- To return to your home directory from anywhere on the system, enter the **cd** command without any argument:

cd

- If you give the **cd** command an ordinary filename, a directory that does not exist, or one that you do not have permission to use, an error message is displayed.

```
$ cd letter
ksh: letter: not a directory
$ cd ../../baddir
ksh: ../../baddir: not found
```

If you misspell a directory name, the shell's spelling aid might be able to assist you. If you attempt to change to a directory and misspell the name, The shell guesses what you mean and asks you to confirm the guess. For example:

```
/cd /etc/default
cd /etc/default?
```

Pressing  tells the shell to change to the directory it displayed. If you do not want to change to that directory, type **n** or **N** and press .

Exercise: Changing directories

Practice navigating around the filesystem using the **cd** and **ls** commands.

You should be able to go anywhere in the filesystem for which you have permission.

Try changing to the directory *Hidden* in your home directory.

AV 4-9

Displaying Text Files

cat	Displays entire file to the screen
more	Displays a text file one page at a time
pg	Displays a text file one page at a time
head	Displays the first ten lines of a file
tail	Displays the last ten lines of a file

ntro0409

© The Santa Cruz Operation, Inc.

4.3 Viewing Ordinary Files

So far, we have viewed the contents of directory files using the list commands, but you have not yet viewed the contents of any ordinary files. SCO UNIX systems have several commands for viewing all or parts of ordinary files.

4.3.1 Viewing text files

Text files are files composed of printable characters and control characters, such as tabs and newlines. Several commands are available for viewing text files; these commands should not be used on directories or binary files:

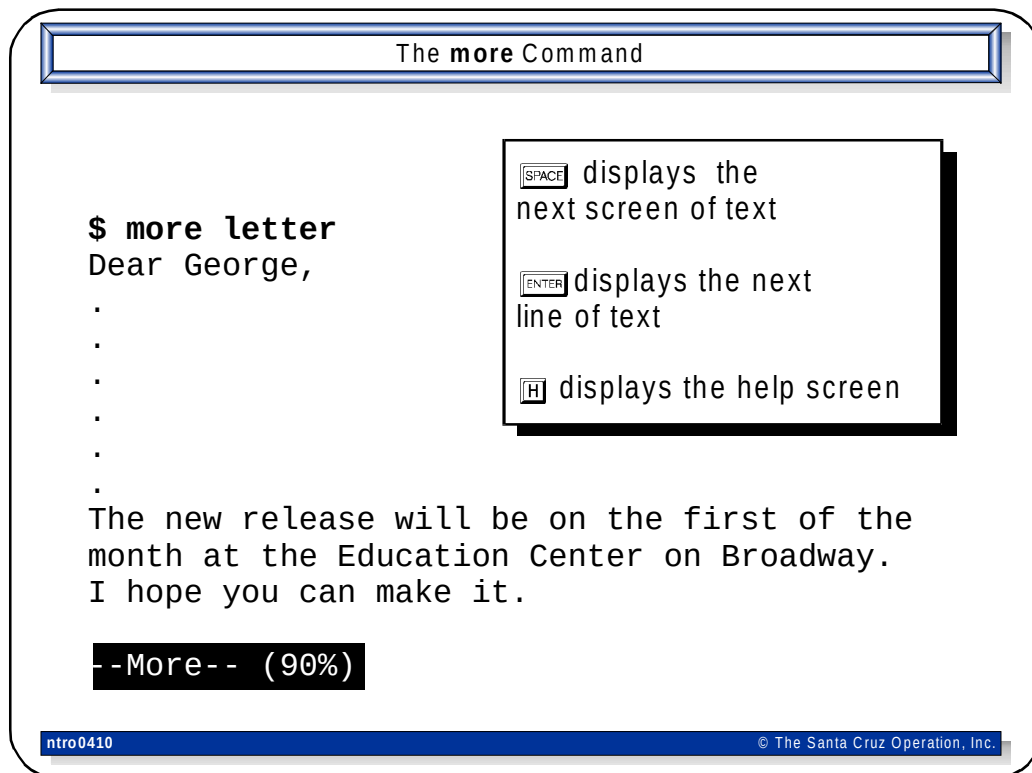
4.3.1.1 The cat command

The syntax for the **cat** (concatenate) command is:

```
cat filename [ filename ... ]
```

- The contents of each file are displayed on the standard output (usually the screen). The screen scrolls if the contents contain more lines than your screen can display.
- To control scrolling, press **CTRL s** to stop scrolling and **CTRL q** to resume
- Press the **DEL** (interrupt) key to terminate the command and return to the shell prompt

AV 4-10



4.3.1.2 The head and tail commands

To display a portion of a file, use **head** or **tail**:

```
head [ -count ] filename [ filename ... ]
tail [ -count ] filename [ filename ... ]
tail [ +n ] filename [ filename ... ]
```

- The **head** command prints the first *count* lines of the specified file(s) on the screen (default=10)
- The **tail** command prints the last *count* lines of the specified file(s) on the screen (default=10) or the end of the file, beginning at line number *n*

4.3.1.3 The more command

The **more** command displays files one page at a time. The syntax is:

```
more filename [ filename ... ]
```

- A prompt appears at the bottom of each screen showing the percentage of the file shown so far:

```
-More- (xx%)
```

- You can respond to the prompt in several different ways:
 - Press  to display the next screenful of text
 - Press  to display only the next line of text
 - Press slash (/) followed by a pattern to search for the next matching pattern
 - Press **h** to display a help screen
 - Press **b** to display the previous screen
 - Press **q** to quit

4.3.1.4 The pg command

The **pg** command displays the contents of files one page at a time, allowing you to go backwards as well as forwards in the file. The syntax is:

`pg filename [ filename ... ]`

- The following responses may be made to the colon (:) prompt:
 - Press  to display the next screen
 - Type **+n** to move forward *n* screens
 - Type **-n** to move backward *n* screens
 - Type **h** to display a help screen

Exercise: Viewing text files

Use **cat**, **head**, **tail**, **more**, and **pg** to view the text files in your home directory.

4.3.2 Viewing data files

Data files are ordinary files that contain characters that are not printable on a terminal or console screen. Data files are generally viewed by programs that know how to handle the binary data in the file. You can use the following commands to view a data file on a terminal screen:

- The **hd** command displays files in hexadecimal format
- The **od** command displays files in octal format

Both commands accept one or more filenames as arguments and support options that vary the format of the output.

Exercise: Viewing data files

What type of file is `/etc/mnttab`? Use the **hd** and **od** commands to view the `/etc/mnttab`. Compare the output of these two commands.

4.4 Organizing Files

As you work within your home directory, you will often need to copy, move, rename, and create files and subdirectories.

4.4.1 Creating directories

- Instead of storing all your files in your home directory, you can create subdirectories to organize your files
- Use consistent naming conventions to differentiate between files and directories. For example, begin directory names with an upper case letter so they will appear first in your directory listings.
- Subdirectories should be used to contain similar files. For example, you could create a subdirectory for all your database files, another subdirectory for your spreadsheet files, and another for your word processing documents.
- Since directories are files, they follow the same naming conventions as ordinary files and are uniquely specified by their full pathname
- To create a directory, use the **mkdir** command:

```
mkdir Dirname [ Dirname ... ]
```

- The **-p** option instructs **mkdir** to create any non-existing parent directories that are required to create the specified *Dirname*:

```
mkdir -p Dirname/Dirname ...
```

Exercise: Creating directories

Use **mkdir** to create a subdirectory called *MyFiles* in your home directory. Use the **-p** option to create a subdirectory named *Poems/Whitman/Anthology*.

AV 4-11

Examples: **cp** and **mv**

```
$ command  source  destination
```

```
$ cp apples oranges peaches Foods
```

Copies *apples*, *oranges*, and *peaches* to the directory *Foods*.

```
$ cp oranges Foods/citrus
```

Copies the file *oranges* to the directory *Foods* and names it *citrus*.

```
$ mv fruit apples
```

Renames the file *fruit* to *apples*.

Note: The directory *Foods* must exist before you attempt the above examples.

4.4.2 Copying files

Use **cp** for making backup copies of files or for making personal copies of another user's files:

```
cp sourcefile destinationfile  
cp sourcefile1 [ sourcefile2 ... ] destinationfile
```

- **cp** duplicates a file, leaving the original file untouched
- WARNING! If you copy a file to a destination filename that already exists, the contents of the destination file will be destroyed.
- You can use the **cp** command to copy a list of files to a destination directory
- In the first version of the syntax, *sourcefile* is copied to *destinationfile*
 - If *destinationfile* is the path to a directory file, then *sourcefile* is copied into that directory with the same filename it had before
 - If *destinationfile* is the path to a non-directory file, then *sourcefile* is copied to that pathname, destroying any file that may already exist with the same filename
- In the second version of the syntax, all *sourcefiles* are copied to *destinationfile*, which must be a directory file. All files copied there retain the same filename they had before.
- All arguments may use either absolute or relative pathnames; as always, the dot (.) or dot-dot (..) notations may be used
- Any directory specified in the *destinationfile* argument must already exist; **cp** will not create needed directories
- You can use the **cp** command to copy a file to your current working directory. For example:

```
cp ../mary/Homework/assign1 .
```

- You might not have permission to copy files into and out of certain directories, in which case you will get an error message
- To make a copy of a directory and all the files in that directory, use the **copy** command, which accepts many different options. For example:

```
copy -mor directory.old directory.new
```

Refer to the manual pages for more information on using **copy**.

Exercise: Copying files and directories

Copy the file **bigfile** to **bigfile.bak**. Then copy the file **empty** to **bigfile**. View the contents of **bigfile**. Do a long directory listing of it. How big is **bigfile**?

Copy the files **/etc/motd** and **/etc/group** to the directory **MyFiles** that you made in the previous exercise.

Copy the **MyFiles** directory to a new directory called **EmptyFiles**, using the **copy** command.

4.4.3 Moving and renaming files

The **mv** command can move a source file to a destination file, or it can move a list of files to a destination directory. The syntax follows the same guidelines as the **cp** command:

```
mv sourcefile destinationfile
mv sourcefile1 [ sourcefile2 ... ] destinationfile
```

- Use the **mv** command to move files from one directory to another or to rename files
- WARNING! If you move a file to a destination file that already exists, the contents of the destination file are destroyed.
- In the first version of the syntax, *sourcefile* is moved to *destinationfile*
 - If *destinationfile* is the path to a directory file, then *sourcefile* is moved into that directory with the same filename it had before
 - If *destinationfile* is the path to a non-directory file, then *sourcefile* is moved to that pathname, destroying any file that may already exist with the same filename
- In the second version of the syntax, all *sourcefiles* are moved to *destinationfile*, which must be a directory file. All files moved there retain the same filename they had before.

- If the source and destination files are in the same directory, the **mv** command performs a rename operation. For example:

```
mv oldname newname
```

Exercise: Moving files and directories

Restore *bigfile* by renaming *bigfile.bak* to *bigfile*. Verify its contents.

Move *bigfile* to the home directory of one of the other students and rename it to *largefile* in the same step. What happened?

Make a new directory called *Save* and move the *MyFiles* directory into the *Save* directory. Verify the results with a recursive directory listing of the *Save* directory.

AV 4-12

Linking Files

Hard links

```
$ ln greeting blink
```

Symbolic links

```
$ ln -s greeting slink
```



```
$ ls -l greeting ?link
```



```
-rw-r--r-- 2 jim group 111 Apr 6 greeting
-rw-r--r-- 2 jim group 111 Apr 6 blink
lrwxrwxrwx 1 jim group   8 Apr 6 slink -> greeting
```

ntro0412© The Santa Cruz Operation, Inc.

4.4.4 Linking files

SCO UNIX systems provide a way for a given file to have more than one name (i.e., directory entry). This procedure is called linking.

The **ln** command creates a link to a file. Its syntax is the same as **cp** and **mv**:

```
ln sourcefile destinationfile  
ln sourcefile1 [ sourcefile2 ... ] destinationfile
```

- Linked files may have different names, but be in the same directory (for example, the command files */bin/l*s and */bin/l*c). Reducing duplication saves disk space.
- Linked files may have the same filename, but exist in different directories. This allows you to change data in one place and have that change show up in files in many different directories.
- In the first version of the syntax, *sourcefile* is linked to *destinationfile*
 - If *destinationfile* is the path to a directory file, then *sourcefile* is linked into that directory with the same filename it had before
 - If *destinationfile* is the path to a non-directory file, then *sourcefile* is linked to that pathname, destroying any file that may already exist with the same filename
- In the second version of the syntax, all *sourcefiles* are linked to *destinationfile*, which must be a directory file. All files linked there retain the same filename they had before.

4.4.4.1 Hard links

The **ln** command creates a hard link by default. Creating a hard link causes an existing file's inode number to be used in multiple directory file entries

- The number of hard links a file has can be seen in the second field of a long directory listing. Files with no extra links have a link count=1.
- The **ln** command increments the link count; the **rm** command decrements it. A file is not physically removed from a filesystem until the link count=0.
- There are limitations to what a user may hard link:
 - You cannot hard link directory files
 - You cannot hard link files between filesystems

4.4.4.2 Symbolic links

Symbolic links, sometimes called soft links or symlinks, associate (link) a pathname to a file. They are a special type of file, like directory files and device files, whose data blocks contain the path of some other file within the hierarchical file structure.

- When a symbolic link file is used in a pathname (absolute or relative), the kernel reads the data in this file and then goes to the pathname that it indicated in its data, as if you had specified that path yourself to begin with.

- Symbolic links are made using the **-s** option to the **ln** command:

```
ln -s source destination
```

The path specified as *source* is stored in the file named *destination*. Unlike hard links, *destination* is a different file with a different inode number.

- Symbolic links do not have the limitations of hard links—they can be made to directories as well as to files in other filesystems
- It is often best to specify *source* with a full pathname so that the symbolic link file can be used when in any working directory. When *source* uses a relative path, it may not work if your current directory is not the same as when the symbolic link file was created.
- Symbolic links maintain a distinction between the link and the original file or directory:
 - Removing the original file or directory removes the data, but does not remove the link. As long as the original file is re-created with the same name, the symbolic link will continue to point to its new data.
 - A symbolic link shows up in long listings as a special file type, designated by the letter **l** as the first character of the long directory listing line
 - The size of the symbolic link is the number of bytes in the pathname of the file to which it is linked
 - The name of the symbolic link shows a pointer (**->**) to the original file or directory when viewed with the **ls -l** command. (Note: in SCO OpenServer Release 5, the **l** command displays symbolic links with the attributes of the file to which they are linked.)

```
lrwxrwxrwx ... 14 Jan 31 20:16 /etc/motd -> /original_file
```


Exercise: File linking

Create a hard link from the file *greeting* to *hlink*.

Create a symbolic link from the file *greeting* to *slink*.

Do a long directory listing of all three files:

```
ls -l greeting hlink slink
```

Note how the different links are identified.

AV 4-13

Filename Expansion

*	Matches any pattern of zero or more characters
?	Matches any single character
[pattern]	Matches any one character in the pattern set
[!pattern]	Matches any one character not in the pattern set

ntro0413© The Santa Cruz Operation, Inc.

AV 4-14

Using Filename Expansion Characters


```
$ ls
Letters/  Projects/  mysteryA  mysteryC  proposal2
News/     Reports/   mysteryB  proposal1  timecal*
```

\$ cat prop*
Displays *proposal1* and *proposal2*.

\$ more mystery?
Displays the contents of all three *mystery* files on the screen one page at a time.

\$ head mystery[AC]
Displays the first 10 lines of the files *mysteryA* and *mysteryC*.

ntro0414© The Santa Cruz Operation, Inc.

4.4.5 Filename expansion characters (wildcards)

You may want to access large numbers of files at once, particularly when reorganizing directories or backing up files. The shell provides several special characters (metacharacters) that offer a “wildcard” notation for specifying multiple files in one string. These filename expansion characters:

- Allow the shell to search for filenames that match a particular pattern
- Are used to abbreviate file or directory lists used as arguments to commands
- Can be used to access long filenames with a shorthand notation
- Can be used on any shell command line
- Are expanded by the shell to one or more filenames before the command begins execution
- Remain unexpanded on a command line if no filenames match the indicated pattern
- Can be dangerous, since they allow commands to operate on a large number of files

4.4.5.1 Asterisk (*)

The filename expansion character, asterisk (*), matches any string of zero or more characters.

- The pattern `le*` matches files such as *letter*, *lease*, and *le*
- The pattern `*` matches all filenames in the current directory, except the hidden files that start with a dot (.)
- The pattern `.*` matches these hidden files only

4.4.5.2 Question mark (?)

The filename expansion character, question mark (?), matches exactly one character.

- More than one `?` can be specified in a pattern
- The pattern `file?` matches the files *file1*, *file2*, and *fileA*, but not *file10* or *file*

- The pattern `name.???` matches the files *name.abc*, *name.xyz*, but not *name.ab*, *name.abcd*, or *name*

4.4.5.3 Brackets ([])

The bracket characters ([]) match a single character from the list of characters enclosed within the brackets.

- The pattern `letter[123]` matches *letter1* and *letter2*, but not *letter12*
- Use a dash (-) to specify a range of characters rather than listing them all. For example, `chapter[0-9]` is shorthand for `chapter[0123456789]` and matches the files *chapter0*, *chapter5*, and *chapter9*, but not *chapters* or *chapter12*.
 - The range must be an ascending one. `[A-Z]` is valid; `[Z-A]` is not.
 - Multiple ranges are permitted. For example, `[A-Za-z]` matches any letter character, upper or lower case.
- An exclamation point (!) as the first character within a set of brackets excludes what follows; i.e., matches all characters except those enclosed in the brackets
- All these metacharacters can be used in combination: `[!A-Z]*.?` matches all names that do not start with an upper case letter, but do have a period (.) in the second to the last position

Exercise: Using filename expansion characters

Enter the command: **ls *** and compare the output with that of **ls**.
Use one argument to **ls** to list all files beginning with a vowel.

AV 4-15

Removing Files: rm

```
$ rm apples
$
```

This removes the file *apples* without any further input from the user.

```
$ rm -i apples
apples: ?
```

The *i* option stands for interactive. The computer asks if you really want to remove the file *apples*. Type *y*  to remove the file.

intro0415 © The Santa Cruz Operation, Inc.

4.4.6 Removing files and directories

When files become outdated, or disk space becomes limited, you may need to remove files and directories from the system.

WARNING! Unless the directories in which you are working were set up to support the SCO OpenServer Release 5 **undelete** command, the actions of the commands presented here are irreversible, and should be invoked with extra caution.

- Use **rm** to remove files that are no longer needed. The syntax is:

```
rm file [ file ... ]
```

- It is a good practice to review the contents of a file before removing it
- The **rm** command can be run interactively using the **-i** option. This is a good idea when removing multiple files using filename expansion characters. The system asks for confirmation for each file before removing that file.

```
$ rm -i *  
file1 ?
```

You must press **y** or **Y** to remove each file; pressing by itself or with any other character, leaves the file untouched.

- The **rm** command will not remove directories unless the recursive option (**-r**) is used. A recursive removal deletes files and directories. When a directory is removed, all the files and subdirectories beneath the directory are removed.

```
rm -r *
```

This command removes everything in the current directory and below except for any hidden files in the current directory. WARNING! Use caution when using **rm** with the **-r** option.

- Another way to remove a directory is with the **rmdir** command, which removes only empty directories. All files within the directory must be removed with **rm**.

```
rmdir empty_directory [ empty_directory ... ]
```


Exercise: Removing files and directories

Remove the *EmptyFiles* directory with the following commands:

```
rm -i EmptyFiles/*
```

```
rmdir EmptyFiles
```

Remove the *Save* directory using the **-r** option, but do not use any filename expansion characters.

AV 4-16

Finding Files

```
$ find . -name chapter1 -print
```

Search for files named *chapter1* in the current directory (.) and all directories below it, and send the pathname of any such files to the screen.

```
$ find .. -name 'sonnet*' -print
```

Searches for files whose names begin with 'sonnet' in the parent directory (..) and all directories below it. Quotes must be used around the filename when shell metacharacters are used.

intro0416

© The Santa Cruz Operation, Inc.

4.5 Working with Files

SCO UNIX systems provide many commands that help you work with your files.

4.5.1 Finding files and directories

The **find** command is used to find the location of files and directories. You specify the name of the file you want to find and from which directory you want the system to start its search:

```
find pathname -name filename
```

- The **find** command will look for the file or directory specified by *filename* starting from the directory specified by *pathname*
- The **-name** option is a literal string and must be typed exactly as shown. The **-name** option specifies the file attribute that is being searched. Other attributes can also be searched as described in the **find** manual page.
- The full pathname of every file with a name matching that specified by *filename* and occurring somewhere beneath the directory *pathname* will be printed to the standard output (normally, the screen)
- For files and directories you not have permission to search, **find** lists the filename with an error message

AV 4-17

Finding Strings

```
$ grep henry /etc/passwd
```

Search the file */etc/password* for all lines containing the name *henry*.

```
$ grep 'my beloved' Poems/*/*
```

Search for the string *my beloved* in all files that are in subdirectories of the *Poems* directory.

intro0417

© The Santa Cruz Operation, Inc.

4.5.2 Finding text within files

The **grep** command is useful for finding words or phrases in text files. It displays to the standard output any line that contains a given string pattern. The syntax is:

```
grep [options] pattern file [ file ... ]
```

- The *pattern* is a *regular expression*, which uses special metacharacters
- The *pattern* in the file must be wholly contained on one line; it cannot begin on one line and complete on the next
- Filename expansion characters may be used to specify the files through which to search. Be careful so that **grep** searches text files only.
- The *pattern* can be only one argument, so use quotes (single quotes (') are usually adequate) if you use shell metacharacters in your *pattern*
- By default, **grep** is case sensitive when it searches for *pattern*. Use the **-i** option to make the search case insensitive.

Exercise: Finding data

Find all occurrences of the file *letter* under the */u* directory.

How many times does the word *Mother* occur in the file *letter*?

How many files in your home directory contain the word *better*?

AV 4-18

Commands for Comparing Files

cmp	Compares two files and reports the first character that differs.
diff	Compares two text files and displays all lines that differ between the two files.
sum	Computes a unique checksum value for a file.

ntro0418 © The Santa Cruz Operation, Inc.

AV 4-19

Comparing Files

```
$ diff greetings /etc/motd
2c2
< Welcome to UNIX
---
> Welcome to SCO UNIX System V
$
$ cmp /usr/bin/more /usr/bin/pg
/usr/bin/more /usr/bin/pg differ: char 25, line 1
$
$ sum /bin/mv
58109 60 /bin/mv
```

ntro0419 © The Santa Cruz Operation, Inc.

4.5.3 Comparing data between files

It is sometimes useful to know how similar two different versions of a file are. SCO UNIX systems provide several commands for comparing files.

4.5.3.1 Comparing text files

The **diff** command compares two text files and displays the differences between them:

```
$ diff file1 file2
n1cn2
< Line from file1
---
> Line from file2
```

If the two files are identical, no output is displayed by the **diff** command. If the files differ, the differences are displayed in segments as shown above.

- The first line of output gives the line numbers that differ between the two files. The line numbers of *file1* are on the left side of the C (*n1*), the line numbers of *file2* are on the right (*n2*).
- Next are the text of the lines that differed, separated from each other by a line beginning with three dashes. Lines beginning with the less-than symbol (<) belong to *file1*, those with the greater-than symbol (>), to *file2*.

4.5.3.2 Comparing data files

The **cmp** command compares any two ordinary files whether they contain text or data:

```
$ cmp file1 file2
file1 file2 differ: char NN, line N
```

- Since binary data cannot be displayed to the screen, **cmp** simply reports at what byte offset the two files start to differ. (Remember that all ordinary files are just strings of bytes to the computer.)
- If the files are identical, no output is displayed

Remember, the **file** command is useful for determining whether a file contains data or text.

4.5.3.3 The **sum** command

The **sum** command can be used to validate that two files are identical by generating a checksum value that is unique to the contents of that file:

```
sum file [ file ... ]
```

Exercise: Comparing files

Use the **diff** command to compare the *greeting* file in your home directory with the file */etc/motd*.

Use the **cmp** command to compare the files *\$HOME/bigfile* and *\$HOME/Miscellaneous/manpage*. Now compare the files */etc/motd* and *\$HOME/greeting*.

Use the **sum** command to get checksum values for each file.

4.6 Lesson Summary

The SCO UNIX operating system stores information in a hierarchical system of files that is organized by directories into a structure that resembles an inverted tree. The system provides numerous commands for accessing these program, text, and data files. As a user, your data may reside in your personal home directory. Various commands and conventions enable you to keep this information accessible, organized, and protected.

4.6.1 Review questions

1. What is the relationship between a file and a directory?
2. Name four different types of UNIX files.
3. How can you determine whether a file contains text or data?
4. Which commands are available for accessing the contents of directories? Of text files? Of data files?
5. What is the difference between an absolute and relative pathname?
6. What is the difference between the **cp**, **ln**, and **mv** commands?
7. What are the benefits of using filename expansion characters? The disadvantages?

8. How would you use **find** with the dot (.) or dot-dot (..) notation to search for a file that is somewhere under your current directory? To find a file that is somewhere under your parent directory?

4.7 Lab: Using the Filesystem

4.7.1 Lab 1: Viewing the filesystem

The goal of this lab is to become proficient with system commands for viewing the directories and different file types that make up a UNIX filesystem.

Overview

You will work with the sample files in your home directory to demonstrate your recognition of useful file naming conventions and your ability to identify and view different file types.

Procedure

Be sure you are in your home directory to start this lab.

1. Display a listing of the files in your home directory. Are they in any special order?
2. Use **lc -a** to display the contents of directory *Lab3.1.1*. Which files do not follow the UNIX file naming conventions?
3. Use **ls** to display the files in the directory *Lab3.1.2*. What makes these filenames useful? Why are some not so useful?
4. Use **If** to determine what kind of files you have in your home directory. Do all directories begin with an upper case letter?
5. Use the **file** command to analyze the following files: *Poems*, *proposal2*, *timecal*, and *empty*.
6. Display the contents of the *greeting* file on your screen.
7. Display the contents of the *letter* file on your screen. Is it more appropriate to use **cat** or **more**?
8. List the filenames stored under the root (/) directory. Are these files or subdirectories, or both?
9. Determine the absolute pathname of your home directory. Use this pathname as an argument to the **If** command. What are two other ways of getting the same listing?

10. List the contents of your *Poems* directory using a relative pathname. Do the same thing using an absolute pathname (that is, beginning with a slash (/)).
11. For every directory stored under *Poems*, display a listing of filenames stored there. Is it easier to use an absolute or a relative pathname? Which do you prefer?
12. Display the contents of the *jerusalem* file, stored under the *Blake* directory, which is found under *Poems*. Try displaying it using a relative pathname and then using an absolute pathname.
13. Use a single **more** or **pg** command to browse each of the files stored under the *Yeats* subdirectory. Which command did you use? Why? How can you tell where one file ends and the next begins?
14. Use the **head** and **tail** commands to look at the top and bottom ten lines of *bigfile*.
15. Use the **ls** command to see what is stored in the */bin* directory. Do you recognize any of the filenames? What kind of files are these?
16. Display the contents of the file *what_am_i*. Which command did you choose?

4.7.2 Lab 2: Managing files and directories

The goal of this lab is to learn to use filename expansion characters with the SCO UNIX commands for creating, copying, moving, linking, and removing files, as well as to work with commands for finding and comparing files.

Overview

You will be using the **cp**, **mv**, **rm**, **ln**, **mkdir**, and **rmdir** commands to manage the files and directories in your home directory. You will also use the **find** and **grep** commands to look for files and data on the system.

Procedure

1. Verify that you are in your home directory using the **pwd** command.
2. Copy the files *proposal1* and *proposal2* into the files *prop1.bak* and *prop2.bak* respectively. Check the results.
3. Change the names of the file *proposal1* and *proposal2* to *prop1* and *prop2* respectively. Check the results.

4. Remove the file *bigfile* and the file *empty*. Check your results.
5. Use a filename expansion character to remove all the files ending in *.bak*. Perform this operation with one command. Try to do it interactively.
6. Move the file *timecal* to your *bin* directory, giving it the name *datecal*.
7. Create a subdirectory called *Class* in your home directory. Verify it is there. Can you make a directory with a one-character name? Try it. Can you create a directory with a name of / (the slash character) or . (the period character)?
8. Copy the file */etc/motd* into the *Class* directory. Copy the file *mystery* from the *Miscellaneous* directory into the *Class* directory. Check to make sure the files have been placed in *Class*.
9. Change your working directory to the *Class* directory. Verify that you have moved using the **pwd** command.
10. Copy the file *twister* from the *Poems* directory into your current directory, keeping the same name. Repeat this operation. How many copies of the file *twister* are there in your current directory now? Explain what has happened.
11. Move to the directory *Poems*. Each of the files in the subdirectory *Shakespeare* is a sonnet. Using shell metacharacters, display sonnets 1 through 4 on your screen.
12. Using a shell metacharacter, display sonnets 5 and 7 on your terminal screen.
13. Change your working directory back to your home directory. Is there more than one way to do this? Verify that you have returned to your home directory.
14. Remove the directory *Class*. You may have to remove some files first. Use filename expansion characters if possible. Remove any one-character directory you may have created.
15. Find how many directories are named *bin* starting from your parent directory. (Hint: don't forget the **-name** and **-print** options.) Were there any error messages reported by the **find** command?
16. Find how many files in your home directory contain the string *other*. Did **grep** find words other than just *other*? How can you make it ignore words containing the string *other* and only find the word *other*?
17. Run the command **tryme**. Create a hard link from the file *tryme* in your *bin* directory to the name *hither* in your current directory. Create a symbolic link from *bin/tryme* to *thither*. Use the **l** command to verify the creation of the two links.

18. Run the new commands: **hither** and **thither**. Then remove the file *bin/tryme*. Now attempt to run **hither** and **thither**. Explain the difference. Copy *hither* to *./bin/tryme* and rerun **thither**. Do a long directory listing of the two links. Are things as they were before? Why not?

Relevant Commands and Files

The following commands and files were presented in this lesson.

Relevant commands

cat	Concatenates and displays files
cd	Changes the working directory
chgrp	Changes the group ID of a file
chmod	Changes file or directory access permissions
chown	Changes the owner ID of a file
cmp	Compares two files
cp, copy	Copies files
diff	Compares two text files
file	Determines the file type
find	Finds files and directories
grep	Searches files for a string pattern
hd	Displays files in hexadecimal format
l	Lists information about directory contents
lc, lf, ls, lx	Lists directory contents
lr	Recursively lists directory contents
ln	Creates a hard or symbolic link to a file
mkdir	Creates a new directory

more	Displays a file one screen at a time
mv	Moves or renames directories or files
od	Displays files in octal format
pg	Displays a file one screen at a time
pwd	Prints the current working directory name
rm	Removes files
rmdir	Removes directories
sum	Calculates a checksum for a file
tty	Displays the name of the terminal

Relevant files

.	The current working directory
..	The parent directory
/	The root directory
/bin/lf	Executable file containing the lf command
/dev	Directory containing device files
/u, /home, or /usr	Directory containing users' home directories

4.8 Answers to Exercises, Labs, and Review Questions

This section provides suggested answers to exercises, review questions, and lab exercises.

Exercise: Viewing the filesystem

The **ls** command shows an alphabetized vertical listing of all files in the directory. The **treed** command displays the files in a horizontal ‘tree’ structure which shows the hierarchical relationship of the files and directories.

Exercise: Displaying the contents of a directory

There should be the following hidden files in your home directory: *.profile*, *.kshrc*, *.sh_history*, and *.lastlogin*. There may also be several others, depending on the type of system being used in the classroom: *.mail_pref*, *.vtlrc*, and *.env*.

Exercise: Displaying file types

You should see the following types listed when using the **file** command:

Table 1: File Types

filename	type
<i>bigfile</i>	English text
<i>what_am_i</i>	data
<i>bin/treed</i>	commands text
<i>Poems</i>	directory

Exercise: Listing files and directories

If you attempt to list a file that does not exist, the system responds with an error message similar to the following:

```
ls: badname not found: No such file or directory (error 2)
```


Exercise: Changing directories

The *Hidden* directory does not have execute permission and a “Permission denied” error message is displayed.

Exercise: Viewing text files

Using the **file** command to determine whether a file contains text or data will prevent screen-display problems that can result from using the commands **cat**, **head**, **tail**, **more**, and **pg** on data files.

Exercise: Viewing data files

The **file** command says that */etc/mnttab* is a data file.

Exercise: Copying files and directories

The file *bigfile* in your home directory is 13649 bytes in size.

Exercise: Moving files and directories

The **mv** command fails with “permission denied”.

Exercise: Using filename expansion characters

The following command lists all files beginning with a vowel:

```
$ ls [aeiou]*
```

The file *empty* is listed when this command is used.

Exercise: Removing files and directories

The command **rm -r Save** removes the Save directory.

Exercise: Finding data

Use the following commands to find the specified data (assuming the current working directory is your home directory):

```
$ find /u -name letter -print
```

```
$ grep Mother letter
```

```
$ grep better *
```

The string “Mother” occurs four times in the file *letter*. Three files contain the word “better”; *letter*, *proposal*, and *mooncat*.

Answers to review questions

1. A directory is a special type of file that has as its data the filenames and inode numbers of the files contained within that directory; an ordinary file stores text, data, program instructions, or is empty.
2. Four different types of UNIX files include: ordinary, directory, device, and symbolic links.
3. The `file` command is used to determine whether a file contains text or data..
4. The **ls** family of commands display the contents of directories; `cat`, `more` and `pg` are used for viewing text files; `hd` and `od` can be used to view the contents of data files.
5. An absolute pathname starts with a slash (/) representing the root directory; relative pathnames start from some current directory and do not begin with a slash.
6. The **cp** command creates a new file on the system whose contents are identical to the one it was copied from. The **mv** command merely changes the pathname of an existing file, and the `ln` command either creates a synonym to an existing file (hard link) or it creates a new file whose content is the pathname of another file (symbolic link).
7. The benefits of filename expansion characters are saving time and keystrokes when doing repetitive tasks on multiple files, reducing typing errors, and including files that may have been forgotten. The disadvantages are removing files that you wanted or including files that you did not want in some command.

8. If you are in your home directory, you would use the following find command to search for a file under your home directory: **find . -name filename -print** If you are in a subdirectory of your home directory, you would use the dot-dot notation: **find .. -name filename -print**.

Lab: Using the file system

This section provides suggested answers to the lab exercises in Lesson 4.

Lab 1: Viewing the file system

Before viewing a file with **cat**, **more**, **head**, **tail**, or **pg**, you should verify that the file is a text file. You can do this using the **file** command.

Lab 2: Managing files and directories

- The **cp** command creates two files with different inodes and their own copies of the same data. The filenames may be the same, but their full pathnames must be different.
 - The **mv** command does not change either the data or the inode of the file; it simply changes the full pathname.
9. The **ln** command, when making a hard link performs the same action as the **mv** command but leaves the original filename in place creating a new name. A symbolic link creates a new filename only.

5

File Security

AV 5-1

File Security

- Identify permissions for ordinary and directory files
- Use **chmod** to change file permissions
- Reassign user and group file ownership
- Define the default permissions for new files

ntro0501 © The Santa Cruz Operation, Inc.

Lesson 5

File Security

5.1 Introduction

When you access SCO UNIX system commands, you access information stored in a structured environment. All of this information is stored in a hierarchical structure that allows you to maintain your data in an orderly and accessible way. It is important that you learn not only how to access this data, but also to control its access.

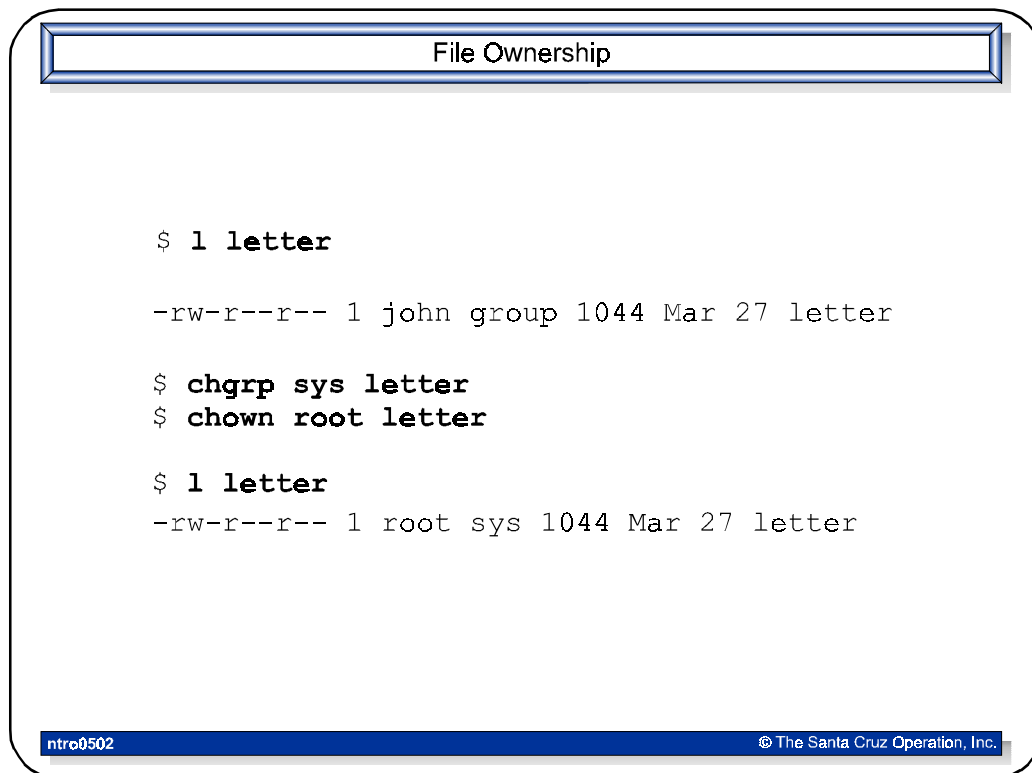
In this lesson you will learn how to use SCO UNIX commands to organize and manipulate files and directories as well as how to manage and maintain your personal *home* directory.

Objectives

This lesson explains the relationship between files and directories and how to work with them using SCO UNIX commands. By the end of this lesson, you will be able to:

- Identify and differentiate the permissions for ordinary and directory files
- Use **chmod** to change the permissions of a file
- Reassign user and group file ownerships
- Define the default permissions for newly created files.

AV 5-2



5.2 Maintaining Security

You have seen examples of not being able to perform certain tasks with UNIX commands because of permission problems. You will now learn how these permissions are controlled.

5.2.1 Ownership

As a means of implementing data security, SCO UNIX systems assign every file an owner that corresponds to one of the authorized login names. Control of a file is determined by the owner of that file (or **root**).

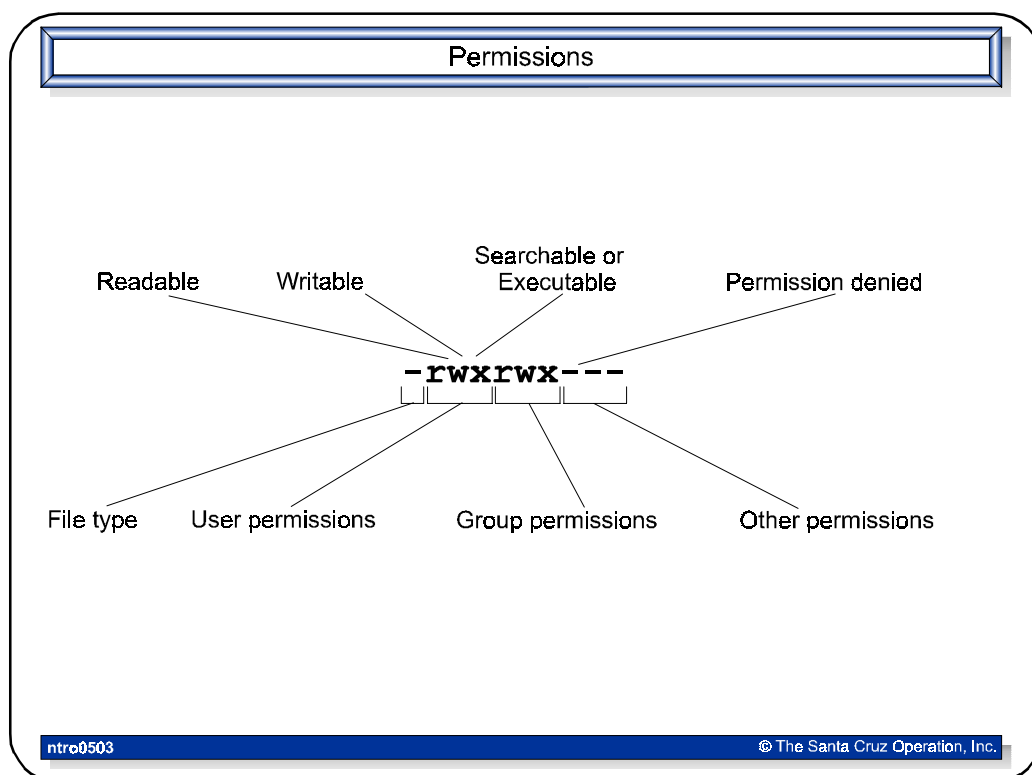
- Every user is uniquely identified when logged in to the system. Your user identification (UID) information can be displayed using the **id** command.
- Whenever you create a file or a directory, it is marked as owned by you, which gives you certain privileges on that file.
- The ownership of a file can be changed and given to another user. Only **root** and the current owner of a file have the privilege of changing the file ownership.
- The property that identifies the file ownership is the user id (UID)
- Use the **chown** command to change a file's UID. The syntax is:

```
chown owner file [ file ... ]
```

owner must be a valid UID or user name; *file* is the name of a file(s)

- If you change the ownership of a file, you will not be able to regain ownership of it unless the new owner or the system administrator reassigns ownership back to you
- **chown** is often used by the system administrator to reassign file ownership after copying important files to another user's directory

AV 5-3



5.2.1.1 Groups

On SCO UNIX systems, every file is also associated with a user group.

- Whenever you create a file or a directory, it is given a group ownership, which gives all members of that group certain privileges over that file
- The property that identifies a file's group ownership is the group id (GID)
- The group id of a file can be changed only by **root** or the owner of the file
- Use the **chgrp** command to change a file's GID. The syntax is:

```
chgrp group file [ file ... ]
```

group must be a valid GID or group name; *file* is the name of a file(s)

5.2.2 Access permissions

Access permissions allow you to determine who can have access to your personal files and directories; they control who can do what to a file.

- Permissions affect three categories of users:
 - The owner of the file (user)
 - A member of the file's group (group)
 - All other users that have access to the system (other)
- Permissions allow three types of access to a file:
 - Read (r)
 - Write (w)
 - Execute or search (x)

AV 5-4

File Permission Types	
READ	Enables users to open files and read its contents using: more, cat, grep, sort, view, <
WRITE	Enables users to open a file and change its contents using: vi, word, >
EXECUTE	Enables users to execute files as commands

ntro0504 © The Santa Cruz Operation, Inc.

- The set of permissions, along with the file type, is called the file *mode*
- You can view the mode of a file using the **l** command
- The first character in each line identifies the file type. A dash (-) in this position indicates an ordinary file, while a **d** indicates a directory
- The next nine characters of the line indicate the current three access types (**rwX**) for each of the three categories of users: user, group and other
- If a user is granted a particular access type, the corresponding letter appears in the listing; if permission is denied for that access, a dash (-) appears

The meanings of the different access types vary, depending on the file type.

5.2.2.1 File access permissions

- *Read* (**r**) permission gives the designated user the ability to open the file and view its contents. Commands that access file contents as input require read access. For example, **cat**, **more**, **file**, and **cp** (for the source file argument(s)).
- *Write* (**w**) permission gives the designated user the ability to open the file and change the contents. Commands that open a file and write to or change the contents include **vi**, **mail**, and **cp** (for the destination file argument).
- *Execute* (**x**) permission gives the designated user the ability to run (i.e., execute) a binary file or to run a correctly written shell script as a command. To invoke a command by typing its name, you must have execute permission for that command file. For example, the **lf** command is found in */bin/lf*. When you type **lf**, you are executing this file.

AV 5-5

Directory Permission Types	
READ	Users can view filenames in the directory
WRITE	Users can create or delete files in the directory
SEARCH	Users can search in the directory and change to it using the cd command

intro0505 © The Santa Cruz Operation, Inc.

5.2.2.2 Directory access permissions

You can use the **-d** option to the **l** command to display information about a directory file (including its permissions) rather than about the files contained within that directory:

```
l -d DirectoryName [ DirectoryName ... ]
```

- **Read (r)** permission allows you to list the names of files stored in that directory (read a table of contents for the directory). This permission also allows the shell to use filename expansion metacharacters (wildcards) to match filenames.
- **Write (w)** permission allows you to remove existing files from or add new files to the directory. Because write permission allows for data removal, usually only the owner of the directory has write permission.
- **Search (x)** permission allows you to search in the directory and change to it using the **cd** command. Without this permission, you cannot use this directory as a component in a pathname. This would effectively block access to this sub-branch of the hierarchical file system.

In general, you need read and search permission to use a directory in any practical way. If search permission is denied on a directory file, the other permissions will usually be denied, too.

Exercise: Viewing permissions

In your home directory, use the **l** command to review your file types and permissions. Do a long directory listing of your local *bin* directory. Is this what you expected? How can you look at the *bin* directory's permissions?

AV 5-6

Changing Permissions: Symbolic Mode

```
$ 1
drwx----- 3 helga training 68 Aug 24 14:25 Letters
-rw-r--r-- 1 helga training 2095 Aug 24 14:32 schedule
-rwxrwxr-- 1 helga training 762 Aug 25 19:41 timecal
```

Both the user and the group have read, write, and execute permissions, while others have only read permission.

```
$ chmod g-w, o+x-r timecal
$ 1
```

```
drwx----- 3 helga training 68 Aug 24 14:25 Letters
-rw-r--r-- 1 helga training 2095 Aug 24 14:32 schedule
-rwxr-x--x 1 helga training 762 Aug 25 19:41 timecal
```

The user retains all permissions, while the group loses write permission, and others lose read permission but gain execute permission.

5.2.2.3 Changing permissions: symbolic mode

Permissions on a file or directory may be modified by the owner, or **root**, using the change mode command, **chmod**.

- Use **chmod** to make changes to the current permission settings:

```
chmod key file [ file ... ]
```

- The *key* is comprised of the following components:

```
[who] [+ | - | =] [mode]
```

- The *who* component of the key can contain one or more of the characters: **u**, **g**, **o**, or **a**, representing user, group, other, or all three categories, respectively. If omitted, **a** is used by default.
- The second component must be one of three characters: **+**, **-**, or **=**, representing adding, removing, or explicitly setting the mode, respectively
- The *mode* component can be any combination of **r**, **w**, or **x**, representing read, write, or execute/search permissions, respectively
- The three components must be entered in order with no spaces between them. Multiple keys can be specified; each is separated by commas (and no spaces).

AV 5-7

Changing Permissions: Absolute Mode

Each permission is assigned a number:

r = 4 w = 2 x = 1

Add these numbers for each user category:

```

-rw-r-----
 421 421 421
  √  √  √
  6   4   0
  
```

Use with **chmod**

```
$ chmod 640 file
```

ntro0507
© The Santa Cruz Operation, Inc.

AV 5-8

Changing Permissions: Absolute Mode

```

$ 1
drwx----- 3 sandy training 68 Aug 24 14:25 Letters
-rw-r--r-- 1 sandy training 2095 Aug 24 14:32 schedule
-rwxrwxr-x 1 sandy training 762 Aug 25 19:41 timecal
  [ ] [ ] [ ]
  7  7  5

$ chmod 751 timecal
$ 1
drwx----- 3 sandy training 68 Aug 24 14:25 Letters
-rw-r--r-- 1 sandy training 2095 Aug 24 14:32 schedule
-rwxr-x--x 1 sandy training 762 Aug 25 19:41 timecal
  [ ] [ ] [ ]
  7  5  1
  
```

4 = r: readable
2 = w: writeable
1 = x: executable

ntro0508
© The Santa Cruz Operation, Inc.

5.2.2.4 Changing permissions: absolute mode

Permissions may be explicitly set using the **chmod** command with a different type of key. This alternate form uses numbers to represent the permissions.

- The *key* is comprised of three (or more) digits, each of which represents the permissions for a category of user (owner, group, other)
- Each digit is determined by assigning a number to the various permissions: *r*=4, *w*=2, and *x*=1.
 - The value 0 is used if a permission is to be denied
 - Permission values are added together to give a number ranging from 0 to 7
- The *key* may form a three-digit number—the first digit sets the user’s permission, the second sets the group’s permission, and the third sets other user’s permissions
- For example, to make the file *myfile* readable and writable by the owner and group, but read-only for all others, use the command:

chmod 664 myfile

AV 5-9

Permissions Exercise																																																																
Directory1: drwxr-x--x file1: -rw-rw-r-- <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 20%;">owner</td> <td style="width: 20%; text-align: center;">__modify</td> <td style="width: 20%; text-align: center;">__delete</td> <td style="width: 20%; text-align: center;">__read</td> <td style="width: 20%; text-align: center;">__execute</td> </tr> <tr> <td>group</td> <td style="text-align: center;">__modify</td> <td style="text-align: center;">__delete</td> <td style="text-align: center;">__read</td> <td style="text-align: center;">__execute</td> </tr> <tr> <td>other</td> <td style="text-align: center;">__modify</td> <td style="text-align: center;">__delete</td> <td style="text-align: center;">__read</td> <td style="text-align: center;">__execute</td> </tr> </table> file2: -rwxr-xr-x <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 20%;">owner</td> <td style="width: 20%; text-align: center;">__modify</td> <td style="width: 20%; text-align: center;">__delete</td> <td style="width: 20%; text-align: center;">__read</td> <td style="width: 20%; text-align: center;">__execute</td> </tr> <tr> <td>group</td> <td style="text-align: center;">__modify</td> <td style="text-align: center;">__delete</td> <td style="text-align: center;">__read</td> <td style="text-align: center;">__execute</td> </tr> <tr> <td>other</td> <td style="text-align: center;">__modify</td> <td style="text-align: center;">__delete</td> <td style="text-align: center;">__read</td> <td style="text-align: center;">__execute</td> </tr> </table> file3: -r-xr-xr-- <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 20%;">owner</td> <td style="width: 20%; text-align: center;">__modify</td> <td style="width: 20%; text-align: center;">__delete</td> <td style="width: 20%; text-align: center;">__read</td> <td style="width: 20%; text-align: center;">__execute</td> </tr> <tr> <td>group</td> <td style="text-align: center;">__modify</td> <td style="text-align: center;">__delete</td> <td style="text-align: center;">__read</td> <td style="text-align: center;">__execute</td> </tr> <tr> <td>other</td> <td style="text-align: center;">__modify</td> <td style="text-align: center;">__delete</td> <td style="text-align: center;">__read</td> <td style="text-align: center;">__execute</td> </tr> </table> Directory2: drwxrwxr-x file1: -rwxr-xr-x <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 20%;">owner</td> <td style="width: 20%; text-align: center;">__modify</td> <td style="width: 20%; text-align: center;">__delete</td> <td style="width: 20%; text-align: center;">__read</td> <td style="width: 20%; text-align: center;">__execute</td> </tr> <tr> <td>group</td> <td style="text-align: center;">__modify</td> <td style="text-align: center;">__delete</td> <td style="text-align: center;">__read</td> <td style="text-align: center;">__execute</td> </tr> <tr> <td>other</td> <td style="text-align: center;">__modify</td> <td style="text-align: center;">__delete</td> <td style="text-align: center;">__read</td> <td style="text-align: center;">__execute</td> </tr> </table>					owner	__modify	__delete	__read	__execute	group	__modify	__delete	__read	__execute	other	__modify	__delete	__read	__execute	owner	__modify	__delete	__read	__execute	group	__modify	__delete	__read	__execute	other	__modify	__delete	__read	__execute	owner	__modify	__delete	__read	__execute	group	__modify	__delete	__read	__execute	other	__modify	__delete	__read	__execute	owner	__modify	__delete	__read	__execute	group	__modify	__delete	__read	__execute	other	__modify	__delete	__read	__execute
owner	__modify	__delete	__read	__execute																																																												
group	__modify	__delete	__read	__execute																																																												
other	__modify	__delete	__read	__execute																																																												
owner	__modify	__delete	__read	__execute																																																												
group	__modify	__delete	__read	__execute																																																												
other	__modify	__delete	__read	__execute																																																												
owner	__modify	__delete	__read	__execute																																																												
group	__modify	__delete	__read	__execute																																																												
other	__modify	__delete	__read	__execute																																																												
owner	__modify	__delete	__read	__execute																																																												
group	__modify	__delete	__read	__execute																																																												
other	__modify	__delete	__read	__execute																																																												

ntro0509
© The Santa Cruz Operation, Inc.

Exercise: Permissions

The slide on the opposite page lists two directories; the first contains three files, and the second contains one file.

Analyze the file and directory permissions listed, and determine which actions users can take on each file.

Place check marks beside the actions that can be taken.

Hint: start with the directory permissions to determine whether or not the user has access to the directory.

5.2.3 Default permissions

You can control what permissions are assigned to files and directories when you create them using the **umask** command.

- By default, the system creates ordinary files with read and write (**rw**) permission for all user categories. It creates directories with read, write, and search (**rwX**) permissions for all user categories.
- When you log into the system, you are given a value called a *umask*. The *umask* tells the system what file permissions should be excluded from which categories when you create your files and directories.
- The **umask** command displays this value:

```
$ umask  
022
```

- This mask indicates that both *group* and *other* will be denied write permissions on files and directories that you create

- Your umask value can be changed by entering a new value as an argument to the **umask** command. The mask may be expressed either as a number, or using the notation allowed with the **chmod** command. The following two commands both exclude other users from write access to newly created files:

```
umask 002  
umask o-w
```

To demonstrate how default permissions are applied to new files based on your umask value, the **touch** command can be used to easily create files. When used with a filename that does not exist, **touch** will create a new, empty file. When used with an existing filename, the file time stamp will be updated but the contents will not be changed. The syntax is:

```
touch filename [ filename ... ]
```

Exercise: Using the umask command

Use the **umask** command to see what your current setting is.

Use the **touch** command to create a file named *perm1*. View its permissions with a long directory listing on that file.

Now set your umask to deny other users read access to your files.

Use the **touch** command to create a new file *perm2* and compare its permissions with *perm1*.

5.3 Lesson Summary

The SCO UNIX operating system stores information in a hierarchical system of files that is organized by directories into a structure that resembles an inverted tree. The system provides numerous commands for accessing these program, text, and data files. As a user, your data may reside in your personal home directory. Various commands and conventions enable you to keep this information accessible, organized, and protected.

5.3.1 Review questions

1. If you are not able to edit a file for which you have write and read permission, what should you check next?

2. What effect does the following command have on Steve's home directory?

```
chmod o-rx /usr/steve
```

3. What would happen if you used this command in your current working directory?

```
chown root .
```

4. What permissions will new files be given if your umask is set to 022?

5.4 Lab: Securing files

You will learn how to assign permissions to files and directories to provide a measure of security and privacy to your files on a multiuser system.

Overview

Using the **chmod** command, you will modify the permissions on files and subdirectories in your home directory.

Procedure

1. Change to your home directory. Type the **l** command. Who owns these files? To which group do these files belong? How can you distinguish files entries from directory entries?
2. Change your working directory to the directory *Shakespeare* in the directory *Poems*.
3. Make the file *sonnet4* read-only, so that only you can read it.
4. Currently, you have permission to write to all the *sonnet* files. Use **chmod** to protect them by taking away your own write permission. Can you do it using a single command with the appropriate filename expansion character?
5. Make a directory called *Public* in your home directory, and make it so that everyone in your group can read and search the directory; all others should have no permission to use it.
6. Try to find what is hidden in the directory called *Hidden*, located in your home directory. Why can't you list the file(s) stored there? Can you make it your current working directory with **cd**?
7. Change the permissions on the directory *Hidden* so that you can access it in the traditional fashion (restore the default directory permissions). See what is in the directory now.
8. With your fellow student's approval, try to remove the file *twister* located in his or her *Poems* subdirectory. Why can't you do it?

5.5 Relevant Commands

The following commands were presented in this lesson.

Relevant commands

cd	Changes the working directory
chgrp	Changes the group ID of a file
chmod	Changes file or directory access permissions
chown	Changes the owner ID of a file
l	Lists information about directory contents
lf	Lists directory contents
touch	Creates an empty file or updates modification times of an exiting file
umask	Gets or sets file-creation mode mask

5.6 Answers to Exercises, Labs, and Review Questions

This section provides suggested answers to the exercises, labs, and review questions.

Exercise: Viewing permissions

To look at the permissions of a directory using the **ls** command, use the **-d** option to avoid displaying the contents of that directory.

Exercise: Permissions

```
Directory 1: drwxr-x--x
file1: -rw-rw-r--
  owner      modify      delete      read      -
  group      modify      -           read      -
  other      -           -           read      -

file2: -rwxr-xr-x
  owner      modify      delete      read      execute
  group      -           -           read      execute
  other      -           -           read      execute

file3: -r-xr-xr--
  owner      -           delete      read      execute
  group      -           -           read      execute
  other      -           -           read      -

Directory 2: drwxrwxr-x
file1: -rwxr-xr-x
  owner      modify      delete      read      execute
  group      -           delete      read      execute
  other      -           -           read      execute
```

Exercise: Using the umask command

Set your umask to 0066 to deny other users write access.

Answers to review questions

1. When file access is denied, check the permissions on the directory, and if necessary, all parents of the directory (up to /). You must have permissions for the directory to access files within the directory.
2. Denying read and search permission for all others on the system would prevent any other users (except root) from making Steve's home directory their current directory. It would also prevent other users from viewing any of his files.

3. Changing ownership of your directory to root (or another user) will prevent you from having full access to your files and subdirectories. Always use caution when changing directory ownership and never change ownership of your home directory.
4. With umask set to 022, new files are created with permissions -rw-r--r--.

Lab: Securing files

This section provides suggested answers to the lab exercises in Lesson 5.

- The **chmod** command can be used with wildcard characters to specify multiple filenames. For example, remove write permission on all files in a directory with:

chmod u-w Poems/Shake*/*

5. When you get an error from the system saying “Permission denied”, first check the permissions on the file you are trying to access. If they are all right, then check the permissions on the directory in which the file resides. If they are all right, check the permissions on the parent directories all the way to the root directory using the **l** command and making sure you have read and execute permission of each directory.

6

Text Editing

AV 6-1

Text Editing

- List the primary functions of a text editor
- Add, replace, change, and delete text in a file using the **vi** editor
- Control the editing environment by setting preferences
- Use the system **spell** program to check the spelling in your text files

ntro0601© The Santa Cruz Operation, Inc.

Lesson 6

Text Editing

Introduction

Text files are ordinary files made up of printable ASCII characters. The system uses many different text files for configuration information. Likewise, you will likely have many text files in your home directory (such as reports, letters, and configuration files) that store your own data. In addition to placing the initial data (text) in these files, you may need to modify them (*edit* them) from time to time.

Several text editors are provided with SCO UNIX systems. In this lesson, you will learn to use one to create new files and edit existing ones.

Objectives

This lesson covers how to perform basic editing tasks using the **vi** text editor. By the end of this lesson, you will be able to:

- List three primary functions of a text editor
- Add, replace, change, and delete text in a file using the **vi** editor
- Control the editing environment by setting preferences
- Use the system spell program to check the spelling in your text files

6.1 Overview of Text Editors

A text editor is a program used for writing and modifying text files. Unlike a word processor, a text editor does not usually include document-formatting features. For this reason, text editors are quicker and more convenient to use.

Word processors generally do not create text files; they create data files containing binary data which can often be interpreted only by the word processing package itself. As you recall, many of the commands you have already used, such as **grep**, **more**, **diff**, are designed to work only on text files—not data files.

6.1.1 Uses of a text editor

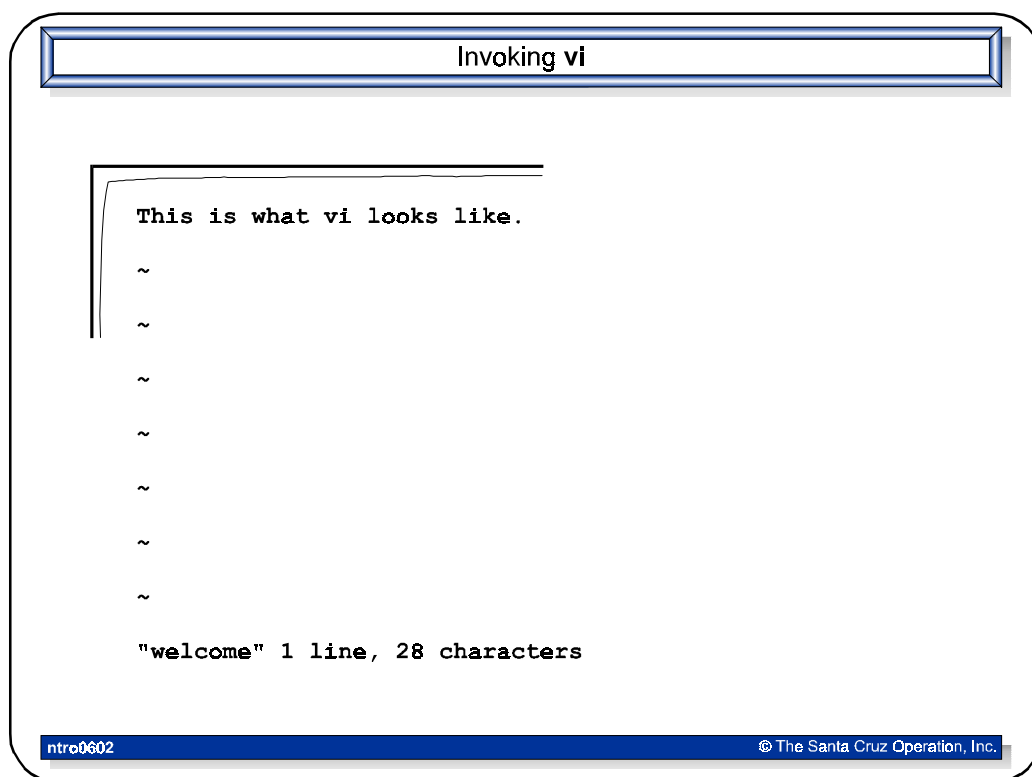
Text editors can be used for a variety of purposes, including:

- Communication needs
 - Composing mail messages
 - Keeping logs
- Editing configuration files
 - Editing utility configuration files
 - Altering system configuration files
- Programming
 - Writing shell programs
 - Writing source code for various language compilers

6.1.2 Overview of vi

- **vi**, which stands for “visual editor,” is a full-screen editor
- **vi** is a command-based editor that uses commands to perform powerful line-oriented and screen-oriented tasks
- **vi** runs in one of two modes: *command* mode for issuing editing commands, and *input* mode for adding or replacing text

AV 6-2



6.2 Using vi

The power of the **vi** extended command library and its availability in graphical and terminal-based environments make it the preferred editor by many system administrators.

The **vi** command has file links to other command names. These commands invoke the same editor, but in different modes of operation:

- **vi**—visual (full screen) mode
- **view**—read-only mode (changes cannot (easily) be made to the file)
- **vedit**—variant of **vi** that has user-friendly options automatically set
- **ex**—line-oriented mode available from the **vi** visual mode with the colon (:) command
- **edit**—variant of **ex** that has user-friendly options automatically set

6.2.1 Invoking vi

The syntax of the **vi** command is:

```
vi [ filename ]
```

- If the file exists, you will see the first screen of its contents
- If you are creating a new file, you will see a message at the bottom of the screen indicating this
- When editing a new or small file, **vi** indicates lines not yet allocated to the file with the tilde (~) character. The first tilde indicates the end of the file.
- When you enter **vi**, you begin in command mode
- Use the colon (:) command to enter the **ex** line-oriented mode of operation. This will place the cursor at the left side of the bottom line.

AV 6-3

Saving and Quitting in vi

:q	Quit, if no changes were made
:q!	Quit, throw away any changes
:w	Write (save) the file
:wq	Write the file then quit vi
ZZ	Write the file if any changes were made, then quit
:x	Same as ZZ
:w newname	Write the file to a new name

ntro0603

© The Santa Cruz Operation, Inc.

6.2.2 Saving and quitting

The **vi** editor creates a temporary copy of the file which is what it directly edits. You do not automatically modify the original file as you modify the screen version of it until you save (update) your changes to the file

- From the command mode, type **:w** to write (save) changes to a file
- Writing a file instructs **vi** to write its temporary buffer out to disk, thereby saving your work
- When you have finished working on a file and want to save your changes and quit the editor, type **:wq**
- The commands **:x** and **ZZ** also save your work and exit the editor
- If no changes were made to the file you are editing, you can quit the editor with the **:q** command. If changes have been made and you don't want to save those changes, you can abandon (abort) your editing session with the command:

:q! 

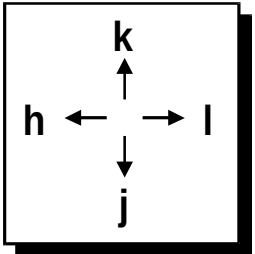
Exercise: Entering and exiting vi

Use **vi** to create a new file called *mine*. Quit from *mine* without saving it. Use the **l** command to see if the file *mine* was created.

Invoke **vi** again with the filename *mine*. This time, quit after saving the file. Use the **l** command to list the file. How big is the file? What is in it?

AV 6-4

Moving in vi



w	Moves the cursor to the beginning of the next word.
b	Moves the cursor to the beginning of the previous word.
\$	Moves the cursor to the end of the current line.
.	Moves the cursor to the beginning of the current line.
 u	Moves the cursor up one half screen.
 d	Moves the cursor down one half screen.

ntro0604© The Santa Cruz Operation, Inc.

6.2.3 Commands for moving the cursor

In a full screen editor, you can move the cursor to the point in the file where you want to enter text or make changes.

If your keyboard has arrow keys, you can use them to move left, right, up, and down. **vi** sounds a bell when you try to move the cursor beyond the end of the file, above the top of the file, or beyond the ends of line within the file.

Preceding a command by a number causes that command to perform its function that number of times.

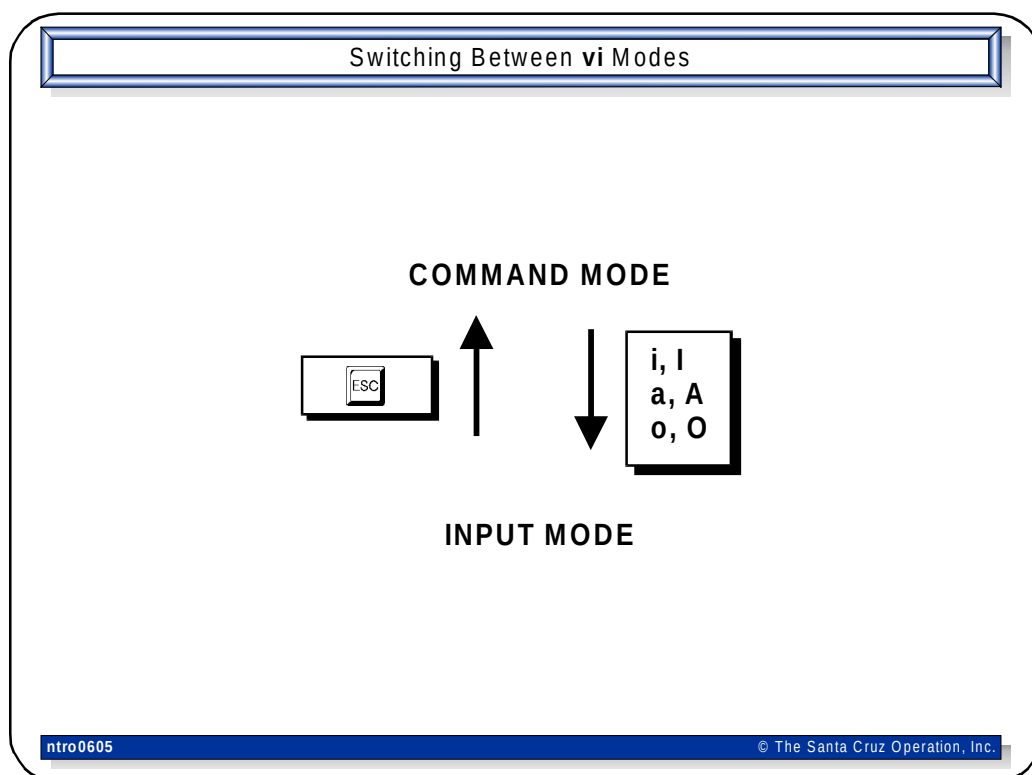
This list summarizes the **vi** movement commands:

- **h** and  move the cursor one character to the left
- **l** and  move the cursor one character to the right
- **k** and  move the cursor up one line
- **j** and  move the cursor down one line
- **w** moves the cursor one word forward
- **b** moves the cursor one word backward
- **\$** moves the cursor to the end of the current line
- **0** (zero) and **^** move the cursor to the beginning of the current line
- **H**, **M**, and **L** move the cursor to the top, middle, and bottom of the current screen
- **nG** moves the cursor to the *n*th line in the file
 - **G** moves the cursor to the bottom line in the file
 - **1G** moves the cursor to the first line in the file
-  **u** scrolls the cursor up one half screen
-  **d** scrolls the cursor down one half screen
-  **b** scrolls the cursor back one full screen
-  **f** scrolls the cursor forward one full screen

Exercise: Navigating within a file

Invoke **vi** with the file *small_town* and practice moving to various locations within the file. Do not attempt to correct any spelling errors at this time.

AV 6-5



6.2.4 Editing text

The **vi** editor uses two modes of operation — *command* and *input*. When in command mode, keyboard input is interpreted as a command; when in input mode, keystrokes are entered into the file as text.

- Invoking **vi** places you in command mode
- You can change from command mode to input mode by using an input command to insert, append, overwrite, or replace text
- You cannot enter commands in input mode; you must return to command mode first by pressing the  key
- If you do not know which mode you are in, press . If you are already in the command mode, the terminal simply beeps.
- To display a message indicating when you are in insert mode, change to command mode and type:

:set showmode

The **vedit** command sets this option automatically.

AV 6-6

Basic vi Commands		
Insert a i o	Delete x dw dd	Modifying Text r cw c\$
Saving & Quitting :wq :q!	Cursor Movement  G 1G	Command Mode 

ntro0606 © The Santa Cruz Operation, Inc.

6.2.5 Entering text

To insert new text into a file, position the cursor at the spot you want to start entering the text. Then enter the appropriate input command without pressing the  key.

These commands put **vi** into input mode:

- **i** inserts text immediately before the cursored character
- **a** appends text immediately after the cursored character
- **o** opens a line below the cursor and places the cursor at the beginning of that new line

The upper case form of these commands modifies their behavior:

- **I** inserts text at the start of the cursored line
- **A** appends text at the end of the cursored line
- **O** opens a line above the cursor and places the cursor at the beginning of that new line

When you finish entering text, remember to press the  key to re-enter command mode before you attempt to move the cursor to another location.

6.2.5.1 Deleting text

Deleting characters, words, and lines of text is always done in command mode. You always remain in command mode after the action.

Like the movement commands, a number typed before a delete command will cause the command to be executed the specified number of times.

These commands delete one character:

- **x** deletes the character the cursor is on
- **X** deletes the character immediately before the cursor
- **J** joins the current line with the following line, replacing the return at the end of the line with a space

The **d** command is followed by a single cursor movement command. It deletes from the current cursor position to wherever the cursor movement command indicates. If the cursor movement command is omitted and **d** is simply repeated, it deletes the entire line.

- **dw** deletes a word; **5dw** or **d5w** deletes five words
- **dd** deletes an entire line; **7dd** or **d7d** deletes seven lines
- **d\$** or **D** deletes from the cursor to the end of the line
- **d0** or **d^** deletes from the cursor to the beginning of the line

Exercise: Entering and deleting text

Delete some of the duplicate lines in the *small_town* file.

Delete some of the repetitive characters using a number followed by the **x** command.

Go into input mode and add your own comment about small towns. Don't forget to press  when you have finished your input.

6.2.6 Changing text

Several commands allow existing text to be changed. As with the delete commands, they may all be preceded by a count value.

Sometimes you may want to replace existing text:

- **r** replaces the cursored character with one character. You are automatically returned to command mode, so there is no need to press .
- **R** continuously replaces text until you press  to return to command mode

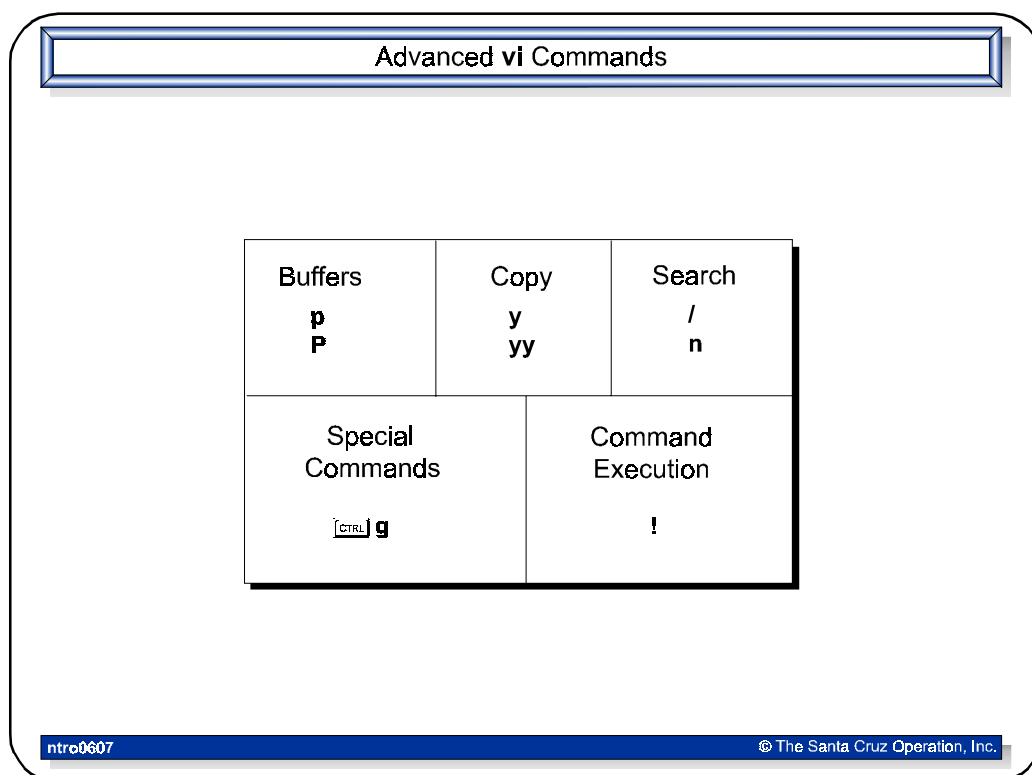
The **s** commands leave you in input mode once you have invoked them; therefore, you must press  upon completing the change to get back to command mode. A dollar (\$) symbol will appear at the last character of the existing text that will be changed by these commands.

- **s** substitutes the current character with zero or more characters. Press  to return to command mode.
- **S** substitutes an entire line of text. Press  to return to command mode.

The **c** command (followed by a single cursor movement command) changes text from the current cursor position to wherever the cursor movement command indicates. If the cursor movement command is omitted and **c** is simply repeated, it changes the entire line.

- **cw** changes a word; **5cw** or **c5w** changes five words
- **cc** changes an entire line; **7cc** or **c7c** changes seven lines
- **c\$** or **C** changes from the cursor to the end of the line
- **c0** or **c^** changes from the cursor to the beginning of the line

AV 6-7



6.2.7 Copying and moving text

Commands can be combined to achieve the cut, copy, and paste features of most word processors.

To cut text, use a single **x** or **d** command to delete the characters, words, or lines you want cut. These commands remove the text you have indicated and place it in a temporary buffer.

To copy text into this same buffer, use the **y** (yank) command, which is followed by a single cursor movement command. It copies to the buffer from the current cursor position to wherever the cursor movement command indicates. If the cursor movement command is omitted and **y** is simply repeated, it copies the entire line to the buffer.

- **yw** copies a word to the buffer; **5yw** or **y5w** copies five words
- **yy** or **Y** copies an entire line to the buffer; **7yy** or **y7y** copies seven lines
- **y\$** copies to the buffer from the cursor to the end of the line
- **y0** or **y^** copies to the buffer from the cursor to the beginning of the line

To paste text (i.e., copy from the buffer), use the **p** command:

- **p** pastes the text from the temporary buffer below the current line or after the cursor, depending on whether the buffer contains a partial line or not
- **P** pastes the text from the temporary buffer above the current line or before the cursor, again depending on whether the buffer contains a partial line or not

Unlike a clipboard, the contents of **vi**'s temporary buffer are saved only until the next editing command when the contents are replaced by that command's text.

These commands undo previous edits:

- **u** cancels your most recent edit
- **U** restores the current line to its previous state

Exercise: Rearranging text

Move the credit line in the *small_town* file to the end of the file.

Try correcting some of the numerous spelling errors. Experiment with the **undo** commands.

6.2.8 Special commands

Many special commands allow you to work more efficiently with the contents of your file. These commands must be invoked from command mode.

6.2.8.1 Convenience commands

- **CTRL g** provides you with a status line that tells you the file name, how many lines are in the file, and on what line number the cursor currently is located
- The dot command (**.**) repeats the last insertion or deletion command that was performed

6.2.8.2 The search command

You can search for patterns of characters within your file. A pattern uses *regular expressions*, like **grep**, and can include spaces.

- **/ pattern** searches forward in the file for the specified *pattern* of characters from the current position. By default, the search will wrap around to the beginning of the file when it reaches the end.
- **? pattern** searches backward in the file for the specified *pattern* of characters from the current position. By default, the search will wrap around to the end of the file when it reaches the beginning.
- When you type the slash (/) or question (?) characters, the cursor moves to the bottom line of your terminal where you can enter a search pattern

- **n** repeats the last search in the same direction
- **N** repeats the last search in the opposite direction

6.2.8.3 ex commands

Recall that **vi** and **ex** are the same editor, but use different modes of operation. **vi** operates in a visual (full-screen) mode; **ex** operates in a line-oriented mode. The **ex** mode is available from the **vi** visual mode with the colon (:) command.

You used this when you saved your file (:w), exited from **vi** (:wq), and executed the **set** command (:set showmode). Other commonly used **ex** commands include:

- Executing an SCO UNIX command without exiting from your **vi** session:

:! command-line

- Reading the contents of another file and placing the text below the cursor:

:r filename

- Reading the standard output of an SCO UNIX command into your current file:

:r !command-line

Exercise: The vi command line

Invoke **vi** with a new filename. Enter the **:r** command to read in the */etc/motd* file. Then enter the **:!** command to find out what time it is right now.

Combine the **:r** and **:!** commands to bring the date into your file:

:r !date

Try the same command with another SCO UNIX command, such as **ls**.

6.3 Customizing vi

The **vi** command has options that affect its behavior. These options customize the editor's performance to suit individual needs and preferences.

There are two kinds of options:

- Switch options, which are either on or off
- String options, which are assigned a string or numeric value

6.3.1 Setting options interactively

The preference settings in **vi** are controlled by options to the **set** command, which is an **ex** mode command. Recall that this mode can be accessed from within **vi** by pressing the colon (:) character.

Switch options are turned on with the syntax:

```
set optionname
```

Switch options are turned off by prepending the word *no* to the option name (no spaces):

```
set nooptionname
```

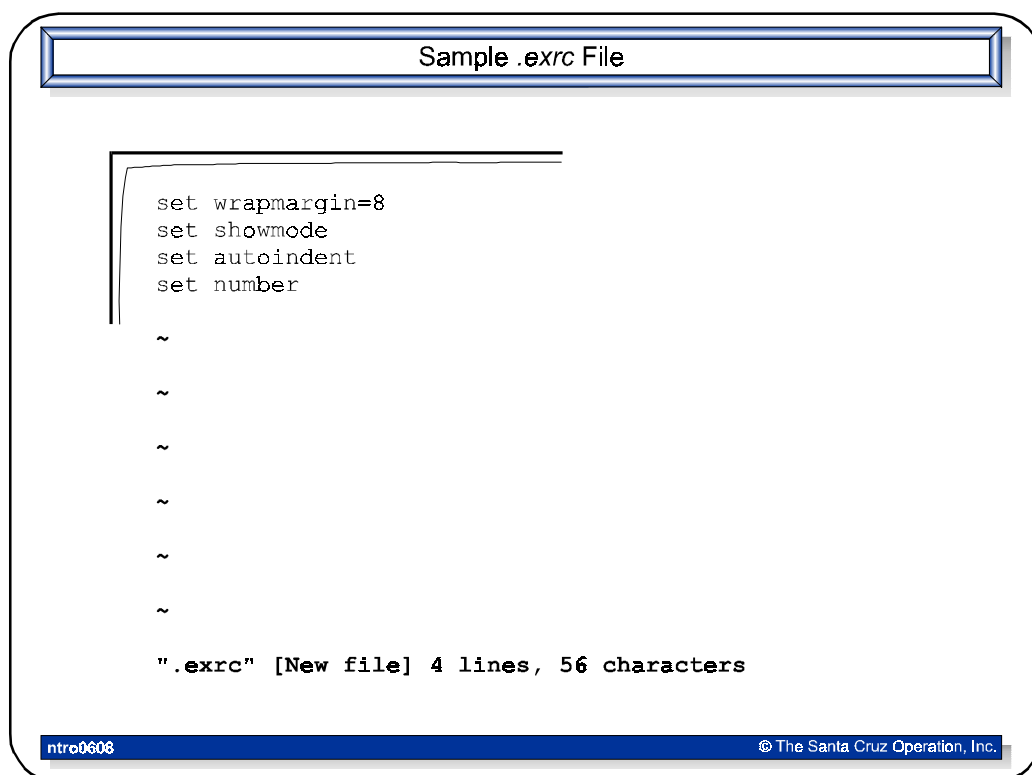
String options are set with the syntax:

```
set optionname=value
```

Several options may be given per line:

```
set optionname optionname=value optionname...
```

AV 6-8



6.3.2 Setting options automatically

Environment files modify the way certain applications behave. To customize the default settings for **vi** each time it is invoked, an environment file `.exrc` is created with the desired values.

- The file `.exrc` is a hidden file that must be located in your home directory
- The `.exrc` file can contain any of the **set** commands supported by **vi**
- The **vi** editor reads the commands from the environment file each time you invoke it and sets those commands for your current session
- You can override the `.exrc` file settings by interactively using the **set** command
- The `.exrc` file should not contain any blank lines. Errors will otherwise result each time you invoke the editor.

6.3.3 The set options

To see a list of the available set options from within **vi**, enter the **set** command with the **all** option:

:set all

Some of the more useful switch options are:

autoindent	Indents each new line to the same tab stop of the previous line ([CTRL] d returns you to the previous tab stop.)
list	Displays tab marks, carriage returns, and end-of-line marks
number	Causes all lines to be preceded with their line numbers
showmode	Displays a message in the lower right-hand corner of the screen when you are in input mode
wrapscan	Causes searches to wrap around past the end of the file

Some of the more useful value options are:

report=<i>n</i>	Sets a threshold for feedback from commands that modify more than <i>n</i> lines (default=5)
tabstop=<i>n</i>	Sets the size of a tab to be <i>n</i> characters (default=8)
wrapmargin=<i>n</i>	Defines the right margin of the display as <i>n</i> characters from the right side of the screen (default=0)

Exercise: The .exrc file

Create a *.exrc* file in your home directory that contains the line:

set number list scroll=6

Now invoke **vi** with the file *small_town*. What differences do you notice?

Enter **CTRL d** to scroll down. What does the **scroll** option do? How can you remove the line numbering?

6.4 Checking Your Spelling

Most word processors have built-in spelling checkers. Text editors, such as **vi**, are simpler utilities that do not contain spell checkers. You can, however, use the UNIX **spell** command to check for misspelled words in a text file.

- The syntax of the **spell** command is:

```
spell [+local_dictionary] file [ file ... ]
```

- **spell** searches one or more files for words not found in its dictionary
- After processing all files, **spell** prints to the screen all words found that did not match any words in its dictionary. These words are printed in ASCII order.
- If there are words that **spell** reports as misspelled, even though they are acceptable to you, it is because that word is not in **spell**'s dictionary. Although you cannot modify **spell**'s dictionary, you can add your own words to a personal dictionary and specify that on the **spell** command line.
- The **+local_dictionary** option refers to a text file that contains one word per line. When a local dictionary file is included on the command line, **spell** will check both the default dictionary and the local dictionary file specified for misspellings. For example:

```
spell +my_words file1 file2 file3
```

spell will check *file1*, *file2*, and *file3* for words not found in either the default **spell** dictionary or in the *my_words* file.

- There are restrictions on using a local dictionary file:
 - There may be no extraneous spaces, including at the end of a line
 - The file must be sorted in ASCII order. (The **sort** command, covered in Section 8.4.2, may be useful for this purpose.)

6.5 Lesson Summary

SCO UNIX operating systems provide the **vi** editor for creating and modifying text files. The **spell** command can be used in conjunction with **vi** to list the misspelled words in your text files.

Review questions

1. List three functions of text editors. How are they different from word processors?
2. What are the two modes in which **vi** works?
3. Name four ways to enter input mode from command mode in **vi**.
4. Describe how to search for strings in **vi**.
5. Describe how you can customize your **vi** editing sessions without having to do so every time you invoke the editor.
6. How can you make the **spell** program accept *SCO* as a correctly spelled word?Lab: Text Editing

The goal of this lab is to become more proficient with the use of **vi** as a text editor. You will create a new directory in which to store files in an editing session, create new files, and edit existing files.

Procedure

1. In your home directory, create a new directory called *Edit*. Make *Edit* your current working directory.
2. Use **vi** to type your name and address into a new file called *information*. Enter the information as though you were addressing an envelope. Save your work and exit **vi**. Look for the file and display it on the screen. Make sure it is correct.
3. Edit the *information* file again. Run the command **banner my_name** and read the output into the top of the file. Save and quit.
4. Copy the files *text.err* and *text.fxd* from your home directory to your current directory. Edit *text.err* and correct the obvious errors. Use the **diff** command to compare your *text.err* file with *text.fxd*.
5. Change to your home directory.
6. Run the **spell** program on the file *small_town* and view the errors.
7. Create your own private *dictionary* file that contains the words **Ayshire** and **mashpit**, each word on its own line.
8. Run **spell** again using your *dictionary* file as an option. Correct the remaining spelling errors in *small_town*.
9. Use **spell** a final time to verify that you have no more spelling errors in *small_town*.
10. Compare *small_town* with the *better_town* file in the *Miscellaneous* subdirectory of your home directory.

6.6 Relevant Commands and Files

The following commands and files were presented in this lesson.

Relevant commands

edit	Variant of ex that has user-friendly options automatically set
ex	Variant of vi that operates in line oriented mode
spell	Finds spelling errors in text files
vi	Text editor that operates in visual (full screen) mode
vedit	Variant of vi that has user-friendly options automatically set
view	Variant of vi that operates in read-only mode

Relevant files

.exrc	File in your home directory that customizes vi sessions
--------------	--

6.7 Answers to Exercises, Labs, and Review Questions

This section provides suggested answers to the exercises, labs, and review questions.

Exercise: Entering and exiting vi

On SCO OpenServer Release 5 and UnixWare 7, a file you create will be 0 bytes in size, that is, it will have no contents. On older versions of SCO UNIX the file may be created with 1 byte, a newline character with which the **vi** utility terminates a line.

Exercise: The .exrc file

You should notice that every line in your file is preceded by its ordinal line number, that at the end of each line there is a dollar sign (\$) character, and that tabs are indicated by the symbol ^I.

If you scroll through the file using **CTRL d**, you will notice that the screen scrolls 6 lines instead of 12.

You can get rid of the line numbering by issuing the following command:

```
:set nonumber
```

Answers to review questions

1. Text editors are used for creating and editing mail messages and short memos, for editing configuration files, and for writing shell command files or source code for various language compilers. The difference between a text editor and a word processor is that a text editor does not have text formatting capabilities, or features for spell-checking, creating bibliographies, table of contents, indexes, etc. Text editors, however, create text files whereas word processors create data files.
2. The **vi** editor works in two modes: command and input. Command mode interprets the characters you type as editing commands, whereas input mode treats the characters you type as data you are inputting to the file.
3. From **vi** command mode, you may enter input mode by pressing the letter **i** for inserting text, the letter **a** for appending text, the letter **o** for opening up a new line of text, and the letter **R** for replacing existing text. The letters **s** and **c** also put you in input mode as you change a character or line of text.

4. Strings may be searched either forward or backward through a text document in **vi**. To search in a forward direction, you type the slash (/) character while in command mode which places the cursor on the bottom line of the screen. From there, you enter the desired search string. **vi** then places the cursor on the first occurrence of that string. Pressing the **n** key finds successive occurrences of the string. Searching backward through a file follows the same procedure except that the question mark character (?) is used to enter the search string on the command line.
5. You can customize your **vi** environment by entering set commands in a *.exrc* file in your home directory. You can set several options with one set command; these options will take effect each time you invoke the **vi** command.
6. To make the spell program not count SCO as a misspelled word, you would enter the word on one line of a text file (e.g., dictionary) and specify the name of that file with the **+** option on the spell command line: `spell +dictionary filename`

Lab: Text Editing

This section provides suggested answers to the lab exercises in Lesson 6.

- When you finish editing the *text.err* file, it should show no differences from *text.fxd*. Similarly, the file *small_town* and *better_town* should be the same.
7. In **vi**, go to the last line of the file by entering the **G** command. Delete the last line with the **dd** command, then go to the first line of the file with the command **1G**. Insert the deleted line above the first line with the **P** command.

7

Printing

AV 7-1

The slide is titled "Printing" and contains the following content:

- Print files to the default printer using the **lp** command
- Direct print requests to other printers
- Monitor the status of print requests and printers
- Cancel print requests
- Describe options to **lp**

ntro0701 © The Santa Cruz Operation, Inc.

Lesson 7

Printing

Introduction

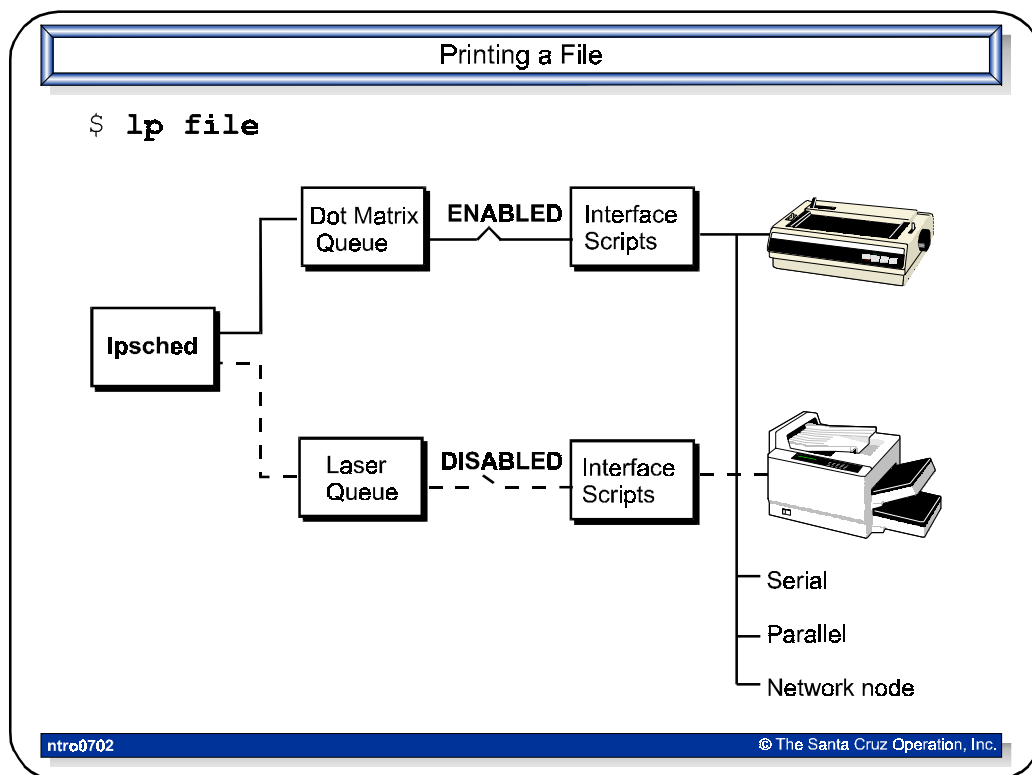
Text files you have created or messages you have received in mail often need to be printed on paper. As one user on a multiuser system, it is important for you to understand how printer requests are handled.

Because printers are a shared resource, the operating system provides a subsystem to manage it, to handle simultaneous requests for its use, and to differentiate jobs among the multiple users. In this lesson you will learn two ways in which the SCO UNIX operating system supports printing.

Objectives

This lesson covers an overview of printers and commands that will help you prepare your files before sending them to the printer. By the end of this lesson, you will be able to:

- Print files to the default printer using the `lp` command
- Direct print requests to other printers
- Monitor the status of print requests and printers
- Cancel print requests
- Describe options to **lp**



7.1 Printer Overview

SCO UNIX supports a wide variety of printer hardware that ranges from high-speed line printers to top quality laser printers. Each type of printer has its place in a large computing environment. The operating system can present these printer services to its users in many different configurations.

Printers are often attached to a computer by a serial or parallel port in the same way that a terminal might be attached to a serial port. Some printers are equipped with network interface cards and accessed as a network node.

Printers are shared among many users by means of the UNIX system's print spooler software, the **lp** subsystem.

7.1.1 Spooled printing

A printer attached to a computer running a multiuser operating system is a shared device—users make multiple requests at the same time. Spooled printing is a method of printing that handles numerous, concurrent requests made to a variety of possible hardware devices.

The SCO UNIX print spooler system is a collection of programs that enable you to submit print requests (jobs), check the status of the print system, track the progress of your requests, and cancel a particular job

- A request to print a file is *spooled* or stored with other printing jobs to be sent to the printer
- Each print job is given a request-ID number and a priority, and waits its turn to be printed. This line of pending print jobs is called a *queue*.
- The SCO UNIX command for spooled printing is **lp** or **lpr**
- Options to the **lp** command will allow you to direct your print requests to a variety of supported printers on the system. Each printer is managed by its own software program called an interface script.

AV 7-3

Basic Print Commands

lp filename	Prints a specified file or files
lpstat	Checks the current status of the default printer
cancel request-ID	Cancels a print request

ntro0703© The Santa Cruz Operation, Inc.

7.2 Printing Files with *lp*

Printing files using the **lp** print service is the most common way that data is printed on SCO UNIX systems. It affords the greatest choice in printer types and is the most efficient way to share a common resource among the users on the system.

7.2.1 Using the print spooler

The print spooler schedules print requests (jobs) as they are lined up in the queue, and it decides to which device the print job should be sent (destination).

- To print a text file to the default system printer, use the **lp** command:

```
lp [ options ] [ file ... ]
```

For example:

```
$ lp mytextfile
request id is laser-450 (1 file)
```

- The **lp** command displays the *request ID* assigned to your request, which is used to monitor or cancel your print request. The request ID is composed of the name of the printer to which your file is being sent and a unique job number.
- Multiple files may be printed at once by specifying any number of filenames as arguments to the **lp** command. The files must be text files; they will be printed in the same order as they appear on the command line. The files are not concatenated. Each file will start at the top of a page.
- The **lp** command prints a banner page at the beginning of each job that identifies the user who sent the job, the job number, and any printing errors that may have occurred. Printing multiple files in one request saves on the number of banner pages printed.
- The **lp** command will normally not make a copy of your file before it is printed. So, if you remove or modify the file before it is printed, it may never get printed or the changes you make may show up in the printed result. To force **lp** to make a copy of your file and use that copy to print, use the **-c** option to **lp**.

WARNING! Removing or modifying files before they are printed may result in unexpected output.

7.2.2 Options to the lp command

The **lp** command supports a large number of options for modifying how your text files are printed:

- Use the **-n** option to print multiple copies of a single file. This command line prints 2 copies of the file *temp*:

```
lp -n2 temp
```

- If more than one printer is configured for your system, use the **-d** option to designate a printer other than the default. The argument to the **-d** option is the name of the desired printer, and it should be entered immediately after the option specifier with no space character before it. This command will send the file *myletter* to the printer named *paintjet*:

```
lp -dpaintjet myletter
```

- Use the **-m** option to have a mail message sent to you when your print job is completed. This option is useful on a busy system with large print jobs, when a delay may occur before your job finishes.

```
$ lp -m my_report this_report
```

```
request id is laser-451 (2 files)
```

```
(some time later...)  
you have mail
```

- The **-t *title*** option prints a title on the banner page of the output

```
lp -t 'Title for my print job' my_report
```

- The **-c** option forces the **lp** command to make a copy of the file(s) to be printed, so that the original may be removed or edited without affecting the printed output
- The **-R** option will remove the file being printed after it has been sent to the printer
- The printer interface script receives arguments appended to the **-o** option to **lp**. Some arguments that may accompany the **-o** option are presented in Section 7.3.1, “Printer formatting features,” on page 7-16.

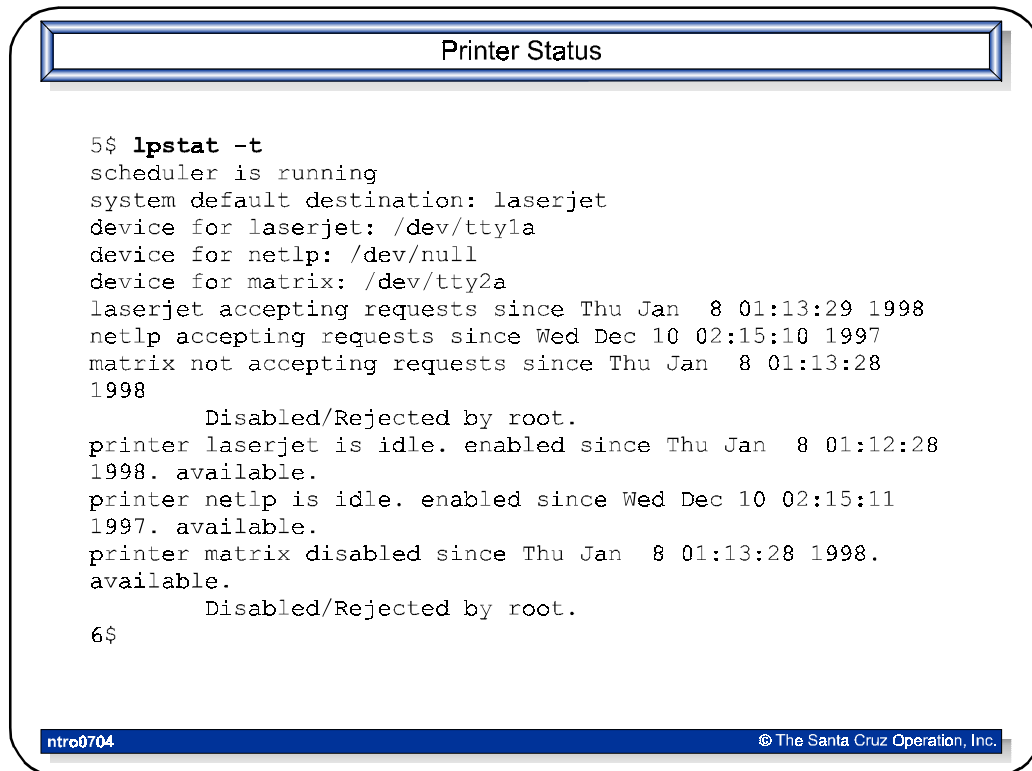
Exercise: Using the lp command

What would this command do?

lp -t 'Month End Report' -n5 report

Print the file */etc/motd* to the default printer and have the spooler mail you a message when it is completed. Read that mail message.

AV 7-4



7.2.3 Monitoring print status

The status of your pending print jobs can be monitored until the job is sent to the printer.

- Use the **lpstat** command with the **-r** option to find out if the print scheduler is running:

```
$ lpstat -r
scheduler is running
```

- If the scheduler is not running, notify your system administrator to configure the system for printing
- Use the **lpstat** command to display all your outstanding print requests:

```
$ lpstat
laser-450          robd          8492          Jun 02 11:23
paintjet-478       robd          781          Jun 02 11:31
```

The output shows the request ID, user name, file size, and time of request. If no print jobs are queued, **lpstat** returns with no output

- Note that with no options, **lpstat** displays information on your print requests only. If you have correct authorization, you can generate a report for all users on your computer using the **-o** option to **lpstat**.
- To find the status of a particular job, specify that job's request ID as an argument to the **lpstat** command:

```
lpstat paintjet-478
```

- The **-p** option to **lpstat** will display status about a particular printer. Adding the **-D** option will include a brief description of the named printer(s):

```
lpstat -p laser paintjet -D
```

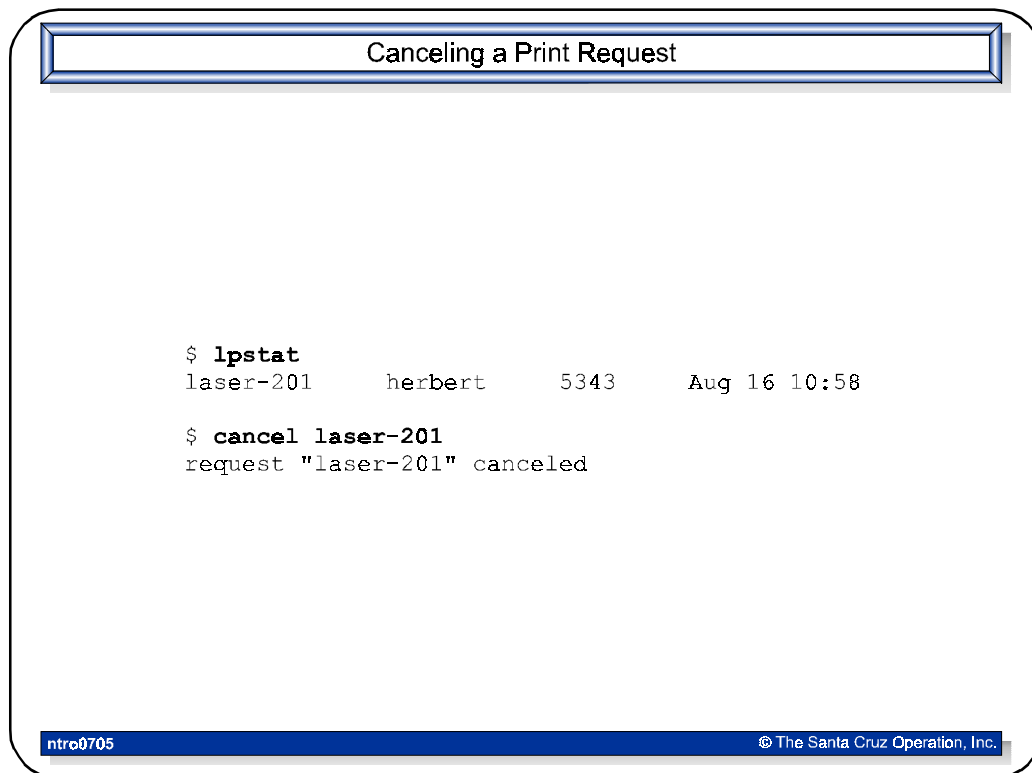
- To get a list of available printers, use **lpstat** with the **-s** option
- With proper authorization, the **-t** option to **lpstat** will provide complete status on the line printer system. These notes clarify some of the output:
 - A printer that has been disabled is currently unable to print jobs; however, jobs are still queued in the waiting list
 - An enabled printer may be idle or actively printing

- An idle printer is available and has no jobs currently pending
- The system administrator has the responsibility of enabling and disabling printers as well as assigning names to the various printers

Exercise: Monitoring print status

Turn your printer offline. Use **lp** to send a request using the file *text.fxd* in your home directory. Use **lpstat** to check the status. Turn your printer online. Does the file get printed?

AV 7-5



7.2.4 Canceling print jobs

If you send a print request to the wrong printer, or you decide to make a modification to the file you want printed, you can cancel a print job that is still in the queue.

- Use the **cancel** command to terminate a pending print request. You must supply the request ID which may be obtained from the **lpstat** command.

```
$ cancel laser-450  
request "laser-450" canceled
```

- Once a job enters the print queue, users can cancel their print jobs

Exercise: Canceling print jobs

Submit a print job, determine its print request ID, and cancel the job.

Can you stop a file from being printed by renaming the file immediately after sending it to **lp**? Try it.

7.3 Preparing Files for Printing

The printer commands enable you to print text files to a number of available printers. UNIX text files are unformatted with the exception of certain control characters, including carriage-returns, linefeeds, formfeeds, and tabs. It is often desirable to provide some formatting of text such as page numbering, page orientation, double spacing, and headers when sending the text to the printer.

7.3.1 Printer formatting features

Various printers support their own formatting features that can be accessed through special command line options to the **lp** command. The **lp** command passes arguments to the **-o** option to the printer interface program for use by that printer. Features offered by various printers might include:

Characters per inch (cpi)	Sets the character pitch (the number of columns per inch)
Length (length)	Sets the page size in number of lines, inches, or centimeters
Double sided printing (double)	Selects double-sided printing
Margins (lm and rm)	Sets the left and right margins
Orientation (land and port)	Sets landscape or portrait page orientations
Banner page (nobanner)	Suppresses printing a banner page with the request
PostScript (postscript)	Selects PostScript printing. PostScript printers generally autodetect postscript files with the sequence <code>%!</code> on the first line
Print mode (raw)	Selects raw printing, which is required to print PostScript files on certain PostScript printers

This command line requests the file `/etc/motd` to be printed in landscape mode, with a left margin of 10 columns, and with no banner page:

```
lp -oland -olm=10 -onobanner /etc/motd
```

Exercise: Formatting text

Use the **-o** option to **lp** to print the file *prop2* so that it prints 25 lines per page.
Hint: Refer to the **lp** manual page for the exact syntax for indicating page length.

7.3.2 Printing data files

In order to print non-text files, such as word-processing documents, spreadsheets, database reports, or graphic presentations, you must use the print facilities available within the applications that generate those data files. They will ultimately rely on the printer system, but they will process the data files to make them manageable by the SCO UNIX system.

7.4 Lesson Summary

SCO UNIX supports spooled printing using the **lp** printer commands. Spooled printing is the most common form of printing on SCO UNIX systems because it offers the efficient use of one or more printers by all the users on the system. The printer commands are easy to use and very flexible. You can print multiple copies of a file, cancel a print request, or ask for special options on a particular printer.

Review questions

1. How would you print three copies of a report to a printer named *Qume3*, if that printer was not the default printer?
2. Give two ways you could find out if a print request you had made was completed.
3. How would you find out if a printer you wanted to use was enabled and accepting requests?
4. How do you cancel a print job?
5. How would you prevent a banner page from printing?

7.5 Lab: Printing

The goal of this lab is to become familiar with the UNIX print commands **lp**, **lpstat**, and **cancel**.

7.5.1 Overview

You will print several files with using **lp** with commonly used options, monitor the status of print jobs and printers, and practice canceling print requests.

7.5.2 Procedure

1. Change to your home directory.
2. List the printers available on your system.
3. Print the files in the directory *Poems/Blake* with one command. What is the request ID?
4. Use the **lp** command with an invalid filename. What happens?
5. Print the file *Edit/information* in landscape mode with no banner.
6. Print *bigfile*. Cancel the print job.

7.6 Relevant Commands

These commands were presented in this lesson.

cancel	Cancels requests to the printer
lp, lpr	Sends a print request to the print spooler
lpstat	Prints information about the status of the lp print service

7.7 Answers to Exercises, Labs, and Review Questions

This section provides suggested answers to the exercises, labs, and review questions.

Exercise: Using the lp command

The command prints five copies of the file *report* along with a banner page bearing the title, “Month End Report”.

Exercise: Monitoring print status

As long as the printer is offline, the print file will remain spooled in the waiting queue. Once the printer is turned online, print jobs will proceed and the file will be printed.

Exercise: Canceling print jobs

With SCO OpenServer Release 5 and UnixWare 7, the **lp** command does not make a copy of the original file; it tracks it by name only. If you rename the file before it is sent to the printer, the file scheduled to be printed will not be found. Versions of SCO UNIX prior to Release 5 make a copy of the file to be spooled, and those copies will be printed regardless of what is done to the original file.

Exercise: Formatting text

To print page lengths of 25 lines using the **-o** option to the **lp** command, you would enter:

```
lp -o length=25 filename
```

Answers to review questions

1. The command would be: **lp -dQume3 -n3 report**
2. To know when a print request is completed, you can periodically issue the **lpstat** command to see the status of your request; it will now show up after it has been printed. Or you can issue the **-m** option to the **lp** command and await notification of the mail message informing you of the completion of the print job.

3. To find out if a printer is enabled and accepting requests, query it with the `-p` option to the `lpstat` command. Use the `-l` option for a full description of the printer's configuration.
4. To cancel a print job, use the `lpstat` command to determine the print job's ID, then use the `cancel` command with the ID number as its only argument.
5. To prevent `lp` from printing a banner page, use the **`-onobanner`** option.

Lab: Printing

This section provides suggested answers to the lab exercises.

- Wildcard characters can be used on the command line to print multiple files using `lp`. To print all the files in the directory *Poems/Blake*, type:

```
lp Poems/Blake/*
```

- Multiple **`lp`** options can be specified to two ways:

```
lp -option1,option2,option3
```

```
lp -option1 -option2 -option3
```

8

Input and Output

AV 8-1

The slide is titled "Input and Output" in a blue-bordered box at the top. Below the title, there is a list of four bullet points. At the bottom of the slide, there is a blue footer bar containing the text "ntro0801" on the left and "© The Santa Cruz Operation, Inc." on the right.

- Identify the three open files an executing program is given when it is started
- Redirect input from files and output to files
- Define the terms pipe, filter, and tee
- Use pipes and tees to combine multiple commands

ntro0801 © The Santa Cruz Operation, Inc.

Lesson 8

Input and Output

Introduction

SCO UNIX commands and utilities commonly get their input from the keyboard and display their output on the terminal screen. For many commands, such as **pr**, it is useful to capture the output and send it to a file or a printer. SCO UNIX systems not only provide these capabilities, but also allow you to chain commands together in such a way that the output of one command becomes the input to another.

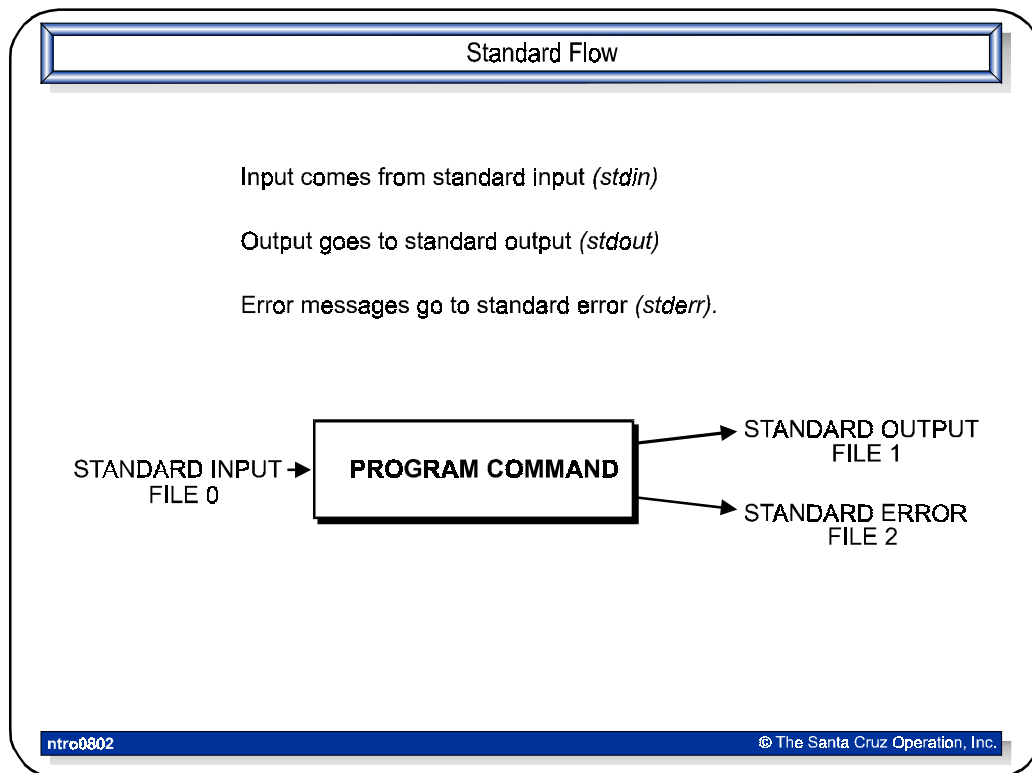
These features not only save you time, but allow you to accomplish tasks that would otherwise not be possible. The ability to control the flow of a program's input and output is provided by the system shells and is called I/O redirection.

Objectives

This lesson covers how to use special shell metacharacters to control the source of a program's input and the destination of its output. By the end of this lesson, you will be able to:

- Identify and explain the three open files an executing program is given when it is started
- Redirect input from files and output to files
- Define the terms *pipe*, *filter*, and *tee*
- Use pipes and tees to combine multiple commands

AV 8-2



8.1 Overview

Almost all activity on a computer involves getting input, processing it in some way, and producing output. The computer has devices for providing input and devices for displaying or storing output. UNIX treats these devices as files.

When a program is invoked, it becomes a process that uses input and output channels. These channels, called *file descriptors*, refer to the files which the process has open for use. When a process starts, it commonly has three file descriptors already opened for it by its parent process (usually the shell): *standard input*, the *standard output*, and the *standard error*.

8.1.1 Standard input

Standard input (*stdin*) is the name of the file descriptor which a program uses for input.

- By default, the standard input refers to your terminal keyboard
- A program reads bytes from your keyboard like it does for any file
- Standard input is identified by file descriptor 0

8.1.2 Standard output

Standard output (*stdout*) is the name of the file descriptor which a program uses for normal output.

- By default, the standard output refers to your terminal screen
- A program writes its normal output to your terminal screen like it does for any file
- Standard output is identified by file descriptor 1

8.1.3 Standard error

Standard error (*stderr*) is the name of the file descriptor which a program uses for error or diagnostic messages.

- By default, the standard error refers to your terminal screen
- A program writes its error messages to your terminal screen like it does for any file
- Standard error is identified by file descriptor 2

Exercise: Standard device files

Enter the **cat** command without any arguments. Enter several lines of text. What happens when you press ? Where does the input come from?

Enter  **d** on a line by itself to exit the program and return to your shell.

AV 8-3

Redirection Syntax

```
command < file  
command > file  
command >> file  
command > file 2>&1
```

ntro0803 © The Santa Cruz Operation, Inc.

8.2 File Redirection

The flow of command input and output can be redirected so that:

- Input comes from a regular file or another device instead of the keyboard
- Output goes to a regular file or another device instead of the terminal screen

The UNIX shell programs, such as the Bourne shell (**sh**), C shell (**csh**), and Korn shell (**ksh**), allow you to redirect the standard input, standard output, and standard error. The shell gets I/O redirection information from special syntax that you enter on a command line.

It is important to note that these shells set up the I/O redirection for a command *before* the command even begins to execute. As a result, the executing command is unaware that any redirection has taken place. If *stdout* had been redirected, the command still writes to it as before, but unbeknownst to it, *stdout* will refer to a file other than the terminal.

8.2.1 Redirecting output

Redirecting output allows you to capture the output of a command and save it in a file, rather than see the output on the terminal screen and lose access to it.

Recall that commands normally have two output file descriptors—*stdout* and *stderr*—and these both usually refer to the terminal screen. Why are there two file descriptors for one device? The reason is that *stderr* usually doesn't get redirected. Therefore, any error or diagnostic messages will go to the default location (i.e., the terminal) where it is more likely to be noticed by the user. If you do want to redirect *stderr*, however, the shell has the flexibility to do that, too.

8.2.1.1 Redirecting the standard output

Standard output is redirected using the (**>**) shell metacharacter on a command line:

```
command > filename  
command 1> filename
```

- The name of the file to which you are directing the output must be specified immediately after the redirect symbol. The separating space is not required.
- You may specify a file descriptor number before the **>** symbol (no spaces permitted). If omitted, the default is file descriptor 1 (the standard output).

- The file redirection can occur anywhere on the command line as long as the redirection symbol and the filename are kept together. By convention, however, I/O redirection syntax is usually specified at the end of a command line.

For example, these command lines are equivalent; they concatenate the contents of */etc/systemid* and */etc/motd* and put them into a file named *greetings*:

```
cat /etc/systemid /etc/motd > greetings  
cat > greetings /etc/systemid /etc/motd
```

- If the redirected file does not exist, it will be created by the shell first, and then the output will be redirected into it
- If the file already exists (and you have the appropriate permissions), the old contents of the file are destroyed and replaced with the output of the executing command
- I/O redirection can be specified on a command line that has no command. The shell will create a zero length file or delete the contents of an existing file:

> *filename*

Exercise: Redirecting the standard output

Redirect the output of the **cal** command to a new file with the name of the current year. Try placing the redirection in different places on the command line, with and without spaces around the **>** symbol. Does it make a difference?

8.2.1.2 Redirecting the standard error

- *stderr* is redirected the same way as *stdout*, using the **>** metacharacter:

```
command 2> filename
```

- Recall that you may specify a file descriptor number before the **>** symbol (no spaces permitted). If omitted, the default is file descriptor 1 (the standard output), so you *must* specify file descriptor 2 to redirect *stderr*.
- Sometimes it is useful to save error messages to a file for future reference, for example, when troubleshooting programs

8.2.1.3 Combining output

- Both the standard output and standard error can be redirected on the same command line:

```
command 1> output_file 2> error_file
```

- To send both *stdout* and *stderr* to the same file, a special notation must be used:

```
command > all_output_file 2>&1
```

The `2>&1` notation means “redirect *stderr* (`2>`) to wherever *stdout* is being redirected (`&1`)”.

- You should not specify the same filename to both redirection symbols. That would use two independent output file descriptors for the file, causing the output from one of them to overwrite data already written by the other. The `2>&1` notation allows one descriptor to control the output instead.

Exercise: Combining outputs

Run the command:

```
find /usr > usr_files 2>errors
```

Were any errors reported?

What does this command do?

```
find . 2> error_file >&2
```

What do you observe when you execute these command lines:

```
ls -l letter bigfiel timecal  
ls -l letter bigfiel timecal > output1  
ls -l letter bigfiel timecal 2> output2  
ls -l letter bigfiel timecal > output3 2> output4  
ls -l letter bigfiel timecal > output5 2> output5  
ls -l letter bigfiel timecal > output6 2>&1  
ls -l letter bigfiel timecal 2> output7 1>&2
```


8.2.1.4 Appending to files

Normally, redirecting output has a destructive effect on existing files. The shell provides a way to add new data—*append*—to an existing file.

- The append symbol (`>>`) allows you to add output to the end of a file:
- To append *stdout* to an existing file, use the syntax shown by either of these command lines:

```
command >> filename  
command 1>> filename
```

- To append *stderr*, use the syntax:

```
command 2>> filename
```

- If the file designated by *filename* does not exist, it will be created

Exercise: Appending data

Append the output of the **date** command to the file in which you saved the calendar output.

8.2.2 Redirecting input

Redirecting input allows a program to get its standard input from a named file rather than from the (default) keyboard.

- *stdin* is redirected using the `<` shell metacharacter on a command line:

```
command <filename  
command 0< filename
```

- You may specify a file descriptor number before the `<` symbol (no spaces permitted). If omitted, the default is file descriptor 0 (the standard input).

- Redirecting *stdin* is less common than redirecting *stdout* or *stderr* because most commands open files (with other file descriptors than 0, 1, or 2) themselves. When programs read from only *stdin*, this redirection may be particularly useful.

As an example, this UNIX command line mails the *schedule* file to the user **susan**.

```
mail susan < schedule
```

These examples clarify the use of input redirection.

- In the first example, **cat** opens the file *textfile* and a new file descriptor is allocated. In the second example, **cat** uses *stdin*, which has been changed by the shell to refer to *textfile*.

```
cat textfile  
cat < textfile
```

- In the first command line, **more** opens the file *myfile* and a new file descriptor is allocated. In the second example, **more** uses *stdin*, which has been changed by the shell to refer to *myfile*.

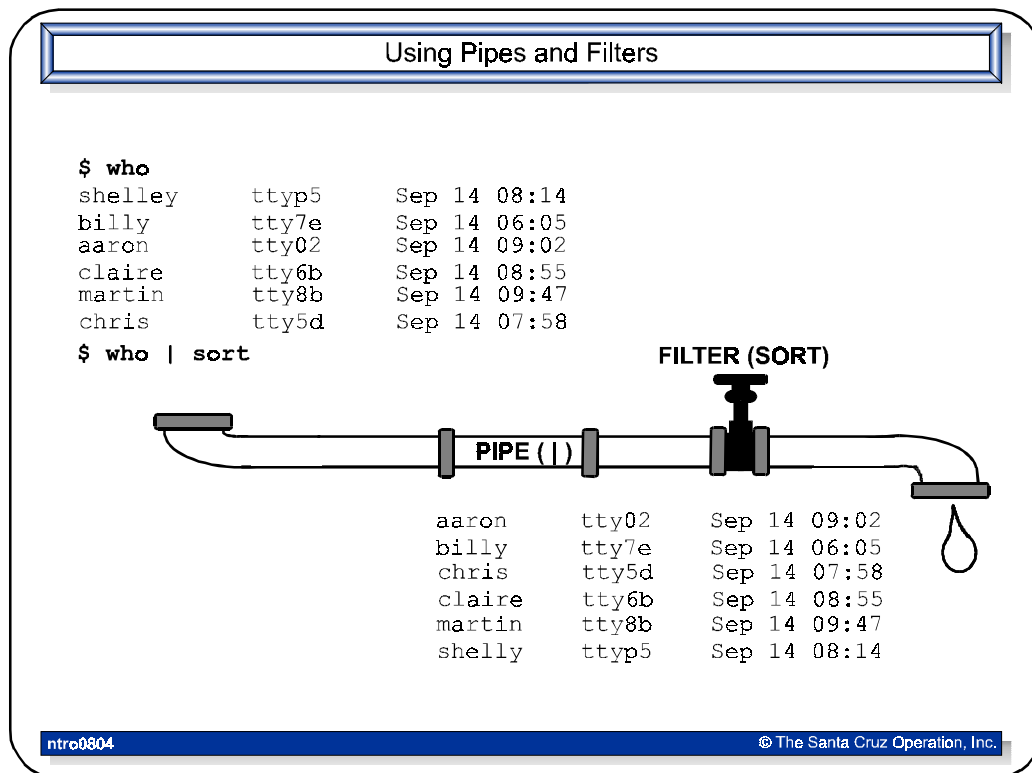
```
more myfile  
more < myfile
```

The difference in operation is apparent when **more** opens a file versus using *stdin*. For example, the status line at the bottom of the screen will not show a filename or the percentage of the file viewed when **more** reads from *stdin*.

Exercise: Redirecting the standard input

Mail your classmate a copy of *sonnet5*.

AV 8-4



8.3 Pipes, Tees, and Filters

Further flexibility in controlling the flow of input and output is provided by the shell pipe construct and by the SCO UNIX **tee** command.

8.3.1 Pipes

You can make the standard output of one command become the standard input of another command by using the shell construct called a *pipe*. Whereas file redirection directs input and output to files, pipes direct the flow of input between processes.

- The pipe symbol (`|`) redirects the standard output of ***command1*** to be the standard input of ***command2***. (An alternate pipe symbol is a caret (^), but it is now mainly used for backwards compatibility.) The syntax is:

```
command1 | command2
```

- In order for a pipe to work, ***command1*** must write its output to *stdout*, and ***command2*** must read its input from *stdin*. You cannot use an interactive program such as **vi** or **email** within a pipeline
- *stderr* cannot (easily) be redirected through a pipeline
- Each command in the pipeline must be completely specified and include all the options and arguments that it requires to run
- With pipes, the step of creating a file to store the output of a command is unnecessary. The output from one command can be directed to the input of another with a pipe:
 - These commands print out two copies of the file *letter*, formatted to fifty lines a page, using a temporary file to accomplish the task:

```
pr -150 letter > tmpfile
lp -n2 tmpfile
rm tmpfile
```

- This command line does the same thing using a pipeline:

```
pr -150 letter | lp -n2
```

8.3.2 Filters

Commands that can read from *stdin* and write to *stdout* are called *filters*.

- Filter programs can be combined with other programs using pipes in the command line to form what is called a pipeline
- Commands that do not read the standard input (for example, **date**) or do not write to the standard output (such as **exit**) are not filters, nor are interactive programs like **mail** or **vi**.
- Many filter commands are designed to be able to read from *stdin*, but only if no other filename is provided on the command line. If a file is specified, it opens that; otherwise, it defaults to using *stdin*. This allows it to be used in a pipeline.
- Many of the commands you have used are filters, such as **head**, **tail**, **cat**, **hd**, and **spell**
- Not all commands in a pipeline need be filters. The first and last commands do not need to be, but all other commands in the pipeline do.

8.3.3 The tee command

The **tee** command is a filter command that reads the standard input and writes it to both the standard output and a specified filename.

```
tee [ options ] [ file ... ]
```

- The **tee** command does not alter its input in any way; it simply outputs two or more copies of it—one to *stdout*, all others to the named files
- The **tee** command creates any named files that do not exist, and destroys the contents of any files that do. Use the **-a** option to append the output instead.
- Use the **tee** command to collect intermediary results of a long pipeline. This command line prints the names of files that are not English words on the screen and saves the entire directory listing in the file *filelist*:

```
ls | tee filelist | spell
```

8.4 More Filter Commands

Many of SCO UNIX commands are filters that default to reading from *stdin* when no filenames are specified on the command line (for example, **cat** and **spell**). In this section, three other useful filters are introduced: **wc**, **sort**, and **tr**.

8.4.1 The **wc** command

The command **wc** counts the number of lines, words, and characters in a file:

```
wc [ -l | -w | -c ] [files ... ]
```

- Lines are determined by the number of newline characters
- Words are any string of characters separated by newlines, spaces, or tabs
- Characters are the number of bytes in the file

wc is often used at the end of a pipeline to count some information. This command counts the number of misspelled words in the three files and displays just the count on the terminal screen:

```
cat file1 file2 file3 | spell | wc -w
```

AV 8-5

Extended Commands

```
$ grep 'john' employees | sort | pr | lp
```

Search the file *employees* for 'john,' alphabetize the data, format the data for the printer, and print the result.

```
$ grep 'john' employees | sort | tee johns | pr | lp
```

Search the file *employees* for 'john,' alphabetize the data, save the sorted data in the file *johns*, format the data for the printer, and print the result.

ntro0805© The Santa Cruz Operation, Inc.

8.4.2 The sort command

The **sort** command arranges data in alphabetical or numerical order and displays the results on the standard output:

```
sort [ options ] [ files ... ]
```

- There are numerous options that control the way sorting is done. By default, **sort** treats each line in the file as a record and rearranges the lines alphabetically, starting at the beginning of the line. Refer to the **sort** manual page for more details.
- As with any filter, **sort** does not modify the input file; it simply writes to *stdout*. To save the sorted output, redirect it to a file or pipe it to another process.
- This command line sorts the contents of the *fruit* file, puts the sorted list in the *salad* file, and displays the number of lines sorted:

```
sort fruit | tee salad | wc -l
```

- This command line sorts the names of all the users currently logged on the system and puts the sorted output in the newly created file, *wholog*:

```
who | sort > wholog
```

8.4.3 The tr command

The **tr** command is a useful command for translating specific characters within a text file. It has no filename command line arguments; its input always comes from *stdin* and its output always goes to *stdout*:

```
tr [ options ] string1 string2
```

- The **tr** command substitutes each character in *string1* with the corresponding character in *string2* in whatever it reads from *stdin*. The number of characters in *string2* should normally be the same as in *string1*.
- This command will turn all vowels to upper case in the file *letter*:

```
tr aeiouy AEIOUY < letter
```

- The *string* arguments may be single characters or ranges of characters:

```
who | tr a-z A-Z
```

- The **tr** command can also strip certain characters from a file:

```
tr -d string1 < inputfile > outputfile
```

- The **tr** command will delete all characters in *string1* from the input. This command line will remove all carriage-returns (`\r`) from a text file. This is a useful way to convert a DOS text file to a UNIX text file:

```
tr -d '\r' < dosfile.txt > unix_file
```

8.4.4 The **pr** command

The **pr** command formats text files to include page numbers, page headers (which includes the file's modification date and time) and numerous other features as specified by the command line options. The syntax is:

```
pr [ options ] file [file ... ]
```

The options control such features such as page length, line width, double spacing, and multi-column output. Some options include:

-d Double-spaces the output

-h Uses the next argument as the header to be printed instead of the filename. For example:

```
pr -h 'New top of the page header' myfile
```

-nck Provides *k*-digit line numbering (default=5). If *c* (any non-digit character) is given, it is appended to the line number to separate it from whatever follows (default=tab).

-o k Offsets each line by *k* character positions (default=0)

The **pr** command sends output to the standard output (usually the screen). Often, you will want to redirect this output to a file or to another command, such as **more**.

8.5 Lesson Summary

When a program begins execution it is allocated three open files: the standard input (*stdin*), standard output (*stdout*), and standard error (*stderr*). These special file descriptors usually refer to your terminal screen or keyboard. The UNIX system shells provide special metacharacters that allow these file descriptors to be redirected to files or piped to other running programs. This overall mechanism is called I/O redirection, which gives you much flexibility in controlling the I/O of your commands to accomplish very useful tasks.

Review questions

1. How does an executing program use the standard input, standard output, and standard error files?
2. What is the difference between the symbols `>` and `>>`?
3. What are the requirements of a program in order for it to be called a *filter*?
4. Describe two ways that the **sort** command can get its input for sorting.
5. What does this command line do:

pr my_file | lp
6. What is the difference between the terms *pipe* and *tee* with respect to input/output redirection?

8.6 Lab: Input and Output Redirection

The purpose of this lab is to gain proficiency in using I/O redirection to perform tasks on the system.

Overview

You will combine commands you have learned in this course using shell redirection, pipes, and tees to perform a variety of tasks on the system.

Procedure

Make sure you are in your home directory.

1. Use the **date** command to record the date and time into a file called *users* in your home directory. Verify the results by displaying the file on your screen.
2. Append a listing of who is currently logged into the system to the file. Again, verify your results.
3. Use the **banner** command and a pipe to mail yourself the message, “Good Work”. Retrieve the message to check the results.
4. Change directory to your *Miscellaneous* subdirectory.
5. Sort the contents of the *fruit* file and display the results on the screen.
6. Do the same task, but send the results to the printer and retain a copy of the output for yourself. Try to do this all in one command.
7. Use the **pr** command to double-space the contents of your sorted *fruit* file, and print the results.
8. Run the **pr** command with the option, **-3 -l24**, using the *fruit* file as input. What do the options to **pr** do?
9. Change back to your home directory.
10. The **grep** command from Lesson 3, “Files and Directories”, is a filter that finds string patterns in files or from the standard input. Find out how many times the word “love” appears in Shakespeare’s sonnets.

11. Use the **tr** command to change all periods in the *letter* file to exclamation points. Can you think of a way to use the **tr** and **wc** commands to count how many exclamation points are now in the file *letter*? (Hint: `\n` represents the newline character.)

8.7 Relevant Commands and Syntax

The following commands and syntax were presented in this lesson.

Relevant commands

pr	Formats text files
sort	Sorts and merges text files
tee	Creates a tee in a pipe
tr	Translates characters in text files
wc	Counts words, lines, and characters in text files

Relevant syntax

< or 0<	Redirects <i>stdin</i>
> or 1>	Redirects <i>stdout</i>
>> or 1>>	Appends <i>stdout</i> to the end of a file
2>	Redirects <i>stderr</i>
2>>	Appends <i>stderr</i> to the end of a file
2>&1	Redirects <i>stderr</i> to the same file as <i>stdout</i>
 or ^	Pipes <i>stdout</i> from one command to <i>stdin</i> of another

8.8 Answers to Exercises, Labs, and Review Questions

This section provides suggested answers to the exercises, labs, and review questions.

Exercise: Standard device files

When the **cat** command is invoked without any arguments, it displays the characters you have typed from the keyboard each time you press .

Exercise: Redirecting the standard output

It does not matter where the redirection symbol is entered on the command line as long as the desired file is the next argument following it.

Exercise: Combining outputs

The errors were reported in the *errors* file; no output came to the screen

The following command would put any error messages along with the pathnames of all files in and below your home directory into the file *error_file*:

```
find . 2> error_file >&2
```

The **ls** commands generate output to both the standard output and the standard error. Unless redirected, these refer to the default screen. The following lists the output destinations of each command line:

stdout=default screen; *stderr*=default screen

stdout=*output1*; *stderr*=default screen

stdout=default screen; *stderr*=*output2*

stdout=*output3*; *stderr*=*output4*

stdout=*output5*; *stderr*=*output5*, but output will not be correct

stdout=*output6*; *stderr*=*output6*, and output will be correctly combined

stdout=*output7*; *stderr*=*output7*, and output will be correctly combined

Answers to review questions

1. An executing program uses the standard input to read input from the keyboard or from some other file redirected by the user. It writes to the standard output, and any error messages generated are sent to standard error.
2. The symbol (>) creates the file with the name given by the following argument. If the file already exists, it destroys its current contents before copying the output of the program to that file. The symbol (>>) directs the shell to append the output of the program to the file, thereby not destroying the existing contents of the file. If the file does not already exist, it is created.
3. To be a filter, a program must read its input from stdin and write its output to stdout.
4. The sort command can get its input from a file specified as an argument on the command line, or it can read the stdin (from the keyboard by default), which may be piped from the output of another command.
5. This command sends the formatted output of my_file to the lp command, which then sends it to the printer. Note that if you had used the (>) symbol instead of the pipe symbol (|), you would not have achieved the desired result. Instead, you would have created a file called lp in your current directory, and it would contain the formatted text of my_file.
6. The tee command reads the standard input and produces two copies of that input for output. One it directs to the standard output; the other it copies to the file named as the argument to the command. A pipe is not a separate process, but rather a shell mechanism that directs the standard output of one process directly to the standard input of another process.

Lab: Input and Output Redirection

This section provides suggested answers to the lab exercises in Lesson 8.

An example of a pipeline shows how a file is sorted, saved to a file, and printed, using a single command line:

7. **sort fruit | tee -a alphafruit | lp**

9

Process Control

AV 9-1

The slide is titled "Process Control" in a blue-bordered box at the top. It contains a list of six bullet points in the center. At the bottom, there is a blue footer bar with the text "ntro0901" on the left and "© The Santa Cruz Operation, Inc." on the right.

Process Control

- Define a UNIX process
- Explain how processes are created and how they terminate
- View status information on your processes
- Run processes in the background
- Use signals to terminate processes
- List two ways of distributing process usage

ntro0901 © The Santa Cruz Operation, Inc.

Lesson 9

Process Control

Introduction

UNIX provides a wide variety of commands that you can execute. Part of what makes UNIX powerful is the fact that you can run multiple commands at the same time.

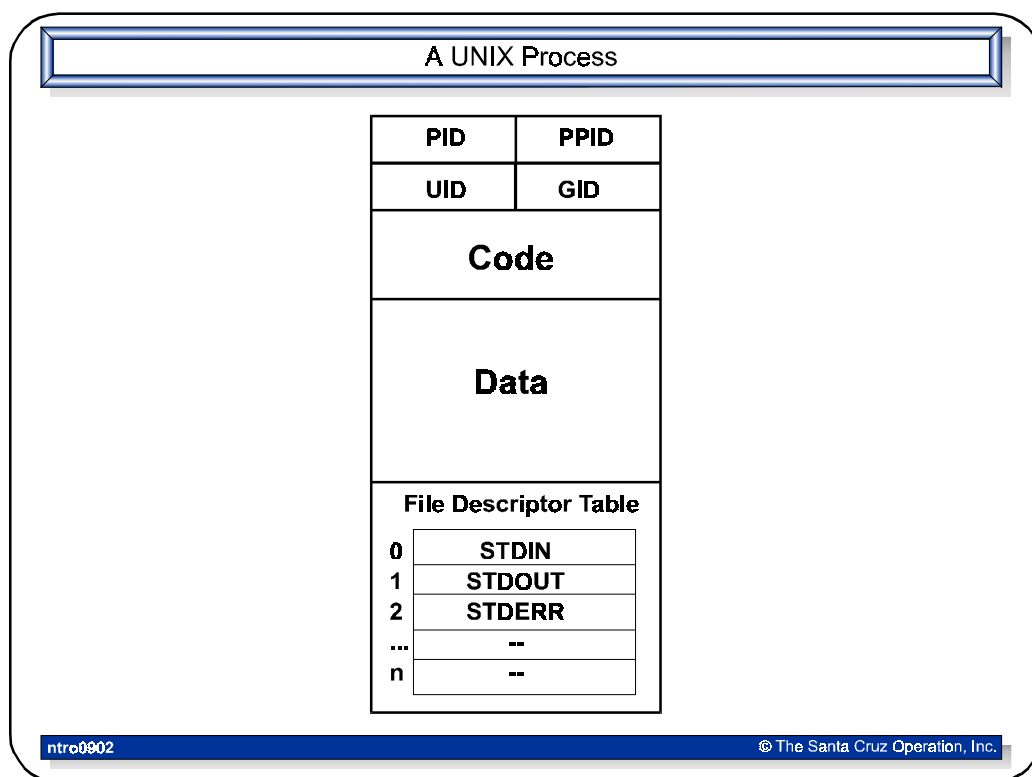
Learning how to run multiple tasks, check on their status and terminate them if they malfunction will help you use the system more efficiently and become more productive. In this lesson you will learn to use SCO UNIX commands to create, monitor, and terminate UNIX processes.

Objectives

This lesson covers how to view, monitor, and control processes. By the end of this lesson, you will be able to:

- Define a UNIX process
- Explain how processes are created and how they terminate
- View status information on your processes
- Run processes in the background
- Use signals to terminate processes
- List two ways of distributing process usage

AV 9-2



9.1 Overview of Processes

A *process* is a program in action. What is a program? It is a file, like nearly everything else in UNIX. Every time you execute a command (that is not built-in to the shell), you create a new process. When a command completes its task, the process terminates. Both users and system programs can create processes which affect the overall performance of the system.

9.1.1 Definition of a process

- A process is an instance of a program that is currently executing on the system. The program file has been loaded into memory and the kernel has allocated system resources to it in order for it to run.
- A process is allocated a section of memory which holds its instructions and whatever data it requires to execute. Processes have a size associated with them that may grow as the process runs.
- A process is generally allocated three open files, *stdin*, *stdout*, and *stderr*, which are in a table of file descriptors that allow it to open a limited number of additional files
- While many processes may exist at one time, the CPU is actively attending to only one process. All other processes are queued up, waiting their turn to have their instructions carried out.
- Processes have a priority value associated with them that influences when they can be scheduled to use the CPU
- Processes are owned by the user who created them and the terminal device, if any, from which they were created
- Processes are created by other processes. The creating process is called the *parent*; the one created is the *child*. Every child process has exactly one parent. This creates a hierarchy of processes not unlike the file system hierarchy.
- Processes are identified by a unique process identification number (PID) that the system uses to keep track of them. The system also keeps track of the parent PID of any process, known as the PPID.

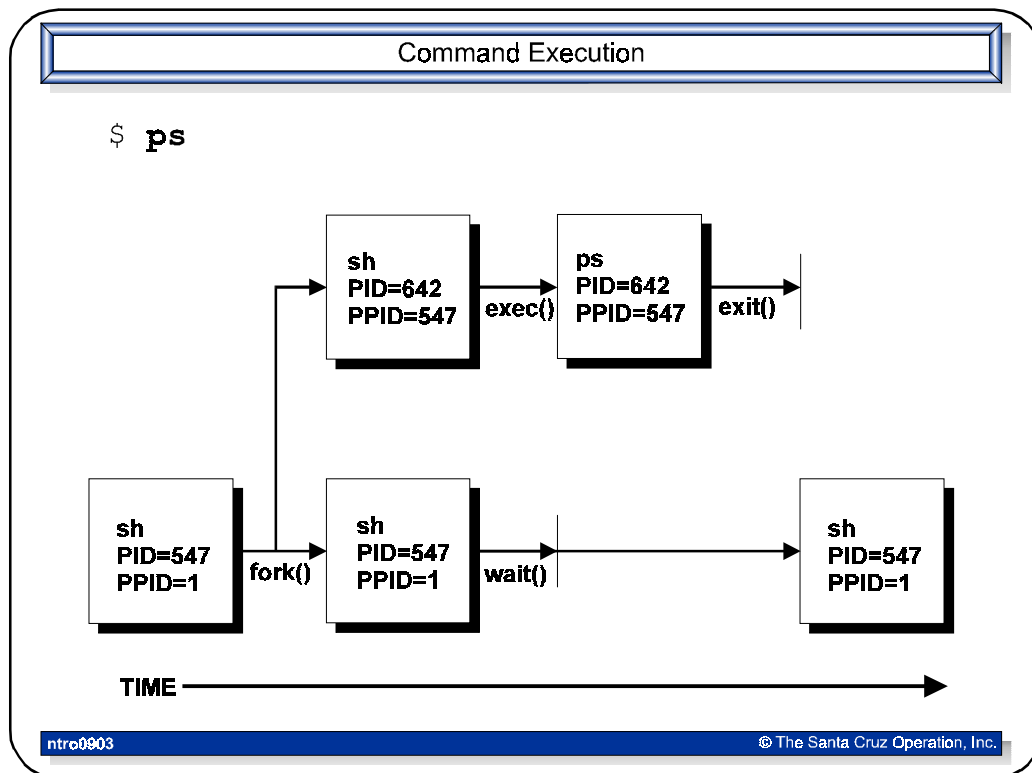
Exercise: Viewing processes

Use the **whodo** command to list what processes are currently being run by users on the system.

whodo

What shell are you running? Note the unique PID values for each process.

AV 9-3



9.2 The Life Cycle of a Process

The SCO UNIX kernel has one system call that creates a new process. Shells use this call to create new processes to run the commands that you enter.

9.2.1 Spawning processes

Creating a process, also called *spawning* or *forking* a process, involves:

- The parent process executes the **fork** system call which creates a new process that is nearly identical to the parent. This child, like all processes, has a unique PID.
- The child process typically overlays its current program (e.g., the shell) with a new program (e.g., **ps**) by calling the **exec** system call. This child process will then be running a new *program*, but it is still the same *process* as before.
- The child eventually terminates by calling the **exit** system call. This causes the process to “die,” returning nearly all of its resources back to the system (the kernel process table entry remains).
- In the meantime, the parent usually issues a **wait** call, which suspends its operation until its child has died

9.2.2 The life of a process

A process lives in computer memory, but does not continuously execute on the CPU. At times, it gives up the CPU while it waits for system events such as I/O completion, death of child, or available buffers. The kernel may also *preempt* a process to give other processes an opportunity to run.

A process's life consists of having its program instructions (its *code*) executed in short time intervals called *time slices*. During that time, the process may allocate more resources and access the computer hardware as needed.

Part or all of the memory of a process may get copied out to a special part of the hard disk to make room for other processes—an action called *swapping*.

9.2.3 Terminating processes

A process ends under one of two conditions:

- Programs terminate with an **exit** system call. Programs that read from the standard input usually exit shortly after they reach an end-of-file. The `CTRL d` character represents an end-of-file, and that is why typing `CTRL d` terminates most processes that read from *stdin*.
- A process may be terminated by the operating system as a result of a signal being sent by a user or by the operating system itself. The operating system uses signals to communicate events that can happen at any time, such as the user pressing the `DEL` key, a user logging out, or the power being turned off.

When you execute a command such as **date**, your shell spawns a child process which executes the **date** program (the instructions are in the */bin/date* program file). After this process has finished writing the date and time to the standard output, it makes the **exit** call to terminate itself. The parent (your shell) waits for its child process to terminate before prompting you for the next command.

AV 9-4

Using ps			
Cumulative execution time for the process			
Controlling terminal for the process			
Process ID number			Command name
PID	TTY	TIME	COMMAND
389	5f	0:23	cs
470	5f	0:02	vi
482	5f	0:01	ps

ntro0904 © The Santa Cruz Operation, Inc.

9.3 Viewing Process Status

SCO UNIX allows you to view the status of your processes. You must be properly authorized to view all processes on the system.

- Use the process status command, **ps**, to see a snapshot of the processes you are running at this time:

```
ps [ option [ arguments ] ... ]
```

- The **TIME** field tells you the cumulative CPU time (in minutes and seconds) used by that process. Notice that although you may have been logged in for a while, the shell generally uses a very small amount of actual CPU time.
- Use the **-e** option to display all the processes running on the system
- Use the **-f** option to **ps** to get more information about your processes, such as:

UID	The login name of the user that started the process
PPID	The parent process's PID
C	An estimate of the recent CPU usage by the process
STIME	The time the process was started
CMD	The command that executed the process

- Use the **-l** option to get a long listing of additional process attributes
- Use the **-u login** option to display the process being run by a particular user
- Use the **-p PID** option to display the status of a particular process

Exercise: Viewing process status

View the current status of your processes with the command:

ps -f

Note that the **ps** command itself is listed, since it was running when the snapshot was taken.

What is the parent process of **ps**? What process started your login? How many processes are running on the entire system?

9.4 Process Control

In addition to the commands provided for controlling processes, the UNIX system shells offer ways for you to control how your processes run.

9.4.1 Background processes

Your shell process is the parent process of most commands that you execute. It normally waits for its child processes to terminate before prompting you for the next command. You can tell the shell not to wait for a child, but to continue accepting additional commands by running a process in the *background*.

- To run a command in the background, append the ampersand (&) metacharacter to the end of a command line. The shell will return with a prompt immediately:

```
$ command &  
[1] 586
```

- The Korn shell displays a job number and the child process ID when you execute a command in the background. The PID can be used to monitor the background process with the **ps** command.
- Background execution is useful for commands that take a long time to finish. It is not appropriate for interactive programs such as **vi**.
- When you run a command in the background, you should redirect the command output to a file; otherwise, it will appear on the screen at unpredictable times
- The shell informs you when the background process terminates. Rather than showing the PID, it simply gives the job number of the child process:

```
[1] + Done                               command &
```

Note: the shell displays this message at the time it displays your command line prompt. If your terminal is idle, you won't see it until the shell displays the next prompt. You can force this action by pressing .

Exercise: Background processes

Note the PID number displayed after running the command:

```
find / > /dev/null 2>&1 &
```

Run the **ps** command. How will you know when **find** is done?

9.4.2 Job control

The Korn shell (**ksh**) has a feature that enhances your ability to control processes beyond those of the other shells. This *job control* feature allows you to suspend running process and move processes from the background to the foreground and vice versa.

- The Korn shell supports a special character (**CTRL** **Z**) that *suspends* the currently running child process and tells the shell not to wait for it to terminate. When a child process is suspended, the job number of the child and the name of the command are displayed:

```
[1] + Stopped                  command
```

- Once you suspend a process you can run it in the background with the **bg** command:

```
bg [ job_ID ... ]
```

- You can bring it back to execute in the foreground with the **fg** command:

```
fg [ job_ID ... ]
```

- The **jobs** command gives you the status of all the child processes you have running:

```
jobs [ job_ID ... ]
```

- A *job_ID* can be either the PID or the job number preceded by a % symbol
- You can use the **sleep** command, which sleeps for the number of seconds specified, to try out job control with your Korn shell

Exercise: Job control

Enter the commands:

```
sleep 60 &  
sleep 100 &  
sleep 30
```

Now enter CTRL **z**. Note the number of the stopped process and the fact that you have your shell prompt back.

Run the **jobs** command to see the status of the child processes.

Use the **bg** command to put the 30-second sleep in the background.

Use the **fg** command to put the 60-second sleep in the foreground.

When the 60-second **sleep** is done, run the **jobs** command. What do you expect?

AV 9-5

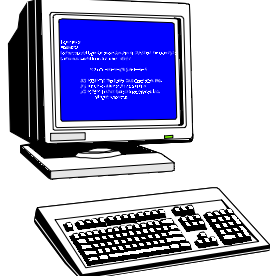
Hung Processes

Dead Terminal



1. `DEL`
2. `CTRL`
3. Cycle power

Live Terminal



1. `ps -t tty`
2. `kill pid`
3. `ps -t tty`
4. `kill -9 pid`

ntro0905© The Santa Cruz Operation, Inc.

9.5 Sending Signals

Signals are events that can interrupt a process at any time and cause it to take an immediate action. The SCO UNIX operating system supports many different kinds of signals, most of which are designed to terminate processes.

Occasionally the instruction a process is running will cause it to get stuck and prevent it from exiting normally. Such a process is said to be *hung*. Whether a process is hung, or you started a process that you suddenly realize you don't want to run, there are times when it is necessary to terminate a process prematurely.

9.5.1 Signals

9.5.1.1 The hangup signal (signal 1)

The operating system sends the *hangup* signal to the child processes of a shell when you log out of that shell.

Signal 1 was originally developed for dialup modems to kill a login process when the phone line was hung up.

9.5.1.2 The interrupt signal (signal 2)

The `DEL` key is generally configured to send the *interrupt* signal which will usually terminate the current process. To stop your process at any time, first try pressing the `DEL` key.

9.5.1.3 The quit signal (signal 3)

- The `CTRL\DEL` key combination sends the *quit* signal to a process which will normally cause it to terminate. Some applications are written to ignore the interrupt signal sent by the `DEL` key. If the `DEL` key does not terminate the process, try `CTRL\`.
- This usually creates a core dump file named *core* or *core.pid*. Unless you have a debugger available, this file is probably useless to you and should be deleted to save disk space.
- Background processes are not affected by either `DEL` or `CTRL\`. For these and other processes that ignore both `DEL` and `CTRL\`, you must use the **kill** command.

9.5.1.4 Other signals

Other signals can be sent to processes under different circumstances. Some of these signals include:

- The *software termination* signal (signal 15)
- The *kill* signal (signal 9). This signal is special in that the kernel does not permit a process to trap or ignore it, therefore it will always trigger the action of termination. For this reason, it is often referred to as the “sure kill” signal.

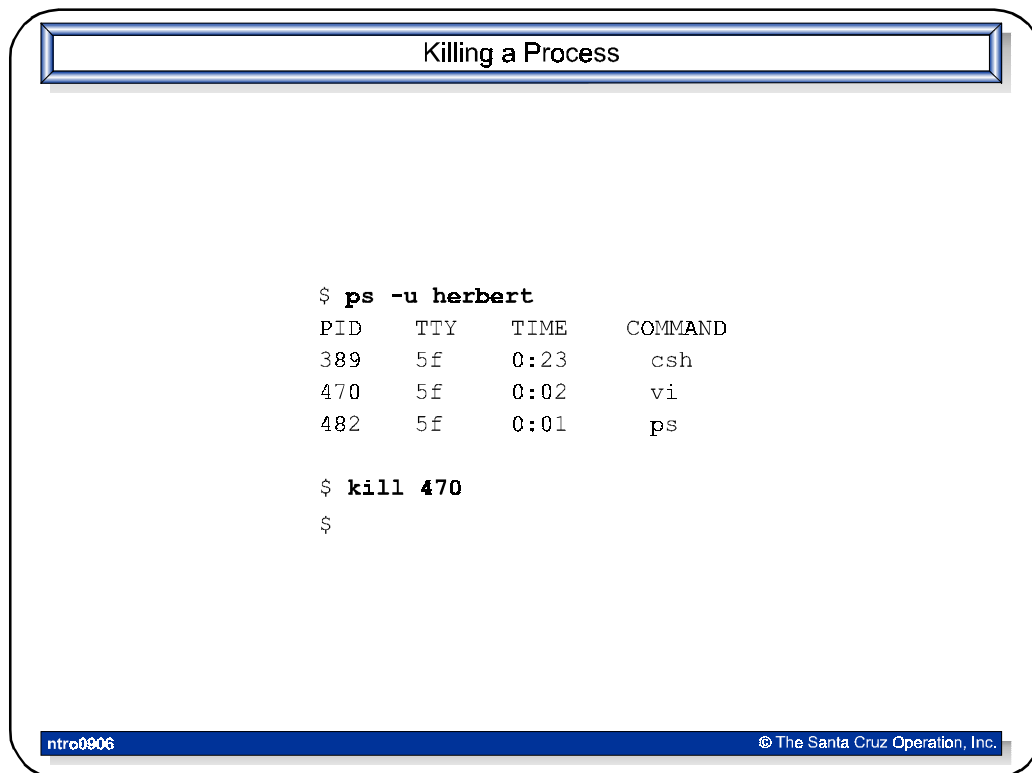
9.5.2 The **nohup** command

If you wish to run a long process in the background that you don’t want to terminate when you log out, you can protect that process using the **nohup** command:

```
nohup command [ arguments ... ] &
```

The **nohup** command redirects the output of the specified command to the file *nohup.out* if output redirection is not otherwise specified.

AV 9-6



9.5.3 Sending signals

9.5.3.1 Keyboard signals

Some signals are easily generated at the keyboard. These key sequences are usually configured to generate signals:

- The `DEL` key sends the *interrupt* signal (signal 2)
- The `CTRL \` key combination sends the *quit* signal (signal 3)

9.5.3.2 The kill command

Some signals are not easily generated with keystrokes. To send these signals you can use the **kill** command.

```
kill [ -signal ] pid [ pid ... ]
```

- The **kill** command can take any signal number as an option. If none is specified, it uses a *termination* signal (15) which can be trapped by a process, allowing it to clean up before it terminates.
- **kill** takes process ID numbers as its arguments. You must be the owner of these processes to signal them with **kill**.
- If **kill** is successful, the shell prints a termination message on your screen
- Remember that signals may force applications to terminate prematurely. This could cause negative side-effects, such as data loss or not setting a terminal back to its normal state. Signals should be sent with caution. Use the *kill* signal (9) as a last resort because processes cannot trap it to clean up first.
- For example, to use the **kill** command to send the kill signal (9) to your process with PID=483, enter:

```
kill -9 483
```

Exercise: Killing processes

Execute the command:

```
find / -name core -print > /dev/null 2>&1 &
```

Run **ps** to verify that **find** is running, then kill the **find** process with the **kill** command.

9.6 Process Scheduling

UNIX provides commands that enable a system to run more efficiently by spreading the process load out over a greater time period. Two of these commands are presented here: one affects the priority value of a process; the other allows the delayed execution of commands.

9.6.1 The **nice** command

Every process has a priority value that the kernel uses in scheduling CPU time. The priority value is based upon how long it has been since the process last had the CPU and a *nice* value between 0 and 39. The higher the nice value, the lower the priority that command will have. A process with a low priority has a reduced likelihood of being scheduled to get the CPU the next time the kernel performs a process context switch.

- The **nice** command is used to increase the nice value for a specific command:

```
nice [ -n increment ] command [ arguments ... ]
```

- It is considerate to use the **nice** command on a busy system, especially when you are about to execute a command that will take a lot of CPU time
- By default, processes have nice values of 20. The **nice** command allows you to increase that value up to the value of 39. For example, this command gives the **find** process a nice value of 35 (assuming the current shell has a nice value=20):

```
nice -n 15 find / -name core > dumps 2>&1 &
```

- If an increment value is not specified as an option to **nice**, then an increment of 10 is assumed and the process will get a nice value of 30
- **nice** works only on new processes you are starting. Use **renice** to affect the nice value of an already running process.

9.6.2 The **at** command

At times, you may want to have a program run at a later time, when the system is not being used so heavily. SCO UNIX provides a command that will execute another command at any specified time. You must have authorization to run this command, which can be allowed or denied on a per-user basis.

- The **at** command reads its input from *stdin*. If *stdin* comes from your keyboard (i.e., you have not redirected it), complete the command by entering  **d** to signify the end of your input.

`at [ options ] time [ date ]`

- To find out who is still on the system at 10:15 pm tonight use the command:

```
$ at 1015pm
> date
> who
>  d
```

- The **at** command will mail the standard output of the executed command to your system mailbox, or you can specify output redirection with the delayed command to save *stdout* to a file of your choice:

```
$ at 1015pm
> date > at_data
> who >> at_data
>  d
```

9.7 Lesson Summary

Programs run on an SCO UNIX system as processes, which can spawn new child processes. System process activity affects overall performance. The operating system provides commands to monitor processes. They may be run in the background, enabling users to perform tasks simultaneously and increase their productivity. However, the more processes running, the greater the load on the system.

Commands are available that enable users to spread their workload over a greater period of time to improve overall system performance. Processes that become too large or that malfunction may have to be killed with an appropriate signal.

Review questions

1. What is the difference between a program and a process?
2. How are processes created?
3. What four pieces of information does the **ps** command give you about your processes when it is run with no command-line options?
4. How do you run a process in the background? Give a second method available to users of the Korn shell.
5. Give three ways you can terminate a process currently reading from *stdin*.
6. As a last resort, how could you kill your own shell process?
7. How can the **at** command be used to distribute the process load on a UNIX system?

9.8 Lab: Working with Processes

The goal of this lab is for you to become familiar with UNIX processes so that you may use the system more efficiently.

Overview

You will use the **ps** command to monitor processes as you create them using SCO UNIX commands.

Procedure

Begin this lab in your home directory.

1. Run the C shell program **csh**. Then run the Bourne shell **sh**. How do you know that you are running a different shell?
2. Run the **ps** command with the **-l** option. Look at the column headed by the symbol **SZ**. This is the size of the process in 1K blocks. Which of the three shells that you are running is the largest?
3. Exit from each of your shells by typing **exit**. What happened when you got to your original shell? What would have happened if you had terminated your original shell?
4. Run the **ps** command with the **-f** option. What is the parent of your shell process? Grandparent? What is the name of the program with a PID of 0? What is its parent?
5. Run the **find** command in the background so that it will find all files on the system. Redirect the *stdout* and *stderr* to */dev/null*.
6. Now try to log out by entering the **exit** command. What does the shell say?
7. Enter **exit** again, and then log back in. Use **ps** to check on the **find** process. What happened?
8. This time, enter the command:

9. nohup find / &

10. Now log out by typing the **exit** command twice, and then log back in. Run the **ps** command. Is the **find** process there?

11. From the data in the *nohup.out* file, how many files on the system was **find** able to locate?
12. Use **vi** to create a file named *atcmd* with these two lines:

13.

```
ps > results  
whodo >> results
```
14. Now run the command:

15.

```
at now + 1 minute < atcmd
```
16. In one minute, look for a file called *results* and view it.
17. Run the same **at** command, but remove the file redirections. Where does the output go now?
18. Use the **kill** command to kill your shell process. Does it work?
19. Try it with another signal. What happened?

9.9 Relevant Commands, Files, and Korn Shell Metacharacters

The following commands, files, and Korn shell metacharacters were presented in this lesson.

Relevant commands

at	Schedules jobs for execution at a particular time
bg	Runs jobs in the background
fg	Runs jobs in the foreground
find	Finds files in a given directory on the system
jobs	Displays the status of jobs
kill	Terminates a process by sending it a signal
ksh	Invokes the Korn shell
nice	Runs a command at a different scheduling priority
nohup	Runs a command immune to hangups and quits
ps	Reports process status
sleep	Suspends execution for an interval of time
who	Determines who is doing what on the system

Relevant files

<code>/dev/null</code>	Special device file that discards all output (the “bit bucket”)
------------------------	---

Relevant Korn shell metacharacters

&	Runs a command in the background
 Z	Suspends the current foreground process

9.10 Answers to Exercises, Labs, and Review Questions

This section provides suggested answers to the exercises, labs, and review questions.

Exercise: Viewing processes

You should be running the Korn shell which will be denoted by the command name **ksh** in the right-most column of the **who** output.

Exercise: Viewing process status

The parent process of the **ps** command is the shell you are running. The parent of your **login** process has a PID of 1, and is the **init** process. To count the number of processes running on the entire system, use the following command:

```
ps -e | wc -l
```

Subtract 1 to account for the header line **ps** generates.

Exercise: Background processes

When a command is running in the background, it will appear in the output of the **ps** command. When it no longer appears in the **ps** status, the process will have terminated. You will also get a one-line message the first time the standard output buffer is flushed after the process has terminated. That message will be:

```
[1] + Done find / > /dev/null 2>& &
```

Exercise: Job control

When the **sleep 60** process is finished, the **jobs** command should show that the **sleep 100** process is still running in the background.

Answers to review questions

1. A program is an executable file that resides as binary code on the hard disk. As with all files, it has an inode number and a filename. A process is a program that is currently executing. It includes all the program data and kernel information used to schedule and control the execution of that program.

2. The processes that you run in a login session are created by your shell. The shell process is the parent of the new processes created when you run commands from the shell. Each new process is referred to as a child process.
3. The `ps` command tells you the process ID number (PID) of each process, the tty device associated with each process (TTY), the cumulative CPU time used by each process (TIME), and the command that executed each process (CMD).
4. You can instruct a process to run in the background by appending the ampersand symbol (&) to the end of the command line invoking the process. If you are using the Korn shell you can press `CTRL` z while a process is running to get the shell prompt back, and then type the `bg` command to run the suspended process in the background.
5. A process reading stdin from the keyboard can be terminated by typing `CTRL` d, representing the end of file. You can signal the process by pressing the `DEL` key which sends the interrupt signal (2) or typing `CTRL` \ which sends the quit signal (3). You can also go to another terminal and terminate the process with the `kill` command.
6. As a last resort, you can terminate your shell process with the command: **`kill -9 pid`** where *pid* is the process ID of your shell.
7. You can use the **`at`** command to distribute the process load on a system by scheduling commands that do not require user input to run at a later time when the system is being used less, such as late at night.

Lab: Working with Processes

The hierarchy of processes is similar to the file system hierarchy in that it has one process, **`sched`**, from which all other processes are descendants, and each process can have only one parent, but multiple children. The **`sched`** process always has a PID of 0.

10

The Shell Environment

AV 10-1

The slide is titled "The Shell Environment" in a blue-bordered box at the top. Below the title, there is a bulleted list of three items. At the bottom of the slide, there is a dark blue footer bar containing the text "ntro1001" on the left and "© The Santa Cruz Operation, Inc." on the right.

The Shell Environment

- List the shells offered by SCO UNIX operating systems
- List the features of the Korn shell
- Customize your login environment

ntro1001 © The Santa Cruz Operation, Inc.

Lesson 10

The Shell Environment

Introduction

Throughout this course, you have been using a shell process to issue commands to the system. The shell has been your interface to the operating system, allowing you access to its many capabilities. In addition, your shell has provided a working environment for you.

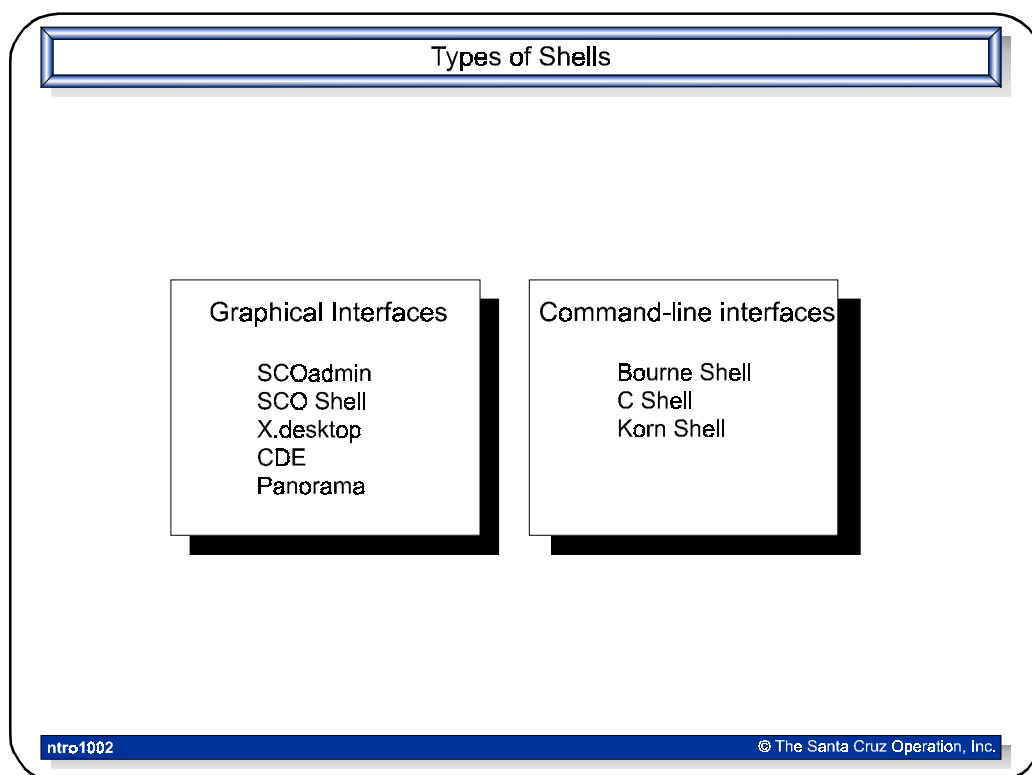
The shell defines many of the behaviors of the system such as where the system will search for commands when you invoke them. By learning to customize your shell environment, you will have an easier time working on a UNIX system.

Objectives

This lesson covers the shell features you have noted in previous lessons and introduces you to several of the Korn shell's more advanced features. By the end of this lesson, you will be able to:

- List the shells offered by SCO UNIX operating systems
- List the features of the Korn shell
- Customize your login environment

AV 10-2



10.1 Shell Review

In Lesson 2: “Overview of SCO UNIX Systems,” you learned that the primary function of a shell is to act as an interface between the user and the operating system. It takes commands issued by the user and translates them into language the kernel understands to execute them.

There are two primary classes of shells defined by the way they perform this function: command-line interfaces and graphical shells.

10.1.1 Command-line interfaces

The command-line interfaces were the first shells written for the UNIX operating system; therefore, these shells are often referred to as UNIX system shells. The SCO UNIX operating system provides several command-line interfaces:

- The Bourne shell, **sh**, and a restricted version of the Bourne shell, **rsh**
- The C shell, **cs****h** (refer to Appendix C for more information on the C shell)
- The Korn shell, **ksh**, and a restricted version of the Korn shell, **rksh**

These shells represent a progression of development, each version providing more features than its predecessor. The Korn shell is actually an extension of the Bourne shell, having been developed from the same program code.

The restricted versions of the above shells limit your access to certain directories, files, and commands.

10.1.2 Graphical interfaces

Graphical interfaces (sometimes referred to as graphical “shells”) offer varying degrees of visibility to the operating system, from simple menus to elaborate desktops. They may be character-based, which allows their use on regular character terminals, or they may be graphical, requiring special hardware and software support.

SCO OpenServer Release 5 provides these programs designed to be used as graphical interfaces:

SCOadmin (scoadmin)	A shell designed for system administrators that provides access to system maintenance functions through menus. There are both character-based and graphical versions of this program, so that it may be run in either environment.
SCO Shell (scosh)	A character-based, desktop shell designed for novice users who do not need the full power of UNIX at their fingertips. It provides several views of the system, including a list of commonly run utilities, such as electronic mail, a calendar, a calculator, and a reminder program.
SCO X.desktop (xdt3)	A full-scaled graphical desktop in a windowing environment supporting a mouse. It provides for direct manipulation of graphic images and <i>drag and drop</i> user interaction.

UnixWare 7 provides these programs designed to be used as graphical shells:

SCOadmin (scoadmin)	An enhanced version of the SCO OpenServer administration shell
CDE	A full-scaled graphical desktop and window manager
Panorama	An alternate graphical window manager

AV 10-3

Shell Features

- Command editing and execution
- Use of filename expansion characters
- Built-in command library
- Redirection of input and output
- Background processing
- Variables to control the shell environment
- Programming interface

ntro1003
© The Santa Cruz Operation, Inc.

AV 10-4

Shell Metacharacters

<code>ENTER</code>	initiates execution	<code>></code>	redirects <i>stdout</i>
<code>;</code>	separates commands	<code>>></code>	appends <i>stdout</i>
<code>&</code>	executes in background	<code><</code>	redirects <i>stdin</i>
<code> </code>	pipe	<code>"</code>	quotes a string
<code>?</code>	represents any one character	<code>[]</code>	specifies one from a set of characters
<code>*</code>	represents any string of characters	<code>\$</code>	references a variable

ntro1004
© The Santa Cruz Operation, Inc.

10.2 Features of UNIX System Shells

In this course, you have used the Korn shell to access the SCO UNIX operating system. All UNIX system shells share the features discussed thus far, with the exception of job control, as noted in the last lesson. Common features of shells include:

- They provide special command-line features for combining commands on one line (`;`), extending commands to multiple lines (`\`), escaping special characters (`'` `"` `\`), and erasing parts or all of the command line (`CTRL h` and `CTRL u`)
- They provide filename expansion characters for matching multiple filenames with a single string and as a shortcut for typing long pathnames (`*` `?` `[ ]`)
- Shells have built-in commands that they can execute directly without creating a new process. Most of these commands have binary counterparts in case a particular shell does not support them. (The commands **cd**, **echo**, **exit**, **pwd**, **kill**, and others are actually built into the Korn Shell.)
- Shells allow for flexible data flow by providing the capabilities of performing input/output file redirection (`>` `>>` `2>` `<`) and piping commands (`|` `^`)
- Shells provide the ability to execute commands in the background (`&`)

The UNIX system shells also provide features that will be introduced later in this lesson:

- A configurable environment that allows you to create and modify commands, prompts, and other behaviors of the system
- A complete, high-level command language that enables you to create simple or complex programs. These shell programs are called *shell scripts* and can be executed as commands by the shell.

In addition, the Korn shell provides some enhanced features that will make your use of the system much easier.

10.2.1 Advanced Korn shell features

There are many advanced features that the Korn shell provides; you have already learned about job control in the last lesson. Additional features that offer great convenience for the user are:

- Command-line history
- Command-line editing
- Alias expansion

10.2.1.1 Command-line history

The Korn shell records all the commands you enter in a session in a hidden file named `.sh_history` in your home directory. The Korn shell uses this file to allow you to easily re-execute commands you typed previously.

- The shell's built-in **history** command allows you to view this file:

```
history [ -n ]
```

- The numeric option *n* specifies the number of past commands to display; the default value is 16.
- To re-execute the last command you entered, simply enter **r** <Enter>
- To re-execute a particular previous command, for example, **ls -l**, enter:

```
r ls
```

- The command does not need to be spelled out in full; it needs to be specified only by enough characters to be able to uniquely identify it.
- You can also use the command line number to re-execute a command. You can get a command's number from the **history** command.

10.2.1.2 Command-line editing

The history function also provides a way for you to edit mistyped command lines.

- Press the  key to put you into a command editing mode where the commands of a system editor (commonly **vi**) can be used to re-edit a previous command

- Use the same commands you use in **vi** to move through your command list as if it were a file, but don't use control, keypad, or arrow keys. Start with the **k** command for moving up the command list; use the **j** command for moving down.
- Pressing  will complete the edit and send the command that is displayed for execution

In addition to correcting mistakes, this editing feature is valuable for altering the parameters to previous commands and re-executing them.

Exercise: Command history and editing

Use the **file** command to determine the file type of `.sh_history`.

Now use the command line editing features of the shell to change the previous command from **file** to **wc**.

10.2.1.3 Alias expansion

The use of the term *alias* here is similar to its use with electronic mail. Here, an alias name represents a commonly typed command, complete with options and arguments if desired.

- Create a new name for a longer command line using the **alias** command:

```
alias newname='command [ arguments ]'
```

- The *command* needs to be enclosed in quotes if you are using any options or arguments that require the use of shell metacharacters, including spaces
- There must be no spaces around the (=) sign
- Once an alias has been assigned, it will last for the duration of the login session
- Aliases cannot be used to reference command-line arguments alone; they must begin with a command name and be used at the start of a command line
- This example of an alias allows you to combine three actions into one:

```
alias gh='cd;echo Home Sweet Home;lf'
```

- You can use the name of an existing command as an alias name to redefine its behavior. But you lose the original behavior of the command you aliased. After defining this alias, the **ps** command will give a full status

```
alias ps='ps -f'
```

- Typing the **alias** command by itself displays your current aliases

Exercise: Alias expansion

View your current aliases by entering the **alias** command by itself. Do you see any familiar commands?

Create the alias **bye** for the commands **clear** and **exit**. Now log out by typing:

bye

AV 10-5

Shell Environment Variables	
HOME	Login directory
MAIL	System mailbox
PATH	Command search path
PS1	Primary prompt string
SHELL	Default shell
TERM	Terminal type

ntro10053

© The Santa Cruz Operation, Inc.

10.3 The Shell Environment

The shell environment is highly configurable to meet your needs and preferences. This configuration is accomplished through special environment variables that hold data and through special files called *scripts* that contain commands that customize your environment when you log in.

This section describes the use of the Bourne (**sh**) or Korn (**ksh**) shells; the C shell (**cs**) uses different syntax.

10.3.1 Environment variables

Your shell process contains space for holding data that you can name and access from the command line. Other programs can also access this data because it can be provided in their environments when they are spawned by your shell.

- The **env** command allows you to see the environment variables that have been defined
- Some of the variables include:

HOME	The full pathname of your home directory
MAIL	The path to your system mailbox
PATH	The directories where the shell looks to find a command
PS1	Your shell's primary prompt string
SHELL	The current shell you are using
TERM	Your terminal type

- Note that by convention, environment variables are named in all upper case characters
- You can view the contents of any environment variable using the **echo** command. Place a dollar sign (\$) character before the variable name:

echo \$PATH

- You can create a new environment variable, or change the value of an existing one. Quotes are needed only if the contents contain a shell metacharacter such as a space or newline.

NAME='new contents'

- The **unset** command deletes an environment variable:

unset NAME

- To have the variable show up in the environment of a child process so the child can access it, you must use the **export** command:

export NAME

Exercise: Environment variables

Display the contents of your `TERM` environment variable. Record this value.

Change your `TERM` variable to the string “guess work”. Try to edit the *letter* file with `vi`. What happens? Set the `TERM` variable to its original value.

Change your `PS1` variable with this command:

`PS1=' $PWD[! ] '`

What does `$PWD` mean? What does the exclamation mark (!) character indicate in `PS1`?

AV 10-6

Sample .profile

```
#
# Copyright (C) 19881991 The Santa Cruz Operation, Inc.
# All Rights Reserved.
#
# User $HOME/.profile commands executed at login time
#
# search path:w
PATH=/bin:/usr/bin:/u/bin:/usr/local/bin:/local/bin:${HOME}/bin
:
#
# login name
LOGNAME=charles
#
# mailbox location
MAIL=${HOME}/.mailbox
#
# default permissions for new files
umask 022
#
# terminal setting
eval `tset m ansi:ansi m :\\e?ansi r s Q`
#
export PATH LOGNAME MAIL
```

intro1006

© The Santa Cruz Operation, Inc.

10.3.2 Environment files

When you log in, certain environment files, called scripts, are read and the commands within them are executed. Some of these commands set environment variables, some establish aliases, and others modify the behavior of the system according to your preferences. It is with these files that you can customize your own working environment.

10.3.2.1 The *.profile* file

When you log in, your shell executes the commands contained in the hidden file *.profile* in your home directory.

- *.profile* is a regular text file that can be edited with **vi**. It was created when your account was first made.
- As with all script files, the shell ignores any lines that begin with a pound (#) symbol. Such lines can be used as comment lines to describe what is being done.
- Your *.profile* is where many of your environment variables are set and where your personal *umask* variable is set
- The **tset** command that sets the terminal type is invoked here
- Before the *.profile* in your home directory is executed, the system-wide */etc/profile* script displays the message of the day (*/etc/motd*) and checks your system mailbox

10.3.2.2 The *.kshrc* file

The Korn shell has an additional command script, *.kshrc*, that is read and executed when the Korn shell is invoked. This hidden script file should contain commands explicit to the Korn shell.

- *.kshrc* is executed after the *.profile* script
- *.kshrc* can be used to set aliases
- The system editor to be used for command-line editing is specified here:

set -o vi

- Unlike *.profile* which executes only for login, *.kshrc* executes for any **ksh** you create. It is a good idea to keep it relatively short.

10.3.3 Configuration commands

Any SCO UNIX command can go into shell scripts. Your *.profile* and *.kshrc* files, however, are designed for initial setup functions only. Commands that affect the behavior of your terminal and shell are the most suitable for those two files. Some of these include:

stty	Sets terminal settings, such as the erase character, delete key, baud rate, etc.
setkey	Programs your function keys to emit strings that could invoke commands
set	A shell built-in that sets predefined variables such as <code>ignoreeof</code> and <code>noclobber</code>
trap	A shell built-in that traps signals and ignores them or performs another action

10.4 Lesson Summary

SCO UNIX system shells provide the user with multiple features for accessing the power of the UNIX operating system. In addition, they offer a great deal of flexibility in customizing your home environment. UNIX system shells provide a number of predefined environment variables that commands and applications can access to modify their behavior to fit your environment. They also provide scripting capabilities that allow the execution of many UNIX commands from regular text files. The Korn shell also offers enhanced features for easing the entry of commands on the command line.

Review questions

1. Give examples of two types of shells available for SCO UNIX systems.
2. Name three ways to re-execute a command that you previously entered using the Korn shell.
3. What must you do to an environment variable to make sure it will be available for other processes to access?
4. When should you put commands in your *.kshrc* file rather than your *.profile*?
5. How could you configure your environment so that after you logged in, you invoke the UNIX **mail** program?

10.5 Lab: Configuring Your Shell Environment

The goal of this lab is to learn to customize your login environment to suit your needs and preferences.

Overview

By modifying environment variables and editing your *.profile* and *.kshrc* files, you will customize your shell environment in a number of different ways.

Procedure

Start this lab from your home directory.

1. Display the contents of your **PWD** environment variable. Change to your *bin* subdirectory and display the same variable. How did it change?
2. Change back to your home directory.
3. Display your **PATH** environment variable. Note the colon (:) separating the different directory names. What is the last directory in which the system searches for commands?
4. Make a new environment variable called **GREETING** and assign it an appropriate salutation. Don't forget to use quotes.
5. Use the **env** command to see if it is in your environment. Is it there? What must you do to put it in the environment?
6. Export the variable **GREETING** and use **env** to verify it's there.
7. Use the **unset** command to unset the variable **PS1**. What happened?
8. Reset the **PS1** variable to the string '!**\$**'. Is your prompt back? What does the number portion mean?
9. Invoke another Korn shell by typing **ksh**. Did the prompt change?
10. Exit the shell you just invoked using **Ctrl**+**d**, and then export the **PS1** variable.
11. Use the **history** command to re-invoke another Korn shell. Is the prompt the same now?
12. Enter the command:

13.set -o ignoreeof

- 14. Can you log out using `CTRL d` as you did before?
- 15. Use the Korn Shell's command editing feature to re-execute the above command with a `+O` instead of a `-O`.
- 16. Now log out of the second Korn shell using `CTRL d`.
- 17. Enter the command:

18.setkey 10 'clear\n'

- 19. What happens when you press the `F0` function key?
- 20. Use `vi` to add this alias to the bottom of your `.kshrc` file:

21.alias bye='clear;exit'

- 22. What can you do to make this alias take effect?
- 23. Use `vi` to add these lines to the bottom of your `.profile`:

**24.trap 'clear;banner good bye' 0
clear
banner good morning**

- 25. What happens when you log out and then log back in again. Log out one more time. What happens?
- 26. Remove the last three lines from your `.profile` and replace them with:
mail
- 27. Quit the mail program (x `ENTER`) and log out using your **bye** alias and then log back in again. What has happened now?

10.6 Relevant Commands and Files

These commands and files were presented in this lesson.

Relevant commands

alias	Korn shell built-in command for creating and displaying aliases
cs	The C shell
env	Sets environment for command execution
export	Bourne and Korn shell built-in command that makes environment variables accessible to child processes
ksh	The Korn shell
rksh	The restricted version of the Korn shell
rsh	The restricted version of the Bourne shell
scoadmin	Invokes SCOadmin applications
set	Shell built-in that sets and unsets internal variables
setkey	Assigns strings to the function keys
sh	The Bourne shell
stty	Sets terminal modes
trap	Shell built-in command that handles signals
tset	Sets terminal characteristics
unset	Shell built-in command for deleting a variable
xdt3	SCO OpenServer graphical X.desktop executable

Relevant files

<i>/etc/motd</i>	The message of the day file
------------------	-----------------------------

<i>/etc/profile</i>	System-wide login environment file for Bourne and Korn shells
<i>.kshrc</i>	Korn shell specific environment file
<i>.profile</i>	Bourne and Korn shell environment file
<i>.sh_history</i>	Korn shell command history file

10.7 Answers to Exercises, Labs, and Review Questions

This section provides suggested answers to the exercises, labs, and review questions.

Exercise: Alias expansion

The Korn shell commands, **history** and **r**, are actually aliases to the **fc** command.

Exercise: Environment variables

If you attempt to use the **vi** editor with a terminal type that it doesn't know, it will issue the following message and put you in open (line editing) mode:

```
guess work: Unknown terminal type
I don't know what kind of terminal you are on -
all I have is 'guess work'.
[Using open mode]
```

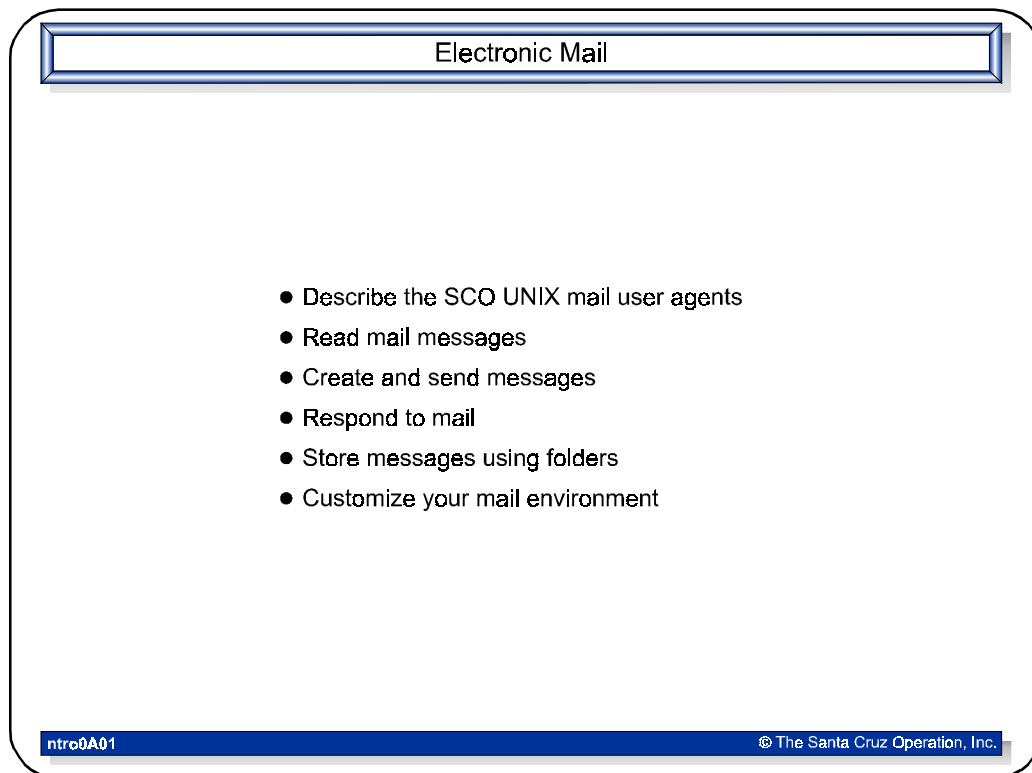
Answers to review questions

1. Two types of shells available in SCO UNIX systems are command-line and graphical interfaces.
2. You can re-execute a command using the editing feature by pressing the  key and then repeatedly pressing the  key until the desired command is displayed on the command line, then pressing . The other two ways are by using the **r** command. You can either type **r** followed by enough of the desired command line to guarantee a unique match to that command, or by typing the number of the desired command within the history list.
3. You must export an environment variable in order for it to be accessible to child processes.
4. Put commands specific to the Korn shell in your *.kshrc* file, such as aliases or commands that configure the editor used for command-line editing.
5. To log into the UNIX mail program, edit *.profile* in your home directory and place the following command line at the end of the file: **mail**

A

Electronic Mail

AV A-1



Appendix A

Electronic Mail

Introduction

Communication via electronic mail (email) is an important way to share information and ideas. Like the text editing utilities, SCO UNIX systems provide several electronic mail facilities.

In this lesson you will use the standard UNIX email program, **mail**, to acquire skills in reading, creating, organizing, sending, and responding to electronic mail as well as learning how to customize your own mail environment.

Objectives

Using the UNIX **mail** program, this appendix covers the skills you need to communicate effectively using electronic mail. By the end of this lesson, you will be able to:

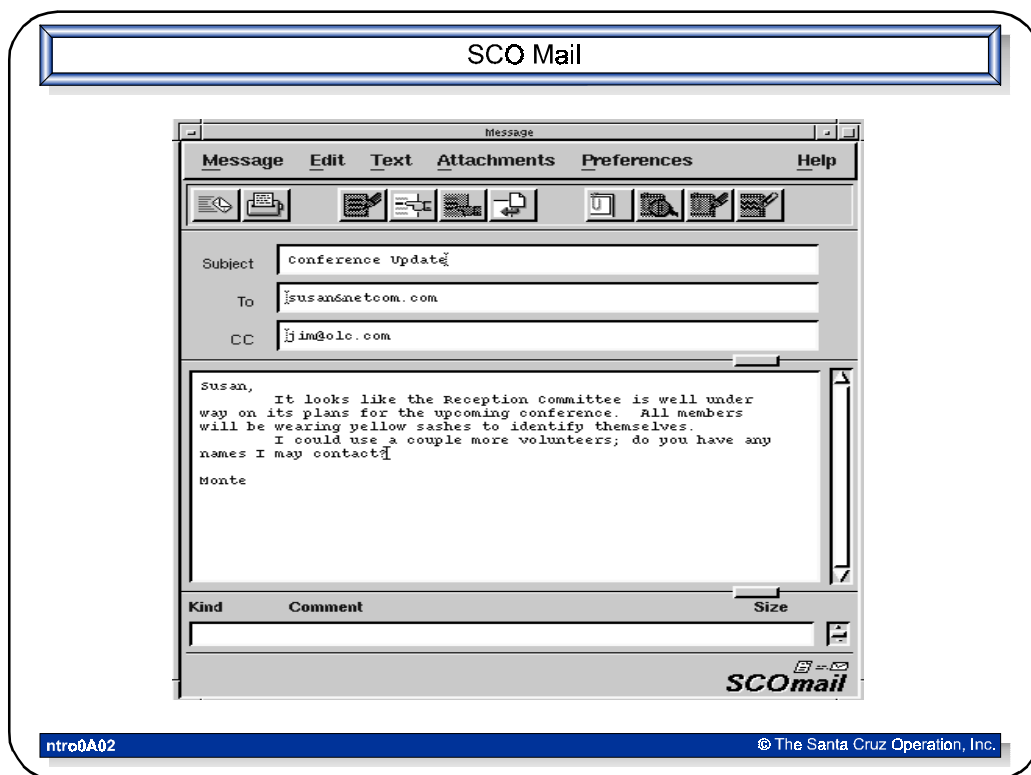
- List the user mail agents available on SCO UNIX operating systems, and describe under what conditions they would be used
- Read your mail messages
- Create your own messages and send them to individuals or groups of users
- Respond to mail sent to you
- Explain how mail messages are stored before and after they have been read
- List the types of customization that can be made in your mail environment

A.1 Electronic Mail Overview

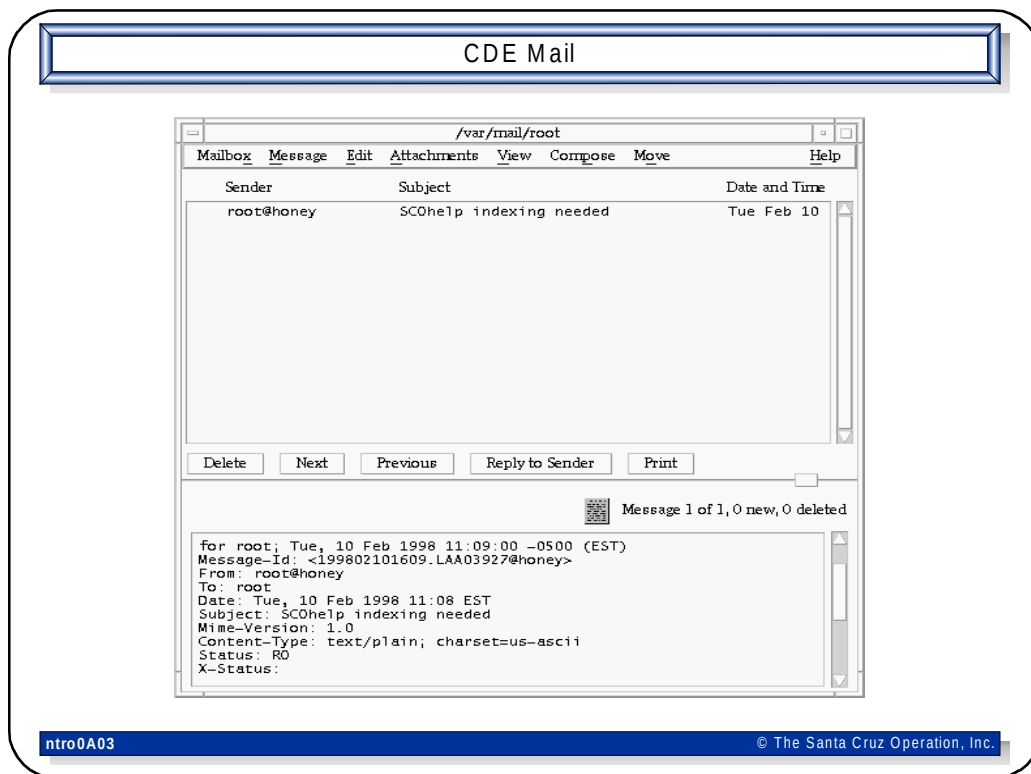
Electronic mail has several features that make it a valuable means of communication within a particular work group, an entire company, or a network of individuals with common interests. Email is:

Fast	Messages are sent in seconds on a single system, and within minutes, messages can reach destinations around the world.
Non-invasive	Messages queue up in a user's personal mailbox without interrupting what they are currently doing, as a phone call would.
Private	Only users to whom the message is sent will receive the message, unlike a facsimile that is often handled and viewed by intervening parties.
Self-documenting	Mailers provide ways to record and track the flow of messages as they are delivered, forwarded, and replied to by other users.

AV A-2



AV A-3



A.2 The UNIX Mail System

Electronic mail on SCO UNIX operating systems is handled by two utilities: a user agent and a transport agent. The user agent is the interactive program that allows you to send and receive mail; the transport agent is a system program that handles the delivery of mail messages.

A.2.1 Mail user agents

Each SCO UNIX operating system has one or more user agents, ranging from the simple text-based **mail** program, to sophisticated graphical interfaces:

UNIX mail	The standard UNIX mailer (mail) included in all versions of SCO UNIX. It provides basic email functionality, as well as the ability to send mail non-interactively.
SCO Shell Mail	A menu-driven mailer in SCO OpenServer that is easy to use, provides many high-level features, and runs on standard character terminals
SCO Mail	A graphical mailer in SCO OpenServer (scomail) that offers the same features as SCO email , but used within a graphical environment
CDE Mail	A graphical mailer in UnixWare 7 (dtmail) accessible from the CDE desktop
Netscape Mail	A graphical mailer in UnixWare 7
Pine	A screen-oriented message-handling tool (pine) with limited features for novice users

A.2.2 Mail transport agents

SCO OpenServer supports two transport agents: MMDF (Multichannel Memorandum Distribution Facility) and **sendmail**. UnixWare 7 includes only the **sendmail** transport agent.

Mail transport agents are configured by the system administrator to use various communication channels (UUCP or TCP/IP) to direct mail to its ultimate destination.

A.2.3 Addresses and mailboxes

Just as the post office uses an address system to deliver mail to individual mailboxes, electronic mail is delivered to individual mailbox files using a system of addresses.

When an account is set up for you, you are given a personal mailbox on the system. It is a file with the same name as your login name and is commonly located in the directory */usr/spool/mail* (the location is configurable by the system administrator).

Once mail messages are read, they may be saved in files in a subdirectory of your home directory, such as *mailfolders*. If you save messages without specifying a folder to put them in, most mailers will use the filename *mbox* as a default.

Addresses may follow several constructions, but typically utilize the Internet syntax which is composed of three parts:

Username[@(*Hostname*.)]*Domain*]

Username The user name is required in all addresses. It is the only identification required when the mail message is going to another user using the same host.

Hostname The name of the machine to which the mail is being addressed. The hostname is required when sending mail to users on other computers unless the mail administrator sets up an aliases file to map users to mail hosts.

Domain A name that an organization registers with the Network Information Center (NIC) that describes the site where their machine is located. For example, in *sco.com*, *sco* is the site, and *com* is the name of the domain (an abbreviation for “commercial institutions”). Domain names are required only if you are sending mail outside of your site, for instance, onto the Internet.

Exercise: Electronic mailboxes

Find your system mailbox. Use **l** to list it. What type of file is it? Use **more** to view the file. Note the addresses used in the mail headers.

AV A-4



A.3 Using the SCO UNIX mail Utility

Each time you log in, the shell informs you if there is mail in your mailbox by printing the message `you have mail` on your screen. You can read your mail at any time.

SCO's email utility, **mail**, is a user agent that offers convenient features for reading, responding to, sending and organizing electronic mail. You can customize your mail environment to make interacting with mail even easier.

A.3.1 Creating and sending mail

Electronic mail may be sent to one or more individuals, to yourself as a reminder, or to a group of users identified by a common name, called an *alias*. Mail messages are usually created with a text editor and may incorporate parts of other messages and text files.

To create a simple mail message for another user on your local system, enter:

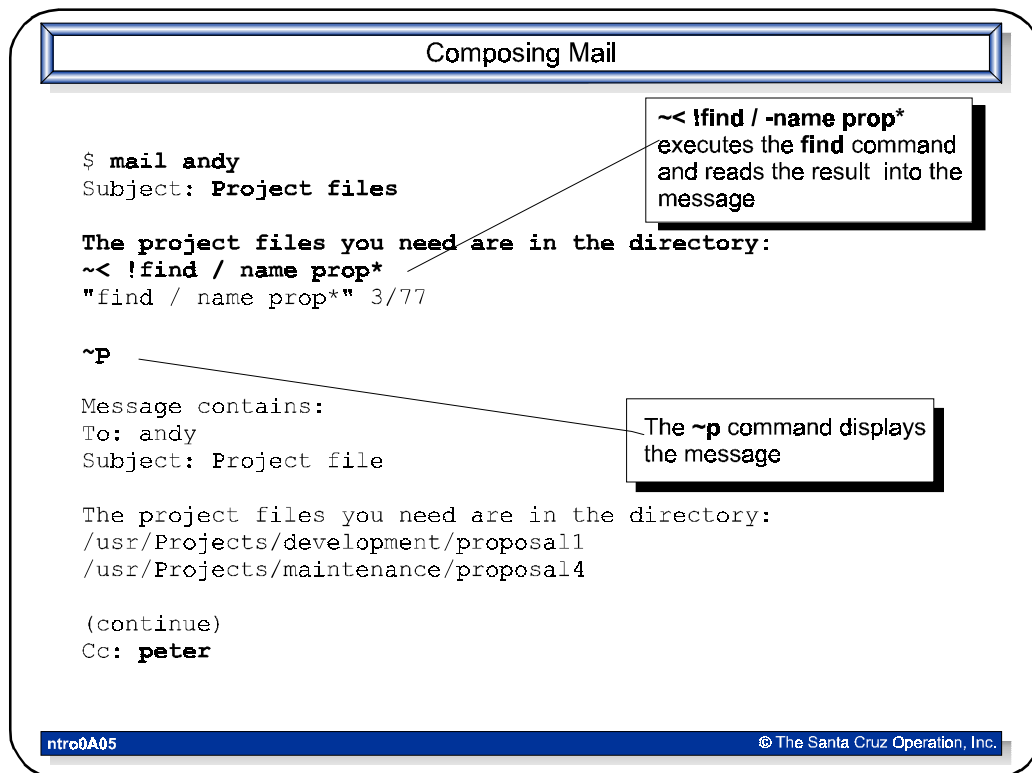
`mail login`

- Type a subject when prompted
- When **mail** enters *compose* mode, type the body of the message
- Use the  key to correct mistakes while typing. You cannot backspace to previous lines in compose mode.
- End the message with  **d**.
- Copy the message to other users when prompted (or press  to skip)
- Cancel a mail message without sending it by pressing the  key twice

Exercise: Creating a mail message

Send a grocery list to yourself as a reminder.

AV A-5



A.3.1.1 Composing mail messages

It may take more than a few simple keystrokes to compose the message you want to send. The **mail** utility supports commands, called tilde escapes, to access advanced features. These commands are used while composing message text, by typing **~command** at the beginning of a new line.

These compose-mode commands are often used to create messages:

~v	Invokes the vi editor for composing message text
~r filename	Reads the contents of the named text file into the message
~h	Edits the message header information—add or modify these fields: To , Subject , Cc , Bcc and Return-receipt-to .
~p	Prints the entire message to the screen
~!command	Executes the shell command and returns
~< !command	Reads the output from the shell command into the mail message

Exercise: Composing mail

Send a message to your system-mate and include the file */etc/motd*. Edit the message to add a personal greeting.

Request a receipt so you will know when it has been read.

A.3.1.2 Sending data files using mail

You may want to include a data file, such as a word processing or spreadsheet document, in your email message. Most graphical mail interfaces support the ability to send data files. However, the standard UNIX **mail** utility allows only ASCII text characters to be included in files and strips out any non-ASCII characters.

To include data within a message sent using **mail**, the file must first be encoded as an ASCII file. Both the sender and recipient must use the same encoding utility to be able to exchange files.

To encode a data file, use the UNIX utility **uuencode** and save the ASCII output:

```
uuencode source_filename remote_filename > encoded_file
```

The source filename refers to the existing data file, the remote filename is the name the file will be given when it is decoded by the recipient. Once encoded, the file can be read into a mail message using the **~r** command

Data files are often sent using **mail** from the command line. For example, these commands encode and mail a spreadsheet to **peter**:

```
uuencode budget.xls budget.xls > budget_encode  
mail -s "Quarterly Budget" peter < budget_encode
```

Exercise: Sending data files

Encode and send the file *what_am_i* to your system-mate. Use “data file” as the subject.

AV A-6

Reading Mail

```
$ mail
SCO OpenServer Mail Release 5.0  Type ? for help.
"/usr/spool/mail/andy": 4 messages 4 new
 N 4 chris      Thu Jan 15 07:31   31/987   Conference
 N 3 susan      Thu Jan 15 07:31   25/870   Conference
 N 2 rob        Thu Jan 15 07:30   12/308   Pure Poetry
>N 1 susan      Thu Jan 15 07:29   14/463   Project X
```

&1

Message 1:

```
From susan Thu Jan 15 07:29:52 1998
From: susan <susan@acme.com>
XMailer: SCO OpenServer Mail Release 5.0
To: andy@acme.com
Subject: Project X
Date: Thu, 15 Jan 98 7:29:52 CST
MessageID: <9801150729.aa15141@mailsrvr.acme.com>
Status: R
```

For all those who are interested, there will be a presentation this afternoon outlining the important aspects of project X. Please r.s.v.p by mailing the coordinator (Shirley Jones) by noon. See you there!

Susan

&

ntro0A06

© The Santa Cruz Operation, Inc.

A.4 Reading Mail

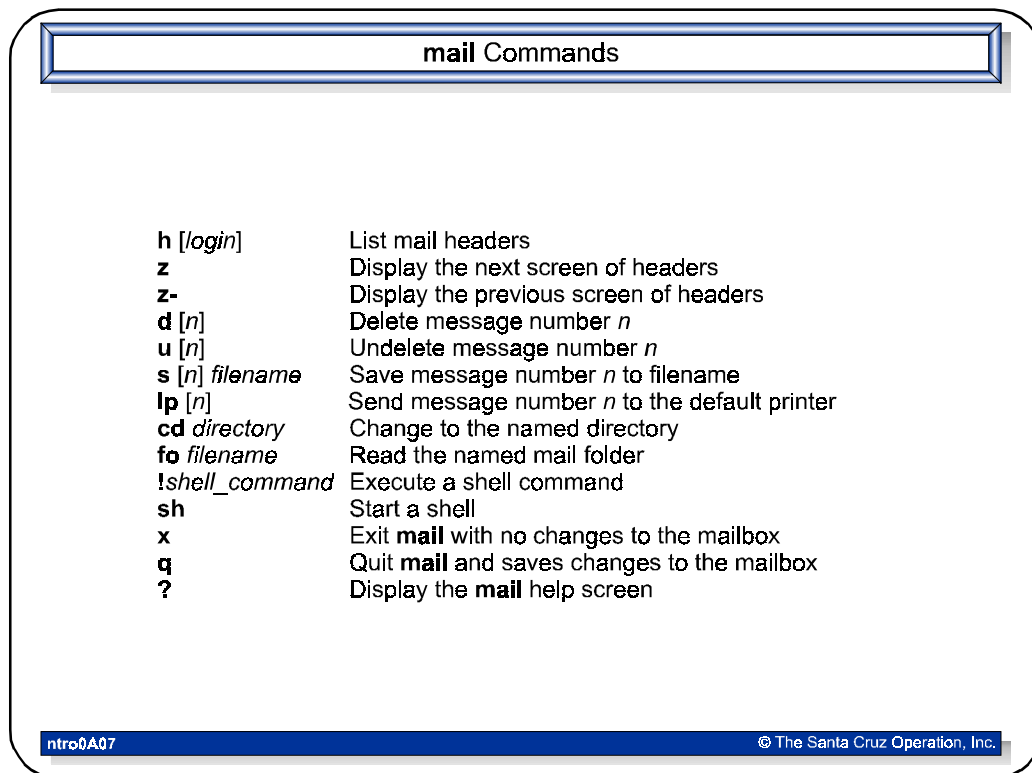
Reading mail is a common task you will do during a mail session. Your mail is summarized in a list of one-line headers that are listed in the order the messages were received. These headers are used to select the mail message(s) with which you wish to work.

- If mail is present, a list of message headers appears on the screen that provide information on each piece of mail.
- Your mailbox lists messages by number, with the most recent at the top. The **>** symbol marks the current message
- The **mail** command prompt is displayed as an ampersand (&) at the end of the message list
- To read a message press  (for the message marked **>**) or type a message number followed by . The contents of the selected message will be displayed on the screen along with the header information.
- After a message is read, it is saved to the *mbox* file in your home directory
- To quit **mail**, type **q** at the prompt (&)

Exercise: Reading mail

Read the grocery list you sent to yourself.

AV A-7



A.4.1 Managing Mail

The **mail** utility provides an extensive command set for manipulating mail messages. Use these commands to:

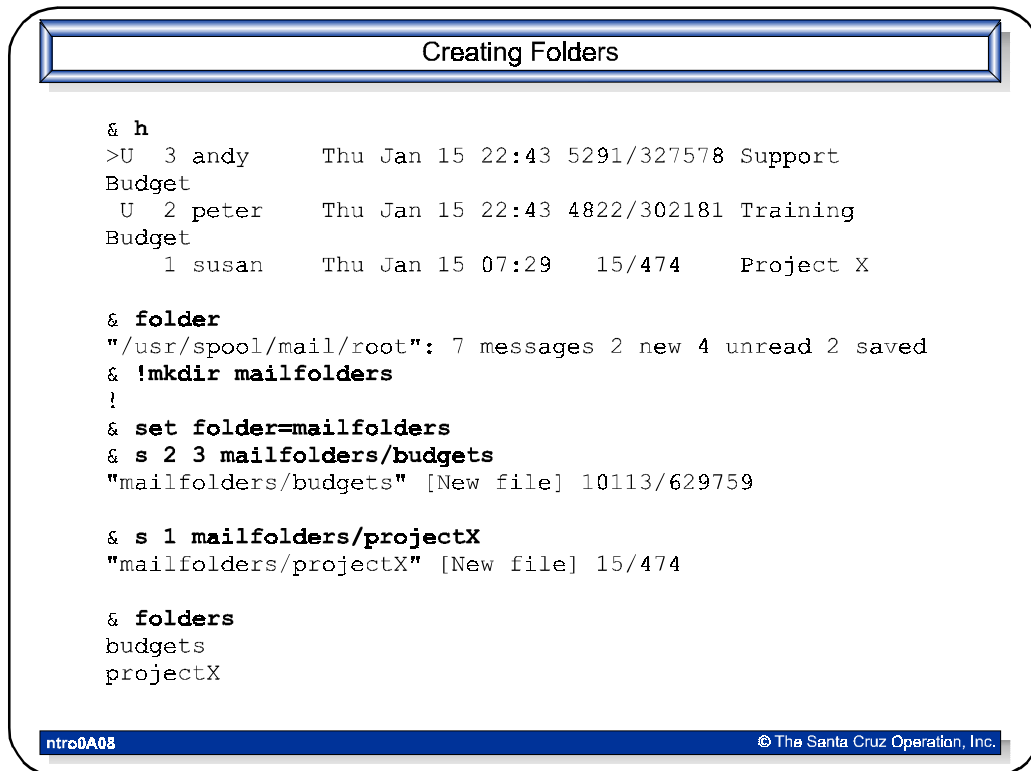
- Get help on commands
- Display message lists
- Save messages in *folders* (files)
- Delete and undelete messages
- Print messages
- Access the shell without leaving **mail**
- Exit mail

Exercise: Using mail commands

List all messages received from your system-mate.

Print and then delete your grocery list.

AV A-8



A.4.1.1 Storing mail

You can establish your own area for storing your mail messages in a way that keeps similar mail messages together. These mail files, or *folders*, are essentially identical to your system mailbox, but they are under your organizational control.

- Mail folders are created automatically when you save messages in a specified directory. Set the directory location with:

```
set folder=directory_name
```

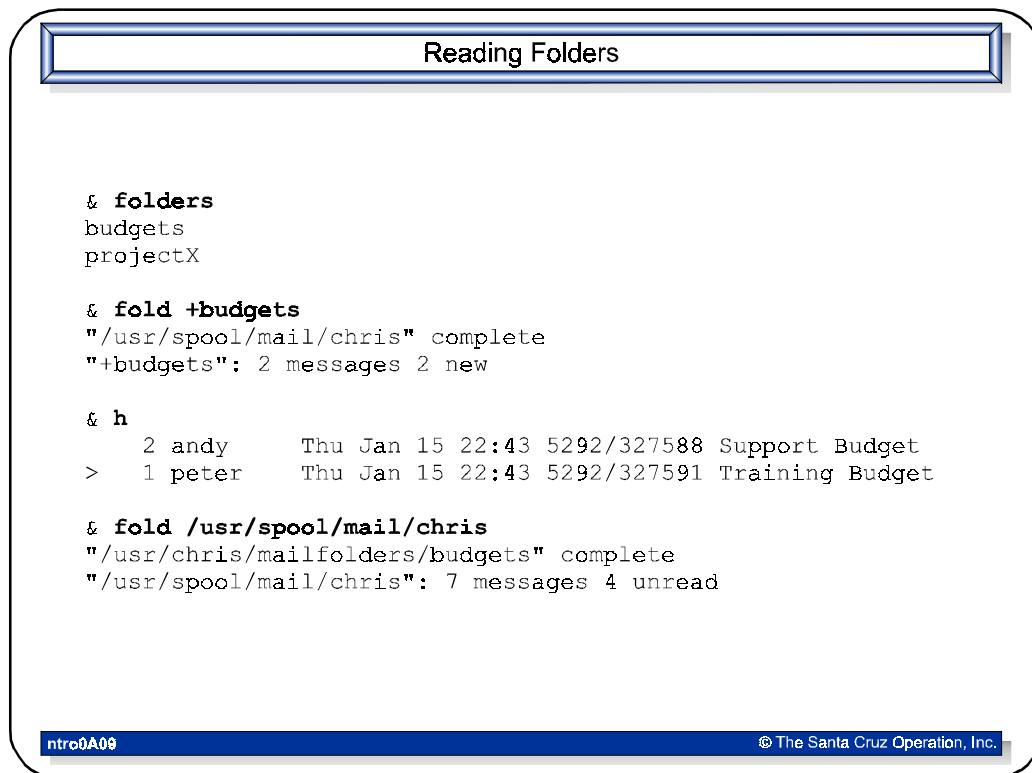
The directory which will contain the folders, *directory_name*, must already exist in your home directory.

- A mail folder is an alternate mailbox; you can do anything with it that you can do with your system mailbox
- Save messages to folders in your storage directory with the **s** command:

```
s 1 mailfolders/memos
```

- Write several messages to a single folder by saving them to the same folder name
- To see a list of all **mail** folders, use the **folders** command

AV A-9



- Use the **fold** command to read a folder in your storage directory:

fold +memos

- To switch back to the default mailbox:

fold /usr/spool/mail/jack

- To read an alternate folder when invoking **mail** from the shell prompt, enter:

mail -f ./mailfolders/memos

Exercise: Managing mail

Create a subdirectory of your home directory for storing mail. Save your messages to files named by the sender of each piece of mail.

View the folders and then read the saved files.

AV A-10

Encoded Mail

```

^A^A^A^A
From: andy Thu Jan 15 22:43:35 1998
From: andy@ulee.com
XMailer: SCO OpenServer Mail Release 5.0
To: chris@ulee.com
Subject: Support Budget
Date: Thu, 15 Jan 98 22:43:30 CST
MessageID: <9801152243.aa17556@oslep.ulee.com>
Status: R0

```

Chris, the Support Budget for 4Q1997 is enclosed below. Andy

```

begin 600 budget.xls
M!@ 3T" ' 30( < %= @ " &T" ( !A0( D "5
M @ "0 :T" * 00( H ') @ "P D" + !Y0( H
.
.
.
M %5# !X ' =FET86PY > ! %#3P > !@ &1A=&$U
M !0 B+'@ 0 !43U0 T,\1X*&Q&N$
/ @ # +

```

```

end
^A^A^A^A

```

ntro0A10

© The Santa Cruz Operation, Inc.

A.4.1.2 Saving data files

When data files are included with mail messages, the encoded portion is indicated with the keywords *begin* and *end*: The *begin* line includes the name and permissions the new file will be given when decoded.

To recreate the original data file, save the message to a file and use the **uudecode** utility:

```
uudecode filename
```

Exercise: Saving mail

Save the message including the file *what_am_i* to a file named *data*. Decode the file and list the directory to find the new file.

AV A-11

Responding to Mail

U	3	rob	Fri Jan 16 09:55	42/593	Pure Poetry
U	2	mark	Thu Jan 15 07:30	13/318	Conference
	1	susan	Wed Jan 14 12:29	15/474	Project X

& R 2

To: mark
Subject: Re: Conference
Cc: shirley

I will be able to help with the conference. Just let me know what you need.

John**& r 1**

To: susan
Subject: Re: Project X

The numbers for this project ...

& f 3 poets

Interpolating: 3
Message 3 forwarded to poets.

ntro0A11

A.4.2 Responding to mail

While reading mail, you can respond to a message by using the **reply** or **forward** commands at the mail prompt (&).

- The **r** command replies to the sender only. The **R** command sends the reply to every recipient of the original mail—all login names listed on the **To**, **Cc**, and **Bcc** lines of the header.
- When replying, **mail** fills in the **To** and **Subject** fields and begins a response by placing the cursor in input mode. The text of the original mail is not automatically included; it can be read into the message using the **~m** command.
- To forward a message to user **login** with no modifications, use the **f** command

Exercise: Responding to mail

Reply to your classmate with a thank-you for sending you a welcome message.

A.5 Using Aliases

An alias is a single name that refers to a group of one or more users. The system administrator may set up system-wide aliases that group users on different machines to the same alias name. You can set up your own personal aliases using the **a** command.

Since aliases allow a single mail message to be sent to many users, they should be used with care when sending or replying to mail.

To set up a mail alias enter:

& a *aliasname login1 login2 ...*

For example, to create an alias called **sales** for users **jane**, **john**, **jim**, **joe**, and **biff**, enter:

& a sales jane john jim joe biff

To list all aliases, type **a** with no arguments:

& a

Exercise: Creating aliases

Create an alias named *friends* that includes all the users who sent you mail.

AV A-12

mail Settings	
askcc	Prompts for the carbon copy list
autoprint	Automatically prints messages after d and u commands
chron	Displays messages in chronological order
hold	Keeps all messages in <i>mailbox</i> after reading
header	Prints the header summary when entering mail
record=filename	Records all outgoing messages in <i>filename</i>
save	Saves interrupted messages in the file <i>dead.letter</i> (default)
sign=string	Sets the variable inserted into a message when the ~a (autograph) command is used
outfolder	Records outgoing messages in files located in the directory specified by the folder variable

ntro0A12

© The Santa Cruz Operation, Inc.

A.6 Customizing Your Mail Environment

There are a set of variables that affect the behavior of the **mail** utility. These options customize **mail** to suit individual needs and preferences. Variables are set in three ways:

- Interactively using the **set** command while in a **mail** session
- Automatically from the mail environment by adding variables to the *.mailrc* file
- Automatically from the shell environment by adding variables to the shell environment file (e.g., *.kshrc*)

A.6.1 Setting options interactively

The preference settings in **mail** are controlled by options to the **set** command. When set interactively within **mail**, options are in effect for the current session only.

There are two types of options used in **mail**:

Switches *On/off* options which are not used with arguments:

`set option`

Variables Options set with a value:

`set option=value`

All options are turned off (or erased) with:

`unset optionname`

This example configures **mail** to save a copy of all outgoing messages to the file *mycopy*:

set record=mycopy

AV A-13



A.6.2 Setting options automatically

The *.mailrc* file contains **set** commands that customize the behavior of **mail** each time it is invoked.

- Set commands are added to the *.mailrc* file, one per line.
- The syntax for **set** is the same as when used interactively
- No lines should be blank; errors will result
- Options selected in *.mailrc* can be overridden by using the **set** command interactively during the **mail** session.

A.6.2.1 Mail environment variables

These variables are set within the shell (e.g., *.kshrc*) and imported when **mail** is invoked:

DEAD	Defines the pathname of the file which saves interrupted or undeliverable messages
EDITOR	Specifies the editor to be used with the ~e (edit) command
HOME	Defines the user's home directory
LISTER	Sets the command used to list the folder directory (ls is the default)
MAILRC	Sets the name of the mail startup file (default is <i>\$HOME/.mailrc</i>)
MBOX	Specifies the pathname of the file in which messages are saved (default is <i>\$HOME/mbx</i>)
PAGER	Sets the command used to display messages (default is more)
SHELL	Sets the command-line interface used with the ~! and sh commands (default is sh)
VISUAL	Specifies the editor to be used with the ~v command

For example, to use the UNIX **pg** command to display mail messages, add these lines to the shell environment file (e.g., *.kshrc*):

```
PAGER=/usr/bin/pg
export PAGER
```

A.6.3 Adding a signature

You can create your own “signature” that can be included in messages you send. A signature is your usually real name along with professional or personal information.

- To add a signature, set the **sign** variable:

```
&set sign="Chris Ryder, Account Manager\nAcme, Inc."
```

- To include the signature within a mail message, use the **~a** command while in compose mode:

Enclosed are the budget reports you requested.

```
~a
Chris Ryder, Account Manager
Acme, Incorporated
```

- The newline (**\n**) and tab (**\t**) formatting characters can be included in a signature string

A.7 Relevant Commands and Files

These commands and files were presented in this appendix.

Relevant commands

mail	Invokes the interactive message processing system
-------------	---

Relevant files

<i>/usr/spool/mail</i>	Directory location of all system mailboxes
<i>\$HOME/mbx</i>	Default name for each user's personal mailbox
<i>\$HOME/.mailrc</i>	Environment file for customizing mail
<i>\$HOME/.kshrc</i>	File for setting environment variables
<i>/usr/lib/mail/mailrc</i>	Global startup file (optional)

B

***Transferring Files
Between Systems***

AV B-1

The slide is titled "Transferring Files Between Systems" in a blue-bordered box at the top. Below the title is a bulleted list of seven tasks. At the bottom of the slide, there is a blue footer bar containing the text "ntro0B01" on the left and "© The Santa Cruz Operation, Inc." on the right.

- Format floppy disks
- Backup files to UNIX-format floppy disks
- Backup files to DOS-format floppy disks
- Determine the type of archive on a floppy disk
- Restore archived files to the hard disk
- List the names of systems on the network
- Transfer files between systems using network commands

ntro0B01 © The Santa Cruz Operation, Inc.

Appendix B

Transferring Files Between Systems

Introduction

The ability to exchange data with other users and systems is an essential requirement of an operating system. SCO UNIX systems provide several services and utilities that ease the task of transferring files between systems.

In this lesson, you will use the commands provided with SCO UNIX systems to copy files to and from floppy disks, and to transfer files to and from other systems on the network.

Objectives

This appendix outlines the commands used to backup user files and transfer files between systems on the network. By the end of this lesson, you will be able to:

- Format floppy disks for use with DOS and UNIX systems
- Backup files to UNIX-format floppy disks
- Backup files to DOS-format floppy disks
- Determine the type of archive on a floppy disk
- Restore archived files to the hard disk
- List the names of systems on the network
- Transfer files between systems using network commands

B.1 Overview

SCO UNIX systems are designed to integrate into computing environments that include a wide range of system hardware and software. To achieve a high level of interoperability, many facilities are included to insure compatibility.

In addition to extensive Internet services, SCO UNIX systems support several other networking components that provide seamless integration with other systems, including:

- Netware and NT servers
- Windows and DOS workstations
- UNIX systems from other vendors

Your particular system may be configured with one or more additional services. These components all include utilities for transferring and sharing files, and accessing remote systems.

Even with minimally-configured systems, a core set of utilities is included with SCO UNIX to perform the basic functions of copying files to floppies and to other systems:

- **tar** and **cpio** backup and restore your own UNIX files, or transfer files to another UNIX user
 - **doscmd** utilities transfer UNIX files to and from DOS systems
 - **ftp** and **telnet** transfer files and access remote systems running the TCP/IP networking transport protocol
- Archiving and Restoring UNIX Files

The commands **tar** and **cpio** are used to backup files to UNIX-format floppies. The advantage of **tar** is simplicity, but it cannot be used with device files or links. The **cpio** command is more sophisticated, but harder to use. For most users, **tar** is sufficient for backing up their files—generally only the system administrator needs to back up device files. However, if you make use of linked files, you will probably need to use **cpio**.

The first step in creating a UNIX backup floppy is to format the disk:

```
format device_name
```

The device filename will differ depending on your operating system. On OpenServer systems, the name of the default 3.5" high-density floppy device is `/dev/rfd0`. On UnixWare 7 systems, the device name is `/dev/rdisk/f03ht`. If no device is specified, **format** will use the default device, which is defined in the file `/etc/default/format`.

B.1.1 Using tar

The **tar** command saves and restores files to and from an archive medium such as a floppy disk or tape:

```
tar [ key ] [ files ]
```

The *key* controls the actions of tar; *files* specifies which files will be backed up or restored. If a directory name is specified, the files and all subdirectory trees are included. **tar** uses the first floppy drive as the default device when reading and writing unless the **f** key specifies another device or file.

AV B-2

Creating tar Backups

```
$ lf
.kshrc      .profile      ex
.lastlogin  .sh_history  mail/

$ tar cv .
Volume ends at 1439K, blocking factor = 9K
seek = 1K      a ./profile 2K
seek = 3K      a ./kshrc 1K
seek = 4K      a ./lastlogin 0K
seek = 5K      a ./sh_history 1K
seek = 6K      a ./ex 320K
seek = 327K    a ./mail/message1 1K
seek = 328K    a ./mail/message2 1K
seek = 329K    a ./mail/message3 1K
seek = 330K    a ./mail/message4 1K
seek = 331K    a ./mail/message5 1K
seek = 332K    a ./mail/message6 1K
seek = 334K    a ./mail/message7 1K
$
```

ntro0B02

© The Santa Cruz Operation, Inc.

B.1.1.1 Creating tar backups

To backup files to a floppy with **tar**, use the **c** option:

```
tar cv [ files ]
```

- The **c** option creates a new archive, overwriting any existing contents of the file or media
- The **v** option displays each file as it is written
- Use the **r** option (in place of **c**) to add files to an existing archive without overwriting the contents
- The **f** option is used to specify a device (or file) to use instead of the default floppy

These examples show how backups are created with **tar** using the default floppy device:

- To backup the files *readme* and *report*, enter:

```
tar cv readme report
```

- To backup the subdirectory *Edit*, enter:

```
tar cv Edit
```

- Relative or absolute pathnames may be given as arguments to **tar**. This command creates an archive of the current directory and all subdirectories:

```
tar cv .
```

- To add files to an existing backup rather than overwriting the contents, use the **r** option:

```
tar rv file3 file4
```

- To view files on a backup archive, use the **tv** keys:

```
tar tv
```

AV B-3

tar Devices


```
$ tar
Usage: tar {txruc}[09vfbkelmnopwAFLTP] [tapefile] [blocksize]
[tapesize] files...
```

Key	Device	Block	Size (K)	Tape
0	/dev/rfd048ds9	18	360	No
1	/dev/rfd148ds9	18	360	No
2	/dev/rfd096ds15	10	1200	No
3	/dev/rfd196ds15	10	1200	No
4	/dev/rfd0135ds9	18	720	No
5	/dev/rfd1135ds9	18	720	No
6	/dev/rfd0135ds18	18	1440	No
7	/dev/rfd1135ds18	18	1440	No
8	/dev/rct0	20	0	Yes
9	/dev/rctmini	20	0	Yes
10	/dev/rdsk/fp03d	18	720	No
11	/dev/rdsk/fp03h	18	1440	No

ntro0B03© The Santa Cruz Operation, Inc.

Alternate devices are specified with one of the *built-in tar* device numbers, or using the **f** key with a device name. To see the list of device numbers, view the file */etc/default/tar*.

Key	Device	Block	Size(K)	Tape
0	/dev/rfd048ds9	18	360	No
1	/dev/rfd148ds9	18	360	No
2	/dev/rfd096ds15	10	1200	No
3	/dev/rfd196ds15	10	1200	No
.				
.				
.				
12	/dev/rdisk/fp03v21	10	20330	No

For example, backup files to a secondary 5.25" double-density drive using:

```
tar cv3 file1 file2
```

Exercise: Creating a tar backup

Use **tar** to backup the *mbox* file. View the contents of the floppy.

Create a file in your home directory that is a tar archive of the *Poems* subdirectory. Use relative pathnames.

AV B-4

Extracting tar Backups

```
$ mkdir restore
$ cd restore
$ tar xv2
x andybud, 1037 bytes, 2K
x en, 114 bytes, 1K
x fold, 719 bytes, 1K
x list, 440 bytes, 1K
x mail.reply, 462 bytes, 1K
x mailfolders/budgets, 655179 bytes, 640K
x mailhe, 913 bytes, 1K
x moreen, 134 bytes, 1K
x sendm, 204 bytes, 1K
x set, 127 bytes, 1K
x tar, 547 bytes, 1K
x tar.list, 838 bytes, 1K
x test, 515 bytes, 1K
x tilde, 482 bytes, 1K
x what_am_i, 352 bytes, 1K
x what_data_out, 352 bytes, 1K
$
```

ntro0B04

© The Santa Cruz Operation, Inc.

B.1.1.2 Extracting files from tar backups

To extract the contents of a **tar** floppy, use the **x** option:

```
tar xv [ files ]
```

- If the archive was created with full (absolute) pathnames, the files will be restored to their original location. Directories will be created by **tar** if necessary.
- If the archive was created with relative pathnames (such as **.**), the files can be restored to a different location. By default, files are written to the current directory.
- To determine whether the archive was created with absolute or relative pathnames, list the files with **tar tv**.
- To override full pathnames, use the **A** option. In this example, the file */README* is written to */u/charles* rather than */*:

```
$ pwd
```

```
/u/charles
```

```
$ tar tv
```

```
rw-r--r-- 13079/1040    713 Dec 17 11:10 1997 /README
```

```
$ tar xvA
```

```
tar: suppressing absolute pathnames
```

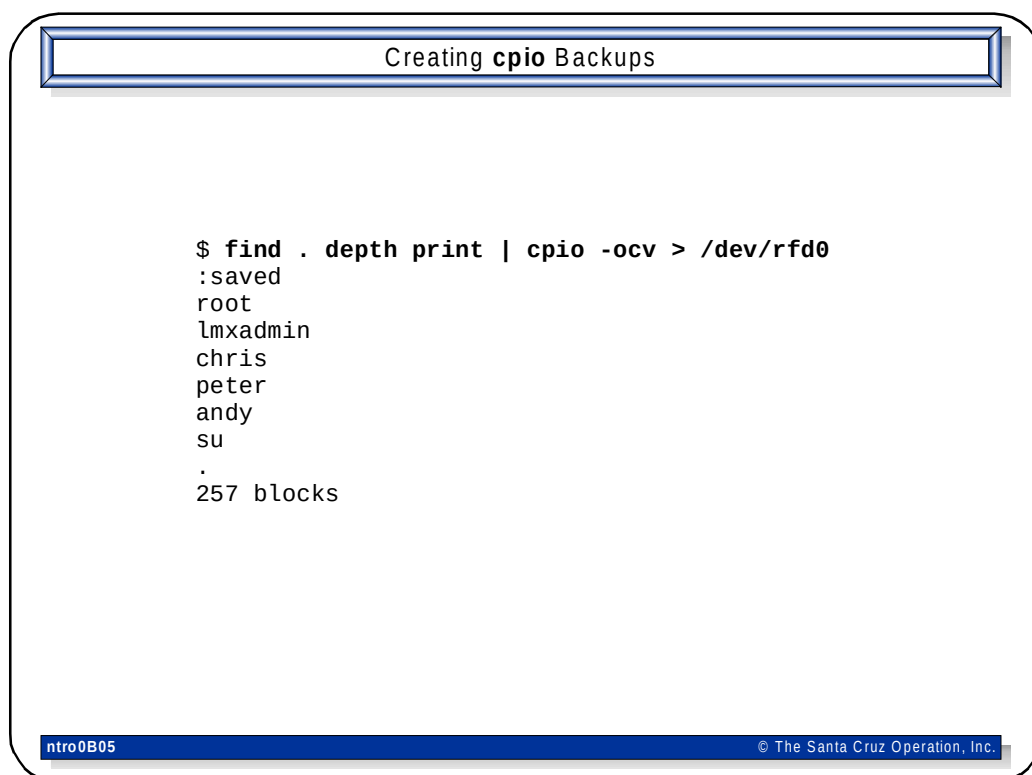
```
x README, 713 bytes, 1K
```

Exercise: Restoring files from tar backups

Create a *temp* subdirectory and restore the *mbox* file from the floppy to *temp*.

Remove the *Poems* directory. Restore *Poems* from your archive file.

AV B-5



B.1.2 Using **cpio**

The **cpio** command is a very flexible, powerful, archival tool. In addition to the basic functions of copying files to and from floppies, it supports:

- Archiving special files, such as devices and links
- Using UNIX commands and wildcard characters for specifying input
- Transferring files between different types of UNIX systems

There are two main options to **cpio** for creating floppy (or tape) archives:

cpio -o (copy out) reads a list of pathnames from the standard input and copies those files to the standard output

cpio -i (copy in) extracts files from the standard input

The **cpio** command is a filter which reads from the standard input (*stdin*) and writes to the standard output device (*stdout*). When creating backups to floppies (or other devices), *stdout* is a device file, such as */dev/rfd0*. When extracting files from a floppy, *stdin* is a device file.

B.1.2.1 Creating cpio backups

To backup files to a floppy with **cpio**, use the **-o** (output) mode:

```
command | cpio -o[ options ] > device
```

- Wildcard characters can be used to archive files matching specified patterns. For example, to archive all filenames starting with the string *prop*, enter:

```
ls prop* | cpio -ocv > /dev/rfd0
```

- The input for **cpio** comes from any UNIX command which generates lists of files, allowing precise specification of the files to be archived. For example, to archive all the files modified within the last seven days, use **find** with the **mtime** option:

```
find . -depth -print -mtime -7 | cpio -ocv > /dev/rfd0
```

Exercise: Creating a cpio backup floppy

Back up the files in the *Edit* directory. View the files on the floppy.

AV B-6



B.1.2.2 Restoring files using **cpio**

To extract files from a **cpio** backup use the **-i** (input) mode:

```
cpio -i[ options ] < device
```

These examples show how files are extracted from **cpio** archives:

- To restore the contents of a floppy backup to your current directory:

```
cpio -icvd < /dev/rfd0
```

- To display a list of files on a backup floppy without extracting the contents, enter:

```
cpio -itv < /dev/rfd0
```

- To extract only the files in the directory *info*, enter:

```
cpio -icvd "info/*" < /dev/rfd0
```

Exercise: Restoring files from a **cpio backup**

Remove the *Edit* directory. Restore its contents from the **cpio** backup floppy.

B.2 Using DOS Floppies

SCO UNIX systems provide a set of commands for formatting, writing, and reading DOS floppies. This allows you to move files between your DOS and UNIX systems.

To simplify the process of transferring files, UNIX automatically maps DOS floppy device **a:** to the UNIX floppy device file */dev/fd0*.

The **doscmd** utilities supported by SCO UNIX include:

dosformat	Formats a DOS floppy
doscp	Copies DOS files to/from UNIX
dosdir	Lists files in the standard DOS format
dosmkdir	Creates a DOS directory
dosrm	Removes a file from a DOS floppy
doscat	Displays the contents of DOS text files to the screen

These examples use the **doscmd** utilities to transfer files to and from DOS format floppies:

- To format the floppy in the first 3.5” high-density drive on an SCO OpenServer system, enter:

```
dosformat /dev/fd0135ds18
```

Note that **dosformat** cannot be used with **a:** substituted for the device name; nor can it be used with the short form of the device name, such as */dev/fd0*.

- To copy the file *readme* to a DOS floppy, enter:

```
doscp readme a:
```

- To list the files on a DOS floppy, enter:

```
dosdir a:
```

- To create the directory *projects* on a DOS floppy, enter:

```
dosmkdir a:projects
```


- To remove the file *reports* in the subdirectory *projects*, enter:

dosrm a:projects/reports

- To display the contents of the *readme* text file, enter:

doscat a:readme

When files are moved from an SCO UNIX system to a DOS system, these filesystem differences must be considered:

Filenames	For best results, files that will be used on both UNIX and DOS systems should follow the DOS 8.3 naming convention. The doscmbd utilities will automatically convert long names, but the resulting filenames may be difficult to use.
Pathnames	When using the doscmbd utilities from UNIX, a forward slash (/) is used in a pathname. When reading from a DOS floppy, use the standard DOS backslash (\) convention when specifying pathnames.
Wildcards	Only the * and ? wildcard characters can be used with doscmbd utilities

B.2.1 DOS file conversion

When files are transferred from DOS to UNIX, the **doscmbd** utilities **doscat** and **doscp** will automatically strip the ^M (carriage return) characters from text files. When text files are transferred to DOS, ^M characters are inserted before each linefeed character.

DOS text files can be moved to a UNIX system using other utilities which do not automatically convert text files. For example, files may be transferred from a DOS system over the network using a TCP/IP utility. When UNIX utilities (such as **vi**) are used directly on DOS text files, the ^M characters are visible.

To remove the carriage return characters, use the UNIX command **dtod** (dos-to-unix):

dtod readme.txt > readme

To insert carriage return characters in a UNIX text file so it can be used under DOS, use **xtod**:

xtod readme > readme.txt

AV B-7

Connecting with ftp

```
$ ftp support
Connected to support.
220
220 support.acme.com FTP server (Version 2.1WU(1)) ready.
Name (support:andy): andy
331 Password required for andy.
Password:
230 User andy logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp>
```

ntro0B07

© The Santa Cruz Operation, Inc.

AV B-8

ftp Commands

get <i>remote-file</i> [<i>local-file</i>]	Retrieves a copy of a file from the remote host
put <i>local-file</i> [<i>remote-file</i>]	Puts a copy of a local file on the remote host
mput <i>local-files</i>	Same as put , but allows use of wildcards
mget <i>remote-files</i>	Same as get , but allows use of wildcards
cd <i>remote-directory</i>	Changes the working directory on the remote host
lcd [<i>local-directory</i>]	Changes the working directory on the local system
ls [<i>remote-directory</i>]	Displays the contents of a remote directory
mkdir <i>remote-directory</i>	Creates a new directory on the remote host
delete <i>remote-file</i>	Deletes a file on the remote host
binary	Sets the transfer type to support binary (data) files
! [<i>command</i>]	Executes a command on the local system
? [<i>command</i>]	Displays a help screen

ntro0B08

© The Santa Cruz Operation, Inc.

B.3 Using TCP/IP

TCP/IP is a widely-used networking suite that allows a high-degree of interoperability among many diverse types of computers.

Many large networks, comprised of all types of hardware and operating systems software, conform to the TCP/IP protocols, including the Internet. TCP/IP allows any computer on the local network or the Internet (referred to as a *host* or *node*) to communicate with any other.

In addition to the many administrative services provided by TCP/IP, the **ftp** and **telnet** commands allow users to access remote systems.

Depending on how your network is configured, the **rsh**, **rcmd** and **rcp** utilities can also be used to access resources on the TCP/IP network.

B.3.1 Transferring files with ftp

The **ftp** command is the user interface to the Internet-standard File Transfer Protocol. **ftp** allows a user to transfer files to and from a remote network site by connecting to the specified host and providing a command interpreter for issuing requests. The syntax is:

ftp *host*

- **ftp** will only grant access to users with valid accounts on the remote system
- Unless otherwise specified on the **ftp** command line, the current working directory is the default for the commands **put**, **get**, and **ls**.
- When used without arguments, the **cd** and **lcd** commands change to the user's home directory on the remote, and local systems, respectively.
- Use **quit** to exit from an **ftp** session

These examples demonstrate some common uses of **ftp**:

- To transfer the file *Proposal1* from the current working directory on the local system to the directory *Reports/Chris* on the remote host:

```
ftp>cd Reports/Chris  
ftp>put Proposal1
```

Another way to accomplish this transfer is:

```
ftp>put Proposal1 Reports/Chris
```

- To create a directory *timesheets* on the remote host and transfer all files in the local directory with names starting with the string *Jan*, enter:

```
ftp>mkdir timesheets
ftp>cd timesheets
ftp>mput Jan*
```

- To get the file *calendar* from the remote host and copy it to the file *1998* in the local directory, enter:

```
ftp>get calendar 1998
```

- To transfer the data file *budget.xls*, first set **binary** mode:

```
ftp>binary
ftp>get budget.xls
```

Exercise: Using ftp

Start **ftp** with the *hostname* provided for this exercise. Login using the account information provided. What is your current directory on the remote system; on the local system?

Get a file and verify that it has been transferred to your local system.

AV B-9

Using telnet

```
$ telnet ftpsrvr
Trying 130.128.209.xxx...
Connected to ftpsrvr
Escape character is '^]'.

SCO OpenServer(TM) Release 5 (ftpsrvr.euro.com) (ttyp5)

login: andy
Password:
Last successful login for andy: Thu Mar 15 22:28:59 1998 on
tty06
Last unsuccessful login for andy: NEVER

                                SCO OpenServer(TM) Release 5

you have mail
TERM = (ansi)
Terminal type is ansi
$
```

ntro0B09 © The Santa Cruz Operation, Inc.

AV B-10

telnet Command Prompt

```
$ hostname
ftpsrvr.euro.com

$ [ ]
telnet>!hostname
mysystem [ ]
$

telnet>!
$ uname -a
SCO_SV mysystem 3.2 2 i386
$ [ ] d [ ]

$ hostname
ftpsrvr.euro.com
```

ntro0B10 © The Santa Cruz Operation, Inc.

B.3.2 Remote login (telnet)

The **telnet** command communicates with another host and provides the user access to a login session on the remote host.

To begin the connection session, enter:

telnet *host*

- When **telnet** connects to a machine, it prompts for your user name and password on that machine so you can log in
- You can do any kind of work at the command line on the remote machine that you would do when logged in to a local system
- While connected through **telnet**, access to the local system is available using the **telnet** escape character (^]) to enter command mode:
 - Shell commands can be executed on the local system from the **telnet>** prompt using **!command**
 - From the **telnet** prompt, use the **!** command with no argument to start an interactive subshell on the local system
 - Return to the remote shell by typing **CTRL d** **ENTER**
- When finished, log out from the remote machine by typing **exit** and return to your local login shell

Exercise: Using telnet

Start **telnet** with the *hostname* provided for this exercise. Log in using the account information provided.

Display the remote host name. Run the **uname** command on the local system.

Log off the remote system.

B.3.3 Finding system names

To see a list of names of other machines on your network, use the **ruptime** command:

```
$ ruptime
seattle    up 26 days, 15:00, 1 user, load average: 2.08, 2.01, 2.00
berlin     down          4:07
sydney     up 43 days  01:27, 4 users, load average: 3.00,
```

To find the network name of your local system, use **hostname**:

```
$ hostname
moscow.tia.com
```

You may also be able to view a list of network system names stored in the file */etc/hosts* (depending on permissions). This file is used by programs that need to translate between host names and Internet addresses.

B.4 Relevant Commands, Files, and Syntax

These commands, files, and devices were presented in this lesson.

Relevant commands

tar	Archives files
cpio	Copies file archives
doscmd	Suite of commands that manipulate DOS files
dosformat	Formats a DOS floppy
doscp	Copies DOS files to/from UNIX
dosdir	Lists DOS files
dosrm	Removes DOS files
dosmkdir	Creates a DOS directory
doscat	Displays the contents of DOS text files
dtox	Converts DOS text files to UNIX
xtod	Converts UNIX text files to DOS
dtype	Determines disk type
ftp	Transfers files over TCP/IP networks
telnet	Logs users onto remote systems
ruptime	Lists the systems on the network
hostname	Displays the network name of the local host

Relevant files

<i>/etc/default/tar</i>	Defines default devices for tar
<i>/etc/default/msdos</i>	Defines default devices for doscmd utilities Relevant devices
<i>/dev/fd0</i>	Block interface to the first floppy drive on SCO OpenServer systems
<i>/dev/rfd0</i>	Character interface to the first floppy drive on SCO OpenServer systems
<i>/dev/rfd0135ds18</i>	Character interface to the first 3.5 inch high-density (1.44MB) floppy drive on SCO OpenServer systems
<i>/dev/dsk/f03ht</i>	Block interface to the 3.5 inch high-density (1.44MB) floppy drive on UnixWare 7 systems
<i>/dev/rdsk/f03ht</i>	Character interface to the 3.5 inch high-density (1.44MB) floppy drive on UnixWare 7 systems
<i>/etc/default/msdos</i>	Defines default devices for doscmd utilities
<i>/etc/hosts</i>	Lists accessible network systems