HP Computer/Instrument Systems
Training Course

HP-UX BASICS II

Student Workbook

H2572-90001
Printed in U.S.A. 11/90

Notice

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

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

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

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

This document contains proprietary information which is protected by copyright. All rights reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company.

Copyright© 1990 by HEWLETT-PACKARD COMPANY

MS-DOS® is a U.S. registered trademark of Microsoft Corporation.

UNIX is a registered trademark of AT&T in the U.S.A. and in other countries.

Product Development
Application Support Division
100 Mayfield Avenue
Mountain View, CA 94043 U.S.A.

Contents

Overview 1
Course Description 1
Student Performance Objectives 1
Student Profile and Prerequisites 2
Curriculum Path 2
Agenda 3

Course Preview 5

Module 1 — Introduction 1-1
Objectives 1-1
1-1. SLIDE: Notation Conventions 1-2
1-2. SLIDE: Prerequisites of This Class 1-3
1-3. SLIDE: What We Will and Won't Cover 1-4
1-4. SLIDE: A Brief History of the UNIX Operating System 1-5
1-5. SLIDE: What Is an Operating System? 1-7
1-6. SLIDE: What Is HP-UX? 1-8
1-7. SLIDE: What HP-UX Provides 1-9

Module 2 — The Shell 2-1
Objectives 2-1
2-1. SLIDE: What Is the Shell? 2-2
2-2. SLIDE: The Local Data Area 2-4
2-3. SLIDE: Shell Variables 2-5
2-4. SLIDE: Setting Shell Variables 2-7
2-5. SLIDE: Referencing Shell Variables 2-9
2-6. SLIDE: The Environment 2-11
2-7. SLIDE: The set and unset Commands 2-12
2-8. SLIDE: The env Command 2-13
2-9. SLIDE: The export Command 2-14
2-10. LAB: The Shell 2-15

Contents

Module 3 — Command Execution 3-1
Objectives 3-1
3-1. SLIDE: How a Command Is Executed 3-2
3-2. SLIDE: Shell Intrinsics versus HP-UX Commands 3-3
3-3. SLIDE: Looking for Commands — whereis 3-4
3-4. SLIDE: The find Command 3-6
3-5. SLIDE: Who Am I? — The id Command 3-7
3-6. SLIDE: The su Command 3-8
3-7. SLIDE: The newgrp Command 3-10
3-8. SLIDE: How Is a Process Formed? 3-11
3-9. SLIDE: The ps Command 3-13
3-10. SLIDE: Background Processing 3-15
3-11. SLIDE: An Example — & 3-17
3-12. SLIDE: Command Substitution 3-18
3-13. SLIDE: Some Related Commands 3-19
3-14. SLIDE: The nice Command 3-20
3-15. SLIDE: The nohup Command 3-21
3-16. SLIDE: The kill Command 3-22
3-17. LAB: Command Execution 3-23

Module 4 — File Name Generation 4-1
Objectives 4-1
4-1. SLIDE: Introduction — Special Characters 4-2
4-2. SLIDE: File Name Generating Characters 4-3
4-3. SLIDE: File Name Generation and Dot Files 4-4
4-4. SLIDE: File Name Generation — ? 4-5
4-5. SLIDE: File Name Generation — [ ] 4-6
4-6. SLIDE: File Name Generation — * 4-7
4-7. SLIDE: Examples — File Name Generation 4-8
4-8. SLIDE: File Name Generation and Quoting 4-9
4-9. SLIDE: Special Characters We Have Seen 4-10
4-10. SLIDE: There Are Three Ways to Escape 4-11
4-11. SLIDE: Quoting — \ 4-13
4-12. SLIDE: Quoting — ' 4-14
4-13. SLIDE: Quoting — " 4-15
4-14. SLIDE: A Summary 4-16
4-15. SLIDE: File Name Generation and Quoting Review 4-17
4-16. LAB: File Name Generation 4-18

Contents

Module 5 — Input and Output Redirection 5-1
Objectives 5-1
5-1. SLIDE: stdin, stdout, and stderr 5-2
5-2. SLIDE: Input Redirection — < 5-4
5-3. SLIDE: Output Redirection — > and > > 5-5
5-4. SLIDE: Error Redirection — 2 > and 2 > > 5-6
5-5. SLIDE: What Is a Filter? 5-7
5-6. SLIDE: The cat Command, Revisited 5-8
5-7. SLIDE: The sort Command 5-9
5-8. SLIDE: The grep Command 5-10
5-9. SLIDE: The wc Command 5-11
5-10. LAB: Input and Output Redirection 5-12

Module 6 — Pipelines 6-1
Objectives 6-1
6-1. SLIDE: Motivation 6-2
6-2. SLIDE: The | Symbol 6-3
6-3. SLIDE: Pipelines versus Input and Output Redirection 6-4
6-4. SLIDE: Some Useful Filters 6-5
6-5. SLIDE: The tee Command 6-6
6-6. SLIDE: The cut Command 6-7
6-7. SLIDE: The pr Command 6-9
6-8. SLIDE: Paging from a Pipeline 6-11
6-9. SLIDE: Printing from a Pipeline 6-12
6-10. SLIDE: Redirecting in a Pipeline 6-13
6-11. LAB: Pipelines 6-14

Module 7 — An Introduction to Shell Programming 7-1
Objectives 7-1
7-1. SLIDE: Introduction to Shell Programming 7-2
7-2. SLIDE: Arguments To Shell Programs 7-3
7-3. SLIDE: Some Special Shell Variables — # and * 7-4
7-4. SLIDE: The shift Command 7-5
7-5. SLIDE: The read Command 7-6
7-6. SLIDE: The expr Command 7-7
7-7. SLIDE: Miscellaneous Techniques 7-8
7-8. SLIDE: Interactive Shell Commands 7-9
7-9. SLIDE: .profile 7-10
7-10. LAB: An Introduction to Shell Programming 7-11

Contents

Module 8 — Shell Programming: Branches 8-1
Objectives 8-1
8-1. SLIDE: Return Codes 8-2
8-2. SLIDE: The test Command 8-3
8-3. SLIDE: The test Command — File Tests 8-5
8-4. SLIDE: The test Command — String Tests 8-7
8-5. SLIDE: The test Command — Numeric Tests 8-9
8-6. SLIDE: The test Command — Other Operators 8-11
8-7. SLIDE: The if Construct 8-13
8-8. SLIDE: The if Construct — Examples 8-14
8-9. SLIDE: The case Construct 8-15
8-10. SLIDE: The case Construct — Pattern Examples 8-16
8-11. SLIDE: The case Construct — Examples 8-17
8-12. LAB: Shell Programming — Branches 8-18

Module 9 — Shell Programming: Loops 9-1
Objectives 9-1
9-1. SLIDE: Loops — An Introduction 9-2
9-2. SLIDE: The while Construct 9-3
9-3. SLIDE: The while Construct — Examples 9-4
9-4. SLIDE: The until Construct 9-5
9-5. SLIDE: The until Construct — Examples 9-6
9-6. SLIDE: The for Construct 9-7
9-7. SLIDE: The for Construct — Examples 9-8
9-8. SLIDE: The break, continue, and exit Commands 9-9
9-9. LAB: Shell Programming — Loops 9-10

Module 10 — Shell Programming: Signals and Traps 10-1
Objectives 10-1
10-1. SLIDE: What Is a Signal? 10-2
10-2. SLIDE: What Is a Trap? 10-3
10-3. SLIDE: The kill Command, Revisited 10-4
10-4. SLIDE: The trap Command 10-5
10-5. SLIDE: Ignoring Signals 10-7
10-6. SLIDE: Placement of the trap Command 10-8
10-7. LAB: Shell Programming — Signals and Traps 10-9

Contents

Module 11 — Offline File Storage 11-1
Objectives 11-1
11-1. SLIDE: Tape Usage 11-2
11-2. SLIDE: The tar Command 11-3
11-3. SLIDE: The find Command, Revisited 11-4
11-4. SLIDE: The cpio Command 11-5
11-5. SLIDE: The tcio Command 11-7
11-6. LAB: Offline File Storage 11-8

Course Review CR-1

Appendix A — Additional Features of the Korn Shell A-1
Objectives A-1
A-1. SLIDE: History of Shells A-2
A-2. SLIDE: Features of the K Shell A-4
A-3. SLIDE: Aliasing A-5
A-4. SLIDE: Command-Line Editing A-7
A-5. SLIDE: File Name Completion A-8
A-6. SLIDE: Command History — Part I A-9
A-7. SLIDE: Command History — Part II A-10
A-8. LAB: Additional Features of the Korn Shell A-11

Appendix B — Solutions B-1

Contents

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Overview

Course Description

This course is designed to be the second course in the BASICS HP-UX curriculum taught by Hewlett-Packard. It assumes that the student knows the material covered in BASICS I (51489). BASICS I covers the HP-UX file system, an introduction to the shell command language, the vi editor, and various utilities. BASICS I assumes that the student knows nothing about HP-UX or any other operating system or utility based on the UNIX operating system.

Student Performance Objectives

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

- Describe HP-UX as an operating system.
- Describe the job of the shell.
- Set and use shell variables for typing aids.
- Move variables from the local data area to the environment.
- Examine the contents of many preset shell variables.
- Describe the difference between a shell intrinsic command and an HP-UX command.
- Explain the search used by the shell to find a command.
- Use the whereis and find commands to locate files in the HP-UX file system.
- Explain how a process is created (fork and exec).
- Use the ps command to monitor running processes.
- Explain how the shell knows who is executing a command.
- Start a process running in the background.
- Start a background process that is immune to the hangup (logoff) signal.
- Stop processes from running by sending them signals.
- Use grave accents (back single quotes) to assign the output of a process to a variable.
- Use the expr command for simple arithmetic and string matching.
- Use metacharacters to generate file names on the command line.
- Save typing by using file name generating characters.
- Name files such that file name generating characters will be more useful.
- Change the destination for the output of many HP-UX commands.
- Change the destination for the error messages generated by many HP-UX commands.
- Change the source of the input to many HP-UX commands.
- Define a filter.
- Use some elementary filters such as sort, grep, and wc.
- Construct a pipeline to take the output from one command and make it the input for another.
- Use the tee, cut, and pr filters.
- Describe the use of the positional parameters 1–9.
- Describe the use of the special shell variables *, #, and $.
- Describe the use of the shift and read commands.
- Describe the use of return codes for conditional branching.
- Describe the use of the test command for setting return codes.
- Use the if and case constructs for branching in a shell program.

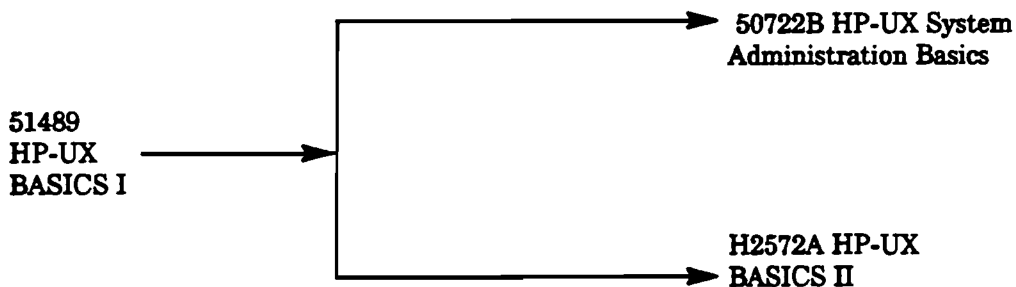
Overview

- Use the while construct to repeat a section of code while some condition remains true.
- Use the until construct to repeat a section of code while some condition remains false.
- Use the iterative for construct to repeat a sequence of commands a known number of times (based on the number of input strings).
- Describe the use of the kill and trap commands.
- Use the trap command for producing interrupt-driven code.
- Use the tar command for storing files to tape.
- Use the find, cpio, and tcio commands for storing files to tape.
- Retrieve files that were stored through tar or cpio.
- Set up aliases to save typing.
- Use the file name completion feature of the K shell.
- Recall and edit commands that have already been typed.

Student Profile and Prerequisites

The ideal student would be one who has taken HP-UX BASICS I or who has mastered its materials. BASICS I covers the HP-UX file system, and introduction to the shell command language, the vi editor, and various utilities included in HP-UX.

Curriculum Path



Overview

Agenda

Day 1, AM	Module 1 — Introduction Module 2 — The Shell Module 3 — Command Execution
Day 1, PM	Module 4 — Filename Generation Module 5 — Input/Output Redirection
Day 2, AM	Module 6 — Pipelines Module 7 — An Introduction to Shell Programming
Day 2, PM	Module 8 — Shell Programming: Branches Module 9 — Shell Programming: Loops
Day 3, AM	Module 10 — Shell Programming: Signals and Traps Module 11 — Offline File Storage Finishing Touches

Overview

Course Preview

HP-UX BASICS II

As you begin this course, please indicate below whether you feel extremely confident (EC), confident (C), fairly sure (FS), not sure (NS), or unable (U) to perform the following.

1.	Logging in and out of HP-UX.	EC C FS NS U
2.	Using shell variables.	EC C FS NS U
3.	Navigating the file system tree.	EC C FS NS U
4.	Understanding permissions and ownership of files.	EC C FS NS U
5.	Using simple file manipulation commands such as cp, ln, and mv.	EC C FS NS U
6.	Understanding metacharacters for file name generation.	EC C FS NS U
7.	Understanding the uses of some of the many directories under HP-UX.	EC C FS NS U
8.	Understanding command execution.	EC C FS NS U
9.	Using vi.	EC C FS NS U
10.	Using the advanced features of the Korn shell.	EC C FS NS U
11.	Using input/output redirection.	EC C FS NS U
12.	Using pipelines.	EC C FS NS U
13.	Using quoting mechanisms to escape the meaning of special characters.	EC C FS NS U
14.	Running multiple processes at one time.	EC C FS NS U
15.	Writing nontrivial shell programs.	EC C FS NS U
16.	Making copies of files to tape.	EC C FS NS U

Course Preview

Module 1 — Introduction

Objectives

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

- Describe the topics that this course will and will not cover.
- Describe HP-UX as an operating system.

1-1. SLIDE: Notation Conventions

Notation Conventions

- [Return]
- [CTRL]-[d]
- command_name
- *user text*
- *[optional items]*
- "filenames"

H2572 1-1. 1 © 1990 Hewlett-Packard Company

Student Notes

This section describes the few notation conventions that are used throughout the course.

On some of the slides which show terminal sessions or fragments of terminal sessions, [Return] is used to denote the pressing of the [Return] key, and [CTRL]-[d] is used to denote the typing of a control-d. These notations are used early in the course to explicitly describe what the user types. As the course advances, it is assumed that you have an understanding of the use of return and control-d, and, therefore, they are not made explicit in the slides.

Italics are used to distinguish user-provided text from adjacent literal text. In describing the syntax of commands, square brackets ([ ]) are used to delimit optional items. This corresponds with the use of square brackets in the *HP-UX Reference Manual*.

The slides that describe the syntax and options of commands are not necessarily complete. Only the most commonly used forms and options are shown.

The text and slides are self-explanatory.

1-2. SLIDE: Prerequisites of This Class

Prerequisites of This Class

It is assumed that you already know how to:

- Log into HP-UX.
- Navigate the HP-UX file system.
- Move, copy, rename, and remove HP-UX files.
- Create and remove directories.
- Use the vi editor to do basic editing of HP-UX files.

H2572 1-2 2 © 1990 Hewlett-Packard Company

Student Notes

The slide shows the topics that were covered in the prerequisite to this course. If some of these concepts are unfamiliar to you, please let your instructor know.

1-3. SLIDE: What We Will and Won't Cover

What We Will and Won't Cover

We will cover:

- The shell.
- Processes and command execution.
- File name generation.
- I/O redirection.
- Pipelines.
- Quoting.
- Shell programming.
- Offline storage.

We won't cover:

- All HP-UX commands.
- HP-UX internals.
- The C language.
- System administration.
- Tuning and performance.
- System calls.
- Programming tools.
- Networking and LANs.

H2572 1-3. 3 © 1990 Hewlett-Packard Company

Student Notes

The slide shows the topics that will be covered in this course. If there are any additional topics that you would like to know about, ask your instructor.

1-4. SLIDE: A Brief History of the UNIX Operating System

A Brief History of the UNIX Operating System

1969	AT&T Bell Labs	UNIX operating system
Early 1970s	University of California at Berkeley	BSD
Early 1980s	Hewlett-Packard	HP-UX

H2572 1-4. 4 © 1990 Hewlett-Packard Company

Student Notes

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

In the early 1970s, the UNIX operating system was licensed to universities and gained a wide popularity because of the following:

- It was small.
- It was flexible.
- It was cheap.

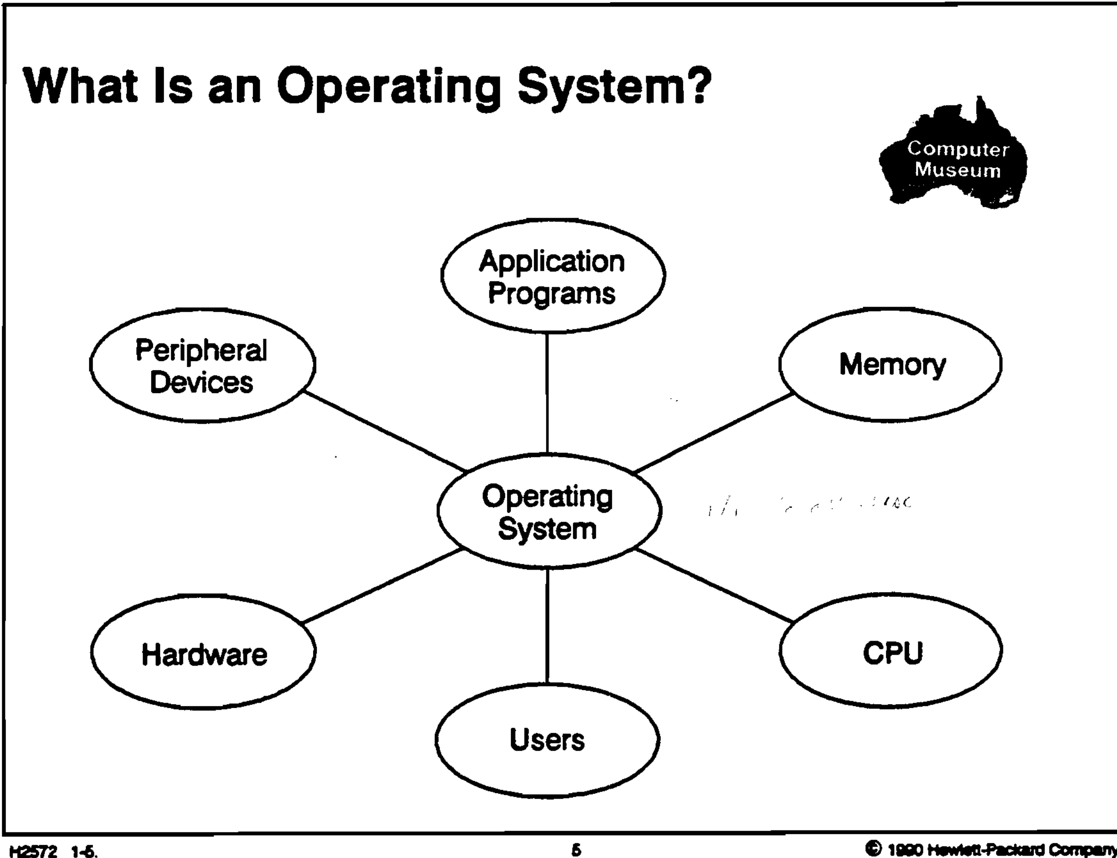
These advantages offset the disadvantages of the system at the time:

- It was "buggy."
- It had little or no documentation.
- It had no support.

When the UNIX operating system reached the University of California at Berkeley, the Berkeley users created their own version of the system with many improvements and a few "disimprovements."

AT&T recognized the potential of the operating system and started licensing the system commercially. Berkeley, meanwhile, made more improvements and added more features. Both systems went through several versions.

1-5. SLIDE: What Is an Operating System?



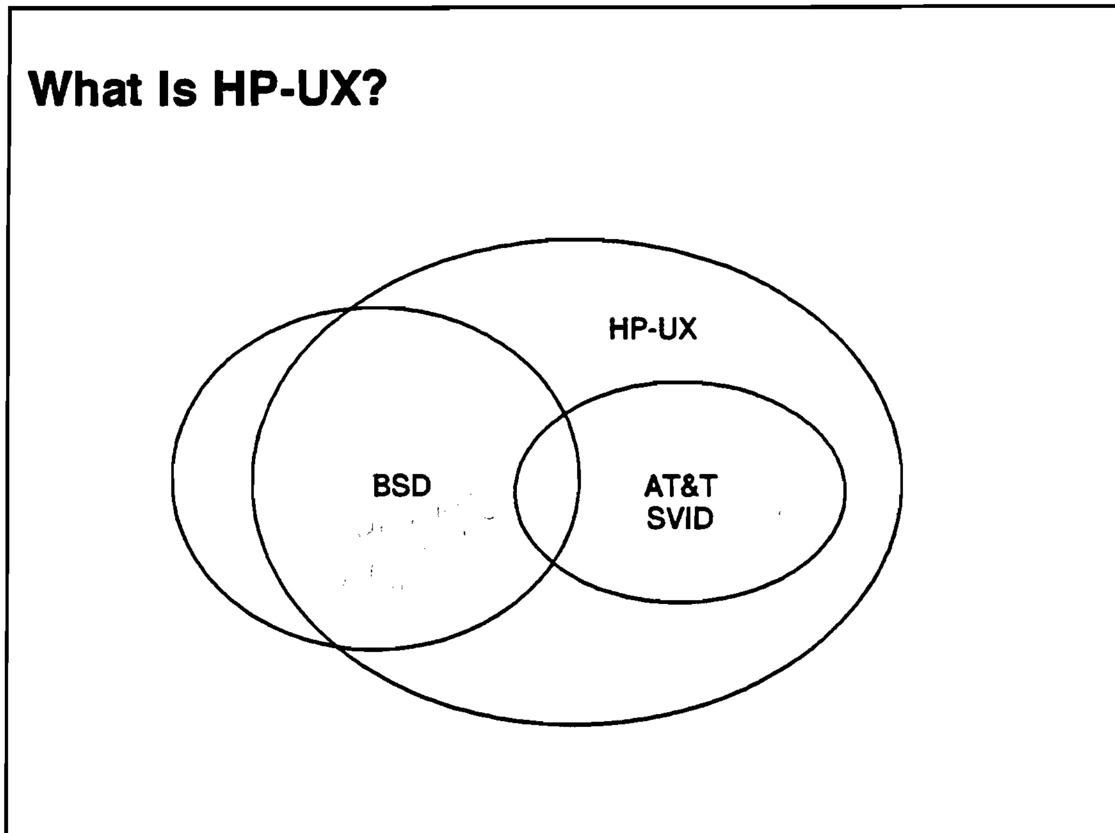
Student Notes

An **operating system** is a piece of software that controls the use of the computer. The operating system (o.s.) takes the demands placed on it by application programs, users, and so forth and fulfills those demands by allocating the system resources (such as memory, central processing unit, and peripheral devices).

In a time-sharing system, such as HP-UX, this is a very complex job. Operating system decisions address issues such as whose turn it is to use the CPU, which devices (printers, terminals, disk drives, and the like) need to be used, and which users or application programs are more important.

HP-UX is an operating system that is optimized for a number of different applications. These include program development, data acquisition, and instrument control. The operating system itself is tunable for a particular application, but this is discouraged because it has been finely tuned already.

1-6. SLIDE: What Is HP-UX?



Student Notes

HP-UX is the AT&T System V UNIX operating system plus "HP value added." HP value added includes Hewlett-Packard capabilities such as graphics, native language support, and features from the UNIX operating system developed at the University of California at Berkeley and elsewhere.

AT&T has established the System V Interface Definition (SVID) to describe System V compatibility. HP-UX conforms to this definition.

1-7. SLIDE: What HP-UX Provides

Student Notes

HP-UX provides an interactive, programmer-friendly working environment that includes the following:

- A powerful command-line interpreter (the shell).
- A rich command language.
- A convenient file system.
- A powerful programming language (C).
- A set of very powerful programming tools.
- An extensible environment that is easily customized.
- Many utilities: print spooler, bc, and so forth.

Module 2 — The Shell

Objectives

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

- Describe the job of the shell.
- Set and use shell variables for typing aids.
- Move variables from the local data area to the environment.
- Examine the contents of many preset shell variables.

Module 2 — The Shell

2-1. SLIDE: What Is the Shell?

What Is the Shell?

A command interpreter that:

- Allows the user to set up his or her environment.
- Issues the prompt.

When a command is entered, the shell:

- Does variable substitution.
- Expands metacharacters.
- Handles input and output redirection and pipelines.
- Performs command substitution.
- Executes the command.

H2572 2-1. 8 © 1990 Hewlett-Packard Company

Student Notes

When you are logged into an HP-UX system, you are issued a prompt. This prompt defaults to a $ symbol in the case of the Bourne and K shells. The default prompt for the C shell is the percent sign (%).

A **shell** is a command-line interpreter and a high-level programming language. As a command interpreter, it processes and executes the command lines that you enter in response to its prompt. As a programming language, it processes groups of commands stored in files called **shell scripts.** Shell scripts will be examined later in the course. But first, let's look at how the shell acts as a sophisticated command interpreter.

When you enter a command line at the shell prompt, the shell performs the following functions for you:

- It substitutes shell variable values for dereferenced variables.
- It generates file names from metacharacters.

- It handles I/O redirection and pipelines.
- It performs command substitution.
- It decides whether the command is a shell intrinsic or an HP-UX executable program.
- It searches for a command if that command is an executable program.

We will look at each of these functions throughout the course, starting with setting and dereferencing shell variables.

2-2. SLIDE: The Local Data Area

Student Notes

The shell uses string variables (variables that are able to take on the value of a string of characters) to represent numbers as well as text. Built into the shell are two areas of memory for use with shell variables: the **local data area** and the **environment.** By default, when a shell variable is set, memory is allocated from the local data area. The variables in this area are private to the current shell. That is, any child processes created (forked) by the current parent shell will not have access to these variables. However, as we will see, variables can be moved into the environment with the export command. Variables in the environment can be accessed by child processes.

2-3. SLIDE: Shell Variables

Shell Variables

- A shell variable is a name that stands for a value.
- All shell variables are initialized to NULL.
- Some shell variables are set for you.
- Users can define and use their own shell variables.

H2572 2-3. 10 © 1990 Hewlett-Packard Company

Student Notes

A **shell variable** is similar to a variable in algebra. It is a name that can stand for a value. All shell variables, by default, are initialized to *NULL* (nothing). These variables can then be set to any string of characters that you desire. When dereferenced (accessed), the value of the variable is used instead of the variable name itself.

There are several special shell variables that are already set for you. These variables can be changed to customize your environment. The following are some of the variables that may be set for you:

HOME	Defines the user's login and original working directory. This is the default directory used for the cd command.
PATH	Defines your execution search path.
PS1	Defines your primary shell prompt.

PS2	Defines your secondary shell prompt.
TERM	Defines your terminal type.
PWD	Defines your Present Working Directory.

There are, of course, many other predefined shell variables. They are all explained in section 1 of the *HP-UX Reference Manual* for your particular shell: sh(1), csh(1), or ksh(1).

2-4. SLIDE: Setting Shell Variables

Setting Shell Variables

Syntax:

name=value

Examples:

```
$ MY_NAME_IS=bullwinkle
$ MESSY=/usr/man/man1/ls.1
$ var5="HEY, Rocky!"
```

H2572 2-4. 11 © 1990 Hewlett-Packard Company

Student Notes

Variables are places to store information that you want to refer to later by name. Specifically, shell variables are used to hold information that you will later want to use as part of a shell command line.

The construct for assigning a value to a shell variable is the following:

name=value

This can be typed at the terminal after a shell prompt. Notice that there is no white space either before or after the equal sign (=). This ensures that the shell will not try to interpret the assignment as a command invocation. The name of a shell variable must start with a letter and can only contain letters, digits, and underscores (_). For example, the following is a legal assignment:

```
his_name=boris
```

It assigns the string "boris" to the variable "his_name". We refer to this operation as *setting* the variable his_name.

It is important that we distinguish between the **name** of a shell variable and the **value** of a shell variable. When we set a variable by performing an assignment statement, such as:

```
MESSY=/usr/man/man1/ls.1
```

we are telling the shell to remember the name "MESSY" and to remember that it has the value "/usr/man/man1/ls.1". The shell does this internally in the local data area.

This slide shows how to assign a value to a variable. We will see how to use this variable/value pair in the next slide.

Notice that although the name of a shell variable may contain both uppercase and lowercase letters, many HP-UX users prefer using uppercase letters in shell variable names to make them distinguishable from command names in shell scripts. Command names in HP-UX are traditionally in lowercase letters.

2-5. SLIDE: Referencing Shell Variables

Referencing Shell Variables

Syntax:

$NAME

Examples:

```
$ echo $MY_NAME_IS
bullwinkle
$ echo $MESSY
/usr/man/man1/ls.1
$ more $MESSY
&<contents of /usr/man/man1/ls.1>
$ echo $var5
HEY, Rocky!
$
```

H2572 2-5. 12 © 1990 Hewlett-Packard Company

Student Notes

Recall that the shell keeps track of variables as name/value pairs. If you want to use a shell variable, that is, use the value associated with the variable's name, the shell must find the name and return the associated value. This procedure is called **variable substitution.** The shell will perform variable substitutions on any command line that contains a $ symbol followed by a legal variable name.

$legal_variable_name

The shell does the following for every command line:

- It scans the command line looking for $ symbols.
- If it sees a $ followed (with no intervening white space) by a legal variable name, the shell removes the *$variable_name* from the command line, goes to its memory, finds the value associated with that name, and plugs that value back into the command line in place of the removed *$variable_name*.

- After all such substitutions have taken place (and after a few other things have happened, as we will see later), the shell executes the command line that has resulted from all of the substitutions.

The command on the slide:

```
echo $var5
```

will report the value stored in the variable named var5. The shell looks up the value associated with the name var5, puts that value on the echo command line, and executes the resulting command line `echo HEY, Rocky!`. The echo command dutifully outputs it. The echo command provides an easy way to find out the value of a shell variable. Notice that it is the shell that performs the substitutions, not the echo command!

2-6. SLIDE: The Environment

The Environment

Each shell has two data areas:

- The local data area.
- The environment.

We will use four commands to explore these areas:

- set.
- unset.
- env.
- export.

H2572 2-6. 13 © 1990 Hewlett-Packard Company

Student Notes

As we have seen, a variable is a name and a value — a name/value pair. There are two places where variables can be stored: in the environment and in the local data area of the shell.

The **environment** is another area of memory that is used by the shell for storing shell variable name/value pairs. The variables that are defined in the environment are available to **child processes.** Child processes are formed by the shell when it executes a command. We will see processes again later in the course. At this point, it is sufficient to understand that any variable defined in the environment can be used by subsequently executing commands.

In this section, we shall discuss four commands for dealing with the shell variables and the environment: set, unset, env, and export.

2-7. SLIDE: The set and unset Commands

The set and unset Commands

set	Reports the names and values of all shell variables in the local data area and in the environment.
unset *[name]*	Resets the value of *name* to NULL.

H2572 2-7. 14 © 1990 Hewlett-Packard Company

Student Notes

The set command is often used to report the names and values of all variables available to the local shell. This list includes variables in the local data area *as well as* those in the environment. Invoke set with no arguments in order to get the list.

The unset command is used to remove the value of variables. The syntax is as follows:

```
unset [variable]
```

When an argument is given, the unset command will reset the value of the named variable to NULL. The Korn shell (ksh) requires a variable; an error will be produced if none is given.

2-8. SLIDE: The env Command

The env Command

env	Reports the names and values of all shell variables in the environment.

H2572 2-8. 15 © 1990 Hewlett-Packard Company

Student Notes

The env command is generally used to report the values and names of variables in the environment.

2-9. SLIDE: The export Command

The export Command

export *name*	Moves the variable *name* from the local data area to the environment.
export	Reports the names and values of variables exported.

H2572 2-9. 16 © 1990 Hewlett-Packard Company

Student Notes

Within a given process, you can declare, initialize, read, and change variables. But a variable is local to the current process. When a process creates (forks) a child process (for instance, by executing a command), the parent does not pass the local variable on to the child. If we desire to pass the variable to the child process, we must first move the variable from the local data area to the environment.

The most common use of the export command is to place variables (whose names are given as arguments to the command) into the environment. With no arguments, export reports what variables were placed into the environment. In other words, export, with no arguments, reports what variables, if any, were arguments to an export command. It reports both the name and the value of each "exported" variable.

The export command is a shell intrinsic; it is not a section 1 command. It can be found in sh(1) and ksh(1). (See also module 3.)

2-10. LAB: The Shell

Lab Objective

To become familiar with the shell as an interpreter of commands. This is achieved by studying shell variables and simple commands.

Directions

Complete the following exercises and answer the associated questions.

1. Set a shell variable named XXX equal to your first name. How do you see the contents of that variable?

2. Now start a child shell by typing ksh. Look at the contents of XXX now. What happened? Now kill the child shell by pressing [CTRL]-[d] [Return]. Does the parent still know about the variable XXX?

3. What command can be typed in the parent shell to enable the child to see the contents of XXX? How can you see all variables that the child shell will inherit?

4. Start another child shell. Look at the variable XXX. Now set the variable XXX equal to your partner's name. Is XXX now a local or environmental variable? List the environmental variables. What is XXX set to?

5. Now remove the variable XXX from the child shell. Does XXX exist either locally or within your environment? Why or why not?

6. Kill the child shell, returning to your LOGIN shell. Does XXX still exist? Why or why not? What commands did you use to verify this?

Module 3 — Command Execution

Objectives

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

- Describe the difference between a shell intrinsic command and an HP-UX command.
- Explain the search used by the shell to find a command.
- Use the whereis and find commands to locate files in the HP-UX file system.
- Explain how a process is created (fork and exec).
- Use the ps command to monitor running processes.
- Explain how the shell knows who is executing a command.
- Start a process running in the background.
- Start a background process that is immune to the hangup (logoff) signal.
- Stop processes from running by sending them signals.
- Use grave accents (back single quotes) to assign the output of a process to a variable.
- Use the expr command for simple arithmetic and string matching.

3-1. SLIDE: How a Command Is Executed

H2572 3-1. 17 © 1990 Hewlett-Packard Company

Student Notes

For the shell to execute a command, it must know two things:

- Where is the command to be executed?
- Who is executing the command?

We will first look at where the command is, and then we will discuss how the shell knows who is executing the command.

3-2. SLIDE: Shell Intrinsics versus HP-UX Commands

Shell Intrinsics versus HP-UX Commands

Shell intrinsics are built into the shell.

cd
set
echo

HP-UX commands live in /bin or /usr/bin.

ls
more
vi

The system locates HP-UX commands by using the PATH variable.

H2572 3-2. 18 © 1990 Hewlett-Packard Company

Student Notes

Some commands that you type at the keyboard are files in directories such as /bin or /usr/bin. These commands are HP-UX commands. But many commands, such as cd, pwd, and echo, are actually built into the shell itself. These commands do not exist as files in the HP-UX file system, but are like subroutines of the shell program. These commands are intrinsic shell commands.

When the intrinsic shell commands are executed, they do not cause a child process to be spawned.

However, commands that exist in the directories are executed as child processes. Since these commands can exist in several directories, the shell must know where to search for them. The PATH variable in your shell defines the directories to search and the order in which they are searched.

HP-UX commands may have the same name as shell intrinsics. However, to access these commands, the user must use the command's absolute PATH name to inform the shell to use *it* rather than the intrinsic of the same name.

3-3. SLIDE: Looking for Commands – whereis

Looking for Commands — whereis

Syntax:

```
$ whereis [ -B | -M | -S ] command
```

Searches a list of directories for a command.

Examples:

```
$ whereis vi
vi : /usr/bin/vi /usr/man/man1/vi.1
$ where ls
ls : /bin/ls /usr/man/man1/ls.1
$ whereis cd
cd :
$ whereis holdyourhorses
holdyourhorses :
```

H2572 3-3. 19 © 1990 Hewlett-Packard Company

Student Notes

HP-UX stores its commands in four main directories: /bin, /usr/bin, /usr/local/bin, and /usr/contrib/bin. The whereis command searches these as well as other directories to determine where a particular command lives. Many users also have a personal bin directory under their login directory. The whereis will not search this directory. Sometimes you lose track of a command and its manual page. HP-UX, through the whereis command, provides a way to locate commands and their manual pages.

The whereis command accepts a single argument that is the name of a command. It returns the location of the executable code and the manual page for the command.

The whereis command searches the following directories:

/bin	/usr/bin	/usr/contrib/bin
/usr/local/bin	/lib	/usr/lib
/usr/local/lib	/usr/contrib/lib	/etc
/usr/contrib/include	/usr/local/include	/usr/games
/usr/contrib/games	/usr/local/games	/usr/src/*
/usr/man/*	/usr/contrib/man/*	/usr/local/man/*

If you want to change the directories that the whereis command searches, use the flags -B, -M, or -S to limit the search to binary, manual pages, or source code, respectively.

See *hier(5)* in your HP-UX manuals for a quick tour through a representative HP-UX directory hierarchy.

3-4. SLIDE: The find Command

The find Command

Syntax:

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

Examples:

```
$ find / -print
/.
/lost+found
/bin
/bin/adb
```

Break

```
$ find / -name myfile -print
/users/user1/myfile
/users/user2/myfile
.
.
.
```

Break

```
$
```

H2572 3-4. 20 © 1990 Hewlett-Packard Company

Student Notes

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

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

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

3-5. SLIDE: Who Am I? – The id Command

Student Notes

Now that we know how the shell finds commands, we must learn how the shell knows who is executing it. This is logical because the permissions do not make sense unless the shell knows to whom they apply.

To find out who the system thinks you are, type the id command.

The id command is a shell intrinsic that displays your user id number (and name) and your group id number (and name). These names may be different from the name you used to log in. The logname or who am i commands will always display the name you logged in under, while the id and whoami commands will display who the shell thinks you are *now*.

3-6. SLIDE: The su Command

The su Command

Syntax:

su *[user_name]* Changes the user-id.

Example:

```
$id
uid=101 (user1), gid=20 (users)
$ su boris
Password:
$ id
uid=303 (boris), gid=21 (badguys)
$ exit
```

user_name defaults to the root.

```
$su
Password:
# id
uid=0 (root), gid=3 (sys)
```

H2572 3-6. 22 © 1990 Hewlett-Packard Company

Student Notes

The su command will allow you to change your user id number. In fact, "su" stands for "switch users" or "set user id." With no arguments, su will attempt to let you change to the user root. For this reason, many people think that the command su means "superuser." The su command can be invoked with an argument specifying the user name of the user that you wish to become.

When invoked, the su command will prompt you for the password of the user you wish to become. If you enter the correct password, su will spawn a subshell and set your user id and group id number to that of the specified user.

Note that if you become superuser, you will get a different prompt.

Caution

To get back to the user you were, *do not* use the su command again. Instead, use the exit command to exit the subshell spawned for you by su.

3-7. SLIDE: The newgrp Command

The newgrp Command

Syntax:

`newgrp [group_name]` Changes the group id.

Example:

```
$ id
uid=220 (bwinkle), gid=50 (meeses)
$ newgrp squirrels
$ id
uid=220 (bwinkle), gid=20 (squirrels)
$ newgrp meeses
$ id
uid=220 (bwinkle), gid=50 (meeses)
$ newgrp badguys
Sorry
$ newgrp
```

H2572 3-7. 23 © 1990 Hewlett-Packard Company

Student Notes

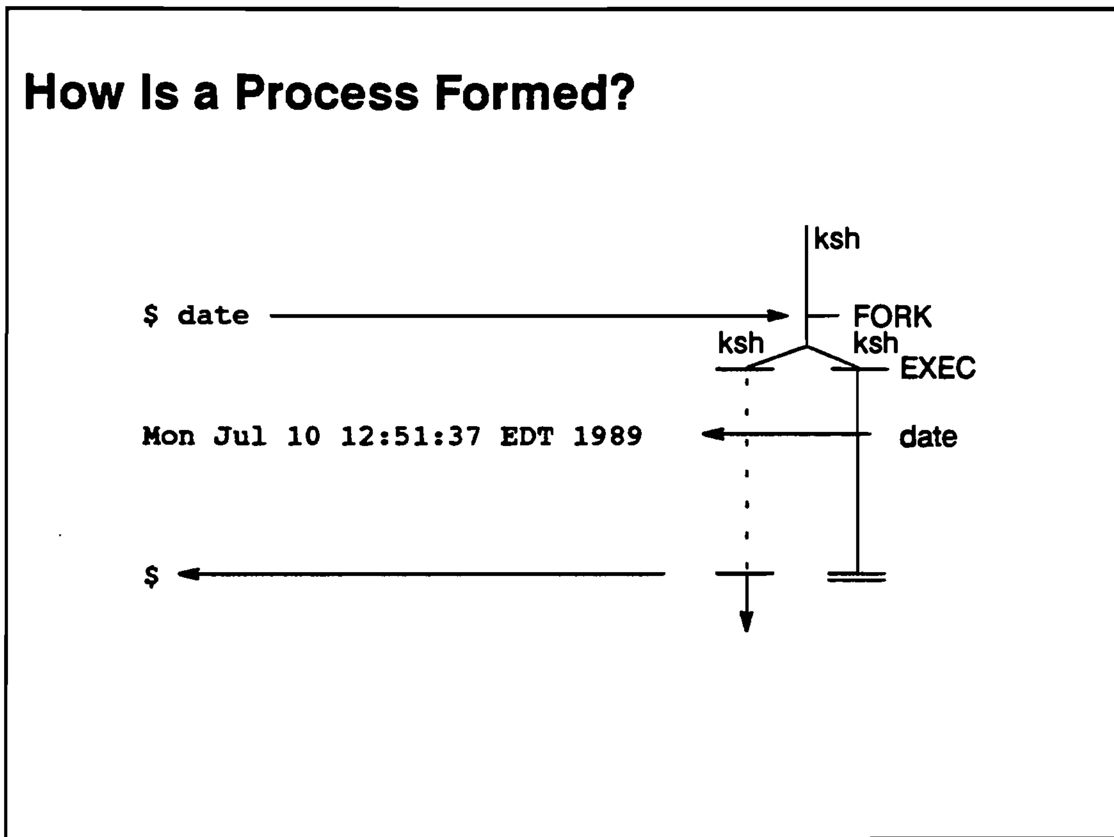
The newgrp command is similar to the su command. The newgrp command will allow a user to change only his group id number and does not spawn a subshell. The syntax is as follows:

`newgrp [groupname]`

You can only change to those groups allowed by the system administrator. If you are not allowed to become a member of the specified group, you will get the message: `Sorry`.

Unlike the su command, however, the newgrp command is used to get back to your original group. There is no child shell spawned by newgrp from which to exit. The newgrp command, with no arguments, will return you to your original login group.

3-8. SLIDE: How Is a Process Formed?



Student Notes

Processes under HP-UX are born (spawned), live, and die. When a shell is running and a user requests that a command be executed, a **child** process is formed. The child process runs until it is completed. The **parent** process (our shell) waits for the child process to terminate and then continues on its merry way. Process creation is accomplished through the operating system with two system calls called fork and exec.

When a child process is formed, first the parent process clones itself. This is called a **fork.** The parent process's executable code, local data area, and environment are cloned. This cloned copy is called a child process. Meanwhile, the parent process goes to sleep and waits for the system to wake it up and tell it that the child process has terminated.

Next, the child process's executable code and local data area are overlaid with the executable code and the local data area for the requested command. This overlaying is called an **exec.** Note that the parent process's environment becomes the child process's environment. Thus, any variables that have been exported are available to the child process.

The child process runs. When it terminates, a signal is sent to the parent process — waking it up and telling it that the child has died.

Just for fun, a child process that has died but whose parent has yet to be told that it is dead is called a **zombie** process. Also, a child process whose parent has died is called an **orphan** process.

3-9. SLIDE: The ps Command

The ps Command

Syntax:

```
ps [-efl]      Reports process status.
```

Examples:

```
$ ps
PID  TTY   TIME COMMAND
1999 tty0p3    0:02 ksh
2001 tty0p3    0:00 ps
$ ps -ef
  UID   PID PPID  C    STIME TTY      TIME COMMAND
  root   35    1  0  Jul 22  ?        0:17 /etc/syncer 120
  root  116    1  0  Jul 22  console  0:00 /etc/getty console
console
  root    2    0  0  Jul 22  ?        0:00 pagedaemon
  root    1    0  0  Jul 22  ?        1:04 init
  root    0    0  0  Jul 22  ?        0:08 swapper
 rocky 1315    1  0 10:30:24 tty0p4   0:07 -ksh [ksh]
 boris 6587    1  0 14:35:04 tty0p3   0:04 -ksh [ksh]
  root 7621    1  0 15:38:36 tty0p2   0:00 /etc/getty -h tty0p2
9600
```

H2572 3-9. 25 © 1990 Hewlett-Packard Company

Student Notes

The ps command gives information about processes. The ps command takes only options; it will not take arguments. If options are not specified, the ps command gives a short report about processes associated with your terminal (that is, your processes).

The ps command is most commonly called with no options or with -ef and sometimes with -efl. The -e option reports information about all processes on the system, not just your own. The -f and -l options report full and long listings which contain a lot of detail about the processes reported.

The ps command has many handy uses. In this section we will use it to illustrate the effects of command execution.

In this slide we show two invocations of ps. The first just reports information about processes associated with our terminal. As we would expect, the processes associated with our terminal consist of a shell (our login shell) and the ps command that is currently running.

The second example shows a portion of the output of a ps giving a full (-f option) listing of *every* (-e option) process on the system. If response time was terrible on a system, we might do a ps -ef to see if there were many processes running. Unfortunately, ps uses a large amount of CPU capacity and will slow the system even more; therefore, its use to investigate slow response time is generally not a good idea.

UID	The real user ID number of the process owner.
PID	The process ID of the process.
PPID	The process ID of the parent process.
C	Processor utilization for scheduling.
STIME	Starting time of the process.
TTY	The controlling terminal for the process.
TIME	The cumulative execution time for the process (min:sec).
COMMAND	The command name.

For more information, see ps(1).

3-10. SLIDE: Background Processing

Student Notes

Some commands take a long time to execute. For example, running the text processors on this course takes a very long time. You probably would not want to wait for them to finish before you went on to other work. It would be wasteful if we had to tie up a terminal just waiting for a slow command to finish.

Fortunately, HP-UX provides a way to execute a command in the background. The shell returns with a prompt immediately and you may do work on your terminal even as the background command is running. The parent shell simply does not wait for the child process to complete.

To put a command in the background, we terminate the command line with an ampersand (&). The ampersand instructs the shell to run the command in the background, leaving the terminal free for other use.

When a command is put into the background, the shell reports the process id number of the background command before you get your shell prompt back.

A command run in the background cannot be killed with the DEL or Break key. However, ending your session by logging off *will* kill it. Later in this unit, we will see how to kill a background command without logging off. Also, we will see how to make the background command immune to the log out signal.

It is important to note that a background process should have *all* of its input and output explicitly redirected. Also, redirecting standard error can be appropriate to some tasks.

3-11. SLIDE: An Example – &

An Example — &

Example:

```
$ sort * > output.sort &
  [1] 11279

$ ps

PID   TTY   TIME      COMMAND
11289  tty01    0:00        ps
11279  tty01    0:01        sort
11259  tty01    0:09        ksh
```

H2572 3-11. 27 © 1990 Hewlett-Packard Company

Student Notes

The slide shows an example of a command that may take a while to run. The user places the command in the background by appending an ampersand (&) after the command.

3-12. SLIDE: Command Substitution

Command Substitution

Use backquotes (`) to execute the enclosed command.

Example:

```
$ CURDIR=`pwd`
$ echo $CURDIR
/users/bwinkle

$ ls
file1  file2  file3
$ ls | cat
file1
file2
file3
$ X=`ls`
$ echo $X
file1 file2 file3
```

H2572 3-12 28 © 1990 Hewlett-Packard Company

Student Notes

Command substitution is another handy feature of the shell. It allows you to capture the output of a command and assign it to a variable, or use that output as an argument to another command. Because most HP-UX commands generate standard output, command substitution can be very useful.

The slide shows a simple example. Normally, the pwd command writes its output to the terminal. By enclosing the command in backquotes (`), which are sometimes referred to as grave accents, we capture the pwd output and assign it to the shell variable curdir. The $ before curdir on the first line is a shell prompt.

Command substitution is useful both in shell programming and in interactive use at the terminal.

A note on efficiency: All things being equal, assigning the output of a command to a variable will be faster than redirecting that output to a file.

3-13. SLIDE: Some Related Commands

H2572 3-13. 29 © 1990 Hewlett-Packard Company

Student Notes

Here we present three commands because of their relationship to and usefulness with background processing: nice, nohup, and kill.

3-14. SLIDE: The nice Command

The nice Command

Syntax:

`nice [-N] command_line` Runs a process at a lower priority. *N* is a number between 1 and 19.

Example:

```
$ nice -10 cc myprog.c -o myprog
$ nice -5 find . -name '*myfile*' -print \
> > /tmp/myoutputfile &
$
```

H2572 3-14. 30 © 1990 Hewlett-Packard Company

Student Notes

The nice command is a way to start a process at a lower than normal priority. The nice command is one of a group of commands in HP-UX called **prefix commands.** The syntax is as follows:

`nice [-`*increment*`]` *command line*

where *increment* is an integer value between 1 and 19. If an increment of 19 is chosen, the process is as "nice" as it can be, allowing other processes to use the system resources before it does. However, the process's priority will slowly increase over time to ensure that the process will eventually run.

3-15. SLIDE: The nohup Command

The nohup Command

Syntax:

nohup *command line* & — Makes a command immune to hangup (logout) no hang up.

Example:

```
$ nohup cat /usr/man/man1/* > bigfile &
  [1] 11290
$ CTRL - d  Return

login: bwinkle
Password:
    .
    .
    .
$ ps -f

 PID PPID          Command
11301     1    . . .     -ksh
11290     1    . . .     cat  . . .
11400    11301    . . .     ps -f
```

H2572 3-15. 31 © 1990 Hewlett-Packard Company

Student Notes

HP-UX provides a mechanism for making commands immune to hangup and logging off: the nohup command. This is most often used in conjunction with background commands. The effect is that you can run long jobs without tying up the terminal, the phone line, or yourself. You can log out and log in later to get the results of nohup background commands.

The nohup command is a prefix command; it is placed on the command line in front of the command that you want to execute. The nohup command will search $PATH for commands that follow it.

When using nohup, output is usually redirected to a file. If the user does not specify an output file, then nohup will create the file nohup.out in the current directory to receive any output. Note that nohup.out will accumulate both stdout and stderr.

See nohup(1) in the *HP-UX Reference Manual* for additional information.

3-16. SLIDE: The kill Command

The kill Command

Syntax:

```
kill [-sig_no] PID [PID ... ]        Sends a signal to specified processes.
```

Example:

```
$ cat /usr/man/man1/* > bigfile &
  [1] 11410
$ kill 11410
[1]+Terminated  . . .
$ kill -9 0
```

H2572 3-16. 32 © 1990 Hewlett-Packard Company

Student Notes

The kill command can be used to kill any command including nohup and background commands. More specifically, kill sends a signal to processes. The default action for a process is to die when certain signals are received. The issuer must be the owner of the target commands; kill cannot be used to kill someone else's commands unless the kill is sent by the superuser.

By default, the kill command sends signal number 15 to the specified process. If the process specified is 0, then kill "kills" all processes associated with the current shell, *including* the current shell.

In the HP-UX world, it is not possible to actually kill a process. The most HP-UX will do is request that a process terminate itself. It is possible to make commands immune to the termination signal. The closest thing to a sure kill that HP-UX will give can be had by using 9 as the signal number.

For a list of signals and their actions, see signal(2) in the *HP-UX Reference Manual*.

3-17. LAB: Command Execution

Lab Objective

To show how the shell executes a process and to describe the user commands for following the process's execution.

Directions

Complete the following exercises and answer the associated questions.

1. When you type the date command, the shell executes it. Where is the date command?

2. Where is the cat command located?

3. Where is the cd command located? What happened? Why?

4. Where is the banana command located? What happened? Why?

5. Use the find command to locate the file called file3. Begin the search at your HOME or LOGIN directory. What command did you use and what is the path name to the file?

6. Work together with your partner on this part of the exercise. Use the su command to switch over to your partner's login id. Now use the id command to see the change. What commands did you use and how do you return to your own login account?

7. What processes, currently running, do you own? What command did you use?

8. What processes are currently running on the system? What command did you use?

9. How would you change your group id? That is, how can you become a member of another group?

10. The id command is an HP-UX command that displays your user id number and group id number (uid and gid). Type this command. What does it tell you? How can you change this information?

11. Type the commands:

    `cd /usr/man/cat1.Z` or `cd /usr/man/man1.Z` (whichever exists on your system)
    `nohup cat * > $HOME/bigfile &`

Then execute a ps -f command. Take note of the PID and PPID of the cat processes. Now log out, log in again, and execute the ps -f command. Where is the cat process? Remove the file called bigfile before the next exercise.

12. The nohup command makes a process immune to death upon the death of its parent process. Try the commands:

    `cd /usr/man/cat1.Z` or `cd /usr/man/man1.Z` (whichever exists on your system)
    `nohup cat * > $HOME/bigfile &`

Now log out and log in again. Execute a ps -f command. Who is the parent of the cat command now?

13. Set a variable called dir equal to the output of the pwd command. What is one way of finding the number of characters in the string contained in that variable?

14. Change into the directory $HOME/tree/subdir3/file4/d2/f3. Set a variable called pdir equal to the output of the pwd command. Now return to your HOME directory. What commands did you use, and how can you return to the f3 directory using the pdir variable?

15. Set the variable whoson equal to a sorted list of the user_ids presently logged into the system. What commands did you use?

16. Execute the following two commands. How are they different?

```
$ date;pwd;banner hello &

$ (date;pwd;banner hello) &
```

Module 4 — File Name Generation

Objectives

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

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

4-1. SLIDE: Introduction – Special Characters

Introduction — Special Characters

- File name generating characters are interpreted by the shell.
- File name generating characters are *not* wildcards!

H2572 4-1. 33 © 1990 Hewlett-Packard Company

Student Notes

File name generating characters are *not* wildcard characters. A **wildcard character** is expanded by a command to match certain file names. In contrast, a **file name generating character** is expanded *by the shell* before the command ever executes. Therefore, the command sees an expanded list just as if you had typed the entire list at the keyboard.

File name generating characters are certainly a typing aid; but they can be more than just that. For instance, if your files and directories are organized according to consistent naming conventions, *file name generatings* can aid in directory searches, file copies, and other common user actions. You can get a list of common HP-UX file name suffix conventions by looking up suffix(5) in your HP-UX manual set.

4-2. SLIDE: File Name Generating Characters

File Name Generating Characters

?	Matches any single character except a leading dot.
[ ]	Defines a class of characters. - is used to define an inclusive range. ! is used to negate the defined class.
*	Matches zero or more characters except a leading dot.

H2572 4-2. 34 © 1990 Hewlett-Packard Company

Student Notes

Special characters are used to match file names. File name generation is a feature of the shell.

The special characters that we will discuss are the following:

?	Matches any single character except a leading dot.
[ ]	Defines a class of characters from which one will be matched (unless it is a leading dot). Within this class, a hyphen (-) can be used between two ASCII characters to mean *all* characters in that range inclusive, and an exclamation point (!) can be used as the first character to negate the defined class.
*	Matches zero or more characters (except a leading dot).

We will see each of these characters in detail.

4-3. SLIDE: File Name Generation and Dot Files

File Name Generation and Dot Files

- File name generating characters will *never* generate a leading dot.
- The leading dot in a dot file must be matched explicitly.

H2572 4-3. 35 © 1990 Hewlett-Packard Company

Student Notes

As with the ls command, dot files are treated differently from ordinary files. The file name generating characters do not match file names that begin with a dot. In this unit, as we discuss the file name generating characters, we will demonstrate how dot files are handled.

4-4. SLIDE: File Name Generation — ?

File Name Generation — ?

The question mark character (?) matches any single character.

`$ echo ???`	Matches all three-character file names.
`$ echo abc?`	Matches four-character file names starting with "abc."
`$ echo ??a??`	Matches five-character file names with an "a" as the third character.

H2572 4-4. 36 © 1990 Hewlett-Packard Company

Student Notes

A question mark matches any single character, but it will not match a leading dot.

File name generation is accomplished by the shell before commands are invoked. Thus, in the example, the shell looks for file names that match the patterns specified. All resulting file names are passed as arguments to the echo command. If there is no match, then the pattern itself is passed as the argument.

4-5. SLIDE: File Name Generation — [ ]

File Name Generation — [ ]

Open and close brackets ([ ]) define a class of characters from which one will be matched.

`$ echo [abc]??`	Matches all three-character file names starting with a, b, or c.
`$ echo ?[a-zA-Z]`	Matches all two-character file names ending in a letter.
`$ echo [!A-Z]???`	Matches all four-character file names that *do not* start with a capital letter.

H2572 4-5. 37 © 1990 Hewlett-Packard Company

Student Notes

Brackets are used to delimit a **character class.** A character class matches any single character from the specified enclosed list.

An exclamation point (!) as the first item inside the bracket stands for the negation of the character class; that is, the class stands for the class of all characters *not* listed inside the brackets.

If a hyphen (-) is placed between two characters within brackets, the character class will be all characters in the ASCII sequence — see ascii(7) — from the first character to the last one inclusive. Thus, the classes:

```
[!123456789]
```

and

```
[!1-9]
```

both stand for any character except the digits 1 through 9.

A leading dot (.) cannot be matched with a character class.

4-6. SLIDE: File Name Generation — *

File Name Generation — *

The asterisk character (*) matches zero or more characters except a leading dot (.).

`$ echo *`	Matches all file names except dot files.
`$ echo .*`	Matches all dot files.
`$ echo *a*`	Matches all file names with an "a" in them except dot files.
`$ echo *e`	Matches all file names ending in "e" except dot files.

H2572 4-6. 38 © 1990 Hewlett-Packard Company

Student Notes

An asterisk (*) matches any string of zero or more characters.

As usual in file name generation, an asterisk (*) will not match a leading dot.

4-7. SLIDE: Examples – File Name Generation

Examples — File Name Generation

Given the directory list, what do the following commands print out?

```
$ ls -a
.               Abc        e35f
..              Abcd       efg
.profile        abc        fe3f
.sh_history     abcdelf    fe3fg
$ echo .[!.]*
$ echo *e[0-9]f
$ echo ?*
$ echo .?*
$echo *????
$echo ????*
```

H2572 4-7. 39 © 1990 Hewlett-Packard Company

Student Notes

The slide shows several examples of file name generation. Given the directory list, tell what the echo commands will reveal.

4-8. SLIDE: File Name Generation and Quoting

File Name Generation and Quoting

Quoting is a mechanism to escape (take away) the special meaning of special characters so that the characters can stand for themselves.

H2572 4-8. 40 © 1990 Hewlett-Packard Company

Student Notes

There are many characters in HP-UX that have special meanings. For example, we have seen that the $ character can be used either literally or to substitute shell variables. Since context is not always sufficient to determine the meaning of a character, it is necessary to have a mechanism that escapes the special meaning of a character and forces it to be treated as just a character. This mechanism is called **quoting.**

4-9. SLIDE: Special Characters We Have Seen

Special Characters We Have Seen

$

? [ ] *

< > > > 2> 2> >

white space, Return, tab

|

H2572 4-9. 41 © 1990 Hewlett-Packard Company

Student Notes

We have seen all of these as special characters:

$	Used for variable substitution.
? [ ] *	Used for file name generation.
< > 2> > > 2> >	Used for input and output redirection.
White space	Used for command line argument delimiter.
Return	Used to delimit a line of input.
Tab	Used for command line argument delimiter.
|	Used for pipelines.

We have seen more than these, but these are a sampling.

4-10. SLIDE: There Are Three Ways to Escape

There Are Three Ways to Escape

Backslash	\
Single quotes	' ′
Double quotes	"

H2572 4-10. 42 © 1990 Hewlett-Packard Company

Student Notes

There are only three ways to turn off the meaning of special characters:

- Backslash (\).
- Single quotes (').
- Double quotes (").

Note that single quotes (') are not the same as the grave accent (‘).

The backslash turns off the special meaning of a special character. Any single character that follows it has its special meaning escaped.

Single quotes will also turn off the special meaning of special characters. Any special character inside the single quotes is escaped. The exception is the single quote itself. A backslash *cannot* be used inside single quotes to escape a single quote because the single quotes escape the backslash.

Double quotes are the trickiest. Inside double quotes, most special characters are escaped. The exceptions are the $ symbol (when used for variable substitution), the double quote ("), the backslash (\), and the grave accent (`). You may use the backslash inside double quotes to escape the meaning of $ or ".

The next few slides give examples of quoting.

4-11. SLIDE: Quoting – \

Quoting — \

\ Takes away the special meaning of the character immediately following.

Examples:

```
$ echo \$ABC
$ABC

$ echo "hello" \> file
hello > file
$ name=bullwinkle
$ echo $name \$name
bullwinkle $name

$ echo one two \
> three four
one two three four
```

H2572 4-11. 43 © 1990 Hewlett-Packard Company

Student Notes

The backslash turns off the special meaning of a special character. Any single character that follows it has its special meaning escaped.

4-12. SLIDE: Quoting — '

Quoting — '

' Takes away the special meaning of all characters inside single quotes except a single quote.

Examples:

```
$ echo '$ABC > hi there!'
$ABC > hi there!
$ echo 'this doesn't work'
>[DEL]
$ name=rocky
$ echo $name \$name '$name'
rocky $name $name
```

H2572 4-12. 44 © 1990 Hewlett-Packard Company

Student Notes

Single quotes will also turn off the special meaning of special characters. Any special character inside the single quotes is escaped. The exception is the single quote itself. A backslash *cannot* be used inside single quotes to escape a single quote because the single quotes escape the backslash. There is no variable substitution or file name generation done inside single quotes.

4-13. SLIDE: Quoting — "

Quoting — "

" Takes away the special meaning of all characters inside double quotes except \, ", $, and ' (grave accent).

Examples:

```
$ ABC=123
$ echo "$ABC"
123
$ echo "\$ABC"
$ABC
$ echo " `pwd` "
/users/bwinkle/solutions
$ name=boris
$ echo $name \$name '$name' "$name"
boris $name $name boris
```

H2572 4-13. 45 © 1990 Hewlett-Packard Company

Student Notes

Double quotes are the trickiest. Inside double quotes, most special characters are escaped. The exceptions are the $ symbol (when used for variable substitution), the double quote ("), the backslash (\), and the grave accent (`). You may use the backslash inside double quotes to escape the meaning of $ or ".

4-14. SLIDE: A Summary

A Summary

Mechanism	Escapes	Exceptions
Backslash	Next character	None
Single quotes	All characters inside ' '	Single quote
Double quotes	All characters inside " "	Back slash Dollar sign Double quote Grave accent

H2572 4-14. 46 © 1990 Hewlett-Packard Company

Student Notes

The slide shows a summary of the quoting characters and their actions.

4-15. SLIDE: File Name Generation and Quoting Review

File Name Generation and Quoting Review

What is the result of:

```
$ file=tmp
$ file_ext=.c
$ echo "?"???
$ echo "["[0-9]*"]"
$ echo "*"
$ echo \???$file_ext
$ echo $file_ext$file
$ grep * *
$ grep '*' *\.c
```

H2572 4-15. 47 © 1990 Hewlett-Packard Company

Student Notes

Because the shell performs the file name generation before the command is executed, any special characters that you wish to be passed to the command as arguments must be quoted to prevent their interpretation by the shell.

The slide shows some examples.

4-16. LAB: File Name Generation

Lab Objective

To practice using metacharacters and quoting on the command line to save typing.

Directions

Complete the following exercises and answer the associated questions.

1. Create a subdirectory called match (mkdir match) and cd to that directory. Using the touch command (syntax: `touch` *filename*), create files so that the following will be true:

   The pattern ?.XX will match exactly *two* file names.
   The pattern *XX will match exactly *three* file names.
   The pattern XX.?? will match exactly *one* file name.
   The pattern XX.* will match exactly *two* file names.

Use the echo command to check your results.

2. Use a single rm command to remove all of the files created in the previous exercise. Hint: First practice using the ls command to check your answer.

3. Change to your HOME directory, and then type the command `ls *` and explain the output.

4. If the command `echo ???XX` produces the output `???XX`, what does it mean?

5. From your HOME directory, what would be the output from the following commands:

   a. `$ echo *.c`

   b. `$ echo m[n-s]*.?`

   c. `$ echo ??????`

   d. `$ echo *[0-9]*`

   e. `$ echo *file`

6. Type an echo command that will produce the following output:

```
$1 million dollars      . . .      and that's a bargain !
```

7. Use a quoting mechanism to search for the pattern * in the file /etc/passwd.

8. Why does echo `'$abc'` produce `$abc`, while echo `"$abc"` produces a blank line (or in ksh, an error)?

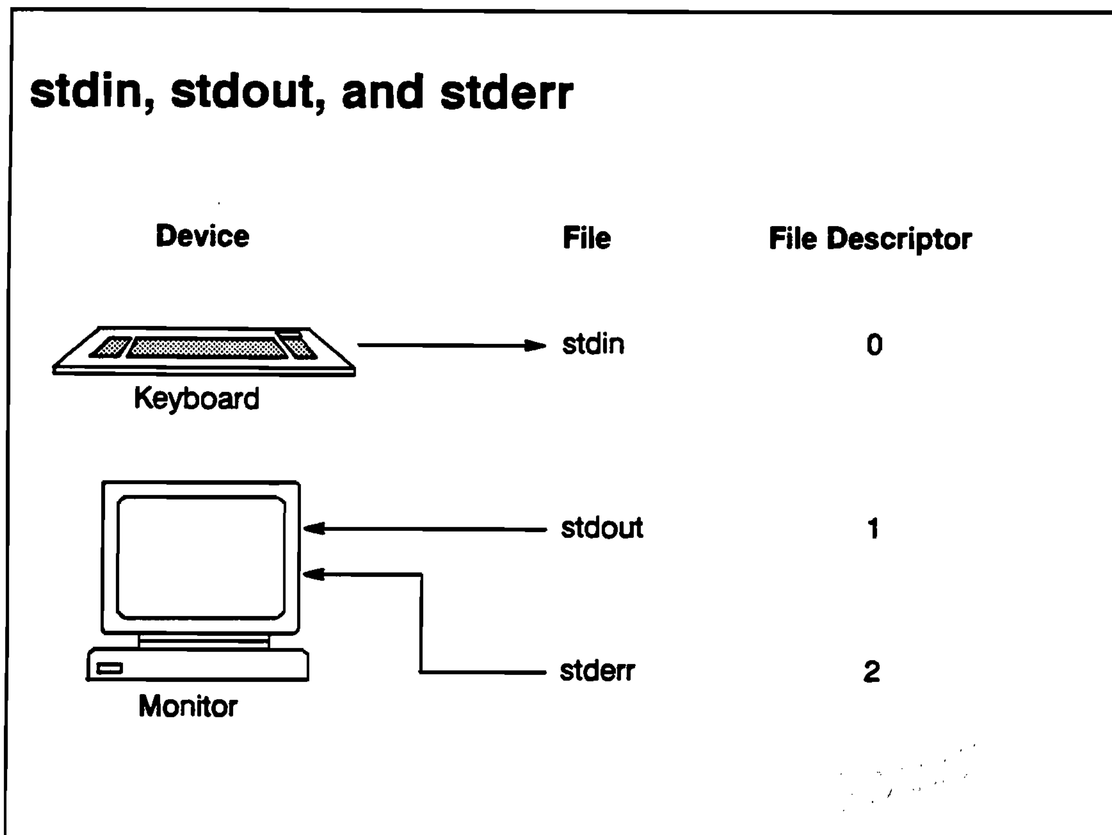
Module 5 — Input and Output Redirection

Objectives

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

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

5-1. SLIDE: stdin, stdout, and stderr



Student Notes

Every time a shell is started, three files are automatically opened for your use. These files are called stdin, stdout, and stderr.

The stdin file is the file from which your shell reads its input. The stdin file is usually called **standard input.** This file is opened with the C language file descriptor, 0, and is usually attached to your keyboard. Therefore, when the shell needs input, it must be typed in at the keyboard.

The stdout file is the file to which your shell writes its normal output. The stdout file is usually called **standard output.** This file is opened with the C language file descriptor, 1, and is usually attached to the monitor part of your terminal. Therefore, when the shell produces output, it is generated at your screen.

The stderr file is the file to which your shell writes its error messages. The stderr file is usually called **standard error.** This file is opened with the C language file descriptor, 2. Like the stdout file, the stderr file is usually attached to the monitor part of your terminal. The stderr file can be redirected independently of the stdout file.

Module 5 — Input and Output Redirection

The purpose of this module is to show you how to change the default assignments of stdin, stdout, and stderr, thus taking the input from a file other than the keyboard and producing output (and error messages) somewhere other than the monitor. Note that the C-shell does not handle stderr.

5-2. SLIDE: Input Redirection — <

Input Redirection — <

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

Example:

```
$ mail bwinkle < another_file
$
```

H2572 5-2. 49 © 1990 Hewlett-Packard Company

Student Notes

For commands that take their input from standard input, we can redirect the input so that it comes from a file instead of from the keyboard. The mail command is often used with input redirection. We can use the editor to create a file containing some text that we want to mail, and then we can redirect the input of mail so that it uses the text in the file. This is useful if you want to save the mail message for future reference.

Input redirection causes no change to the input file.

5-3. SLIDE: Output Redirection — > and > >

Output Redirection — > and > >

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

Examples:

Overwrite:

```
$ ls > another_file

$ date > who.log
```

Append:

```
$ date >> another_file

$ who >> who.log
$ ls >> who.log
```

H2572 5-3. 50 © 1990 Hewlett-Packard Company

Student Notes

If a command line contains the output redirection symbol (>) followed by a file name, the standard output from the command will go to the specified file instead of to the terminal. If the file did not exist before the command was invoked, then the file is automatically created. If the file *did* exist before the command was invoked, then the file will be *overwritten;* the command's output will completely replace the previous contents of the file.

If we want to append to a file instead of overwriting, we can use the output redirection append symbol (> >). This will also create the file if it did not already exist. There must be *no* white space between the two > characters.

The slide shows both kinds of output redirection.

5-4. SLIDE: Error Redirection – 2> and 2> >

Error Redirection — 2> and 2> >

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

Examples:

Overwrite:

$ cp 2> *another_file*

Append:

$ cp 2>> *another_file*

H2572 5-4. 51 © 1990 Hewlett-Packard Company

Student Notes

Most HP-UX commands produce diagnostics if something goes wrong. Diagnostic output, also known as stderr, is usually sent to the terminal. It can be redirected separately from stdout. This separate redirection is handy to avoid getting diagnostic messages in a file that is supposed to contain only redirected stdout. To redirect stderr, we use 2>. There must be *no* white space between the 2 and the > character. Similar to output redirection, this will create a file if necessary, or overwrite the file if it exists. We can append to a file using the 2> > symbol.

5-5. SLIDE: What Is a Filter?

What Is a Filter?

Any command that reads input from stdin and produces output to stdout is a filter.

Examples:

- cat
- more
- sort
- grep
- wc

H2572 5-5. 52 © 1990 Hewlett-Packard Company

Student Notes

Now that we know how to redirect the input to and output from a process, it is interesting to note that some processes use stdin, stdout, and stderr all at once!

A process that reads its input from stdin and writes its output to stdout is called a **filter.**

This section is provided to introduce you to three useful filters: sort, grep, and wc.

5-6. SLIDE: The cat Command, Revisited

The cat Command, Revisited

Concatenates and displays files to stdout.

Recall:

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

Examples:

```
$ cat file1 file2 > file3

$ cat > file4
This is input
CTRL - d

$ cat
This is input
This is too
CTRL - d
This is input
This is too
$
```

H2572 5-6. 53 © 1990 Hewlett-Packard Company

Student Notes

Now we can see how the cat command got its name. Recall that the cat command was designed to concatenate and display files. We saw how cat can be used to display files. Now witness how cat can be used to concatenate the files. Using input and output redirection, we can take the output of a cat command (usually with more than one file name on the command line) and put it into another file. This other file now contains the concatenated contents of all files listed on the command line.

The slide shows some additional uses of the cat command.

In the last two examples, the cat command gets its input from stdin and puts its output to stdout.

5-7. SLIDE: The sort Command

The sort Command

Sorts lines of fields.

Syntax:

```
sort [-ndtX] [+field_no] [file]
```

Examples:

```
$ sort new.jersey

$ sort < new.jersey

$ sort -nt: +2 /etc/passwd

$ sort -d new.jersey
```

H2572 5-7. 54 © 1990 Hewlett-Packard Company

Student Notes

The sort command is powerful and flexible. The lines of a file or files can be sorted on particular fields. Sorting can be performed numerically (the -n option) or lexicographically (no option) or in dictionary order (the -d option). The fields are delimited by tabs or by the character specified as *X* when using the -t*X* option. Like most filters, sort will take files as arguments or will read from standard input.

You can also tell sort to sort at a particular field by using a + symbol followed by the field number. The sort command assumes that the field numbering starts with zero.

Simple sorts are very easy, but more complex sorts may take a few tries to get the command line syntax right. Do not get frustrated. The power and ease of this program make the learning headaches worthwhile.

See sort(1) in the *HP-UX Reference Manual* for a full discussion of sort capabilities.

5-8. SLIDE: The grep Command

The grep Command

Syntax:

`grep [-inv] pattern [file]` — Searches files for a pattern and outputs any line containing the pattern.

Examples:

```
$ grep user /etc/passwd
$ grep -i user /etc/passwd
$ grep -v user /etc/passwd
$ grep -n like < new.jersey
$ grep -niv like new.jersey
```

H2572 5-8. 55 © 1990 Hewlett-Packard Company

Student Notes

The grep command is very useful. It takes a (usually quoted) pattern as its first argument, and takes any number of file names as its remaining arguments. It searches the named files for lines that contain the named pattern. The grep command then displays the lines that contain the pattern.

There are three very popular options to grep: -n, -v, and -i.

- The -i option tells grep to ignore the case of the letters in the pattern.
- The -v option displays the lines that *do not* contain the pattern.
- The -n option prepends line numbers to each line displayed.

As with all filters, grep reads from standard input if no file is specified.

5-9. SLIDE: The wc Command

The wc Command

Syntax:

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

Examples:

```
$ wc -l new.jersey

$ wc -w < new.jersey

$ wc -c < new.jersey

$ wc new.jersey

$ wc < new.jersey

$ wc -clw < new.jersey
```

H2572 5-9. 56 © 1990 Hewlett-Packard Company

Student Notes

The wc command is a counter. It counts the number of lines, words, and characters in a file. The command has options -l, -w, and -c. The -l option will display the number of lines, the -w option will display the number of words, and the -c option will display the number of characters.

The order of the options determines the order of the output.

5-10. LAB: Input and Output Redirection

Lab Objective

To practice input and output redirection and filters.

Directions

Complete the following exercises and answer the associated questions.

1. Redirect the output of the date command to a file called date.out in your HOME directory.

2. Append the output of the ls command to the file date.out. Look at the contents of date.out. What do you notice?

3. Using input redirection, mail the file date.out to your mail partner.

4. Type the cp command with no arguments. What happens? Now try redirecting the output from this command to the file cp.file. What happens? What must you do to redirect that error message to a file called error.file?

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

6. Redirect the output of the following command

```
grep "user[0-9][0-9]*" < /etc/passwd
```

to a file called grepped. Now count the number of lines in that file. How many are there?

7. Using the above techniques, how many users are logged in on the system?

8. How many login accounts are set up on the system? What command did you use to find out?

9. Sort your names file and save the output in a file called names.sort. Sort the names file in reverse order and save the output back into the names file. What commands did you use?

10. (Advanced) How many ordinary files do you have in your HOME directory? What commands did you use?

Module 6 — Pipelines

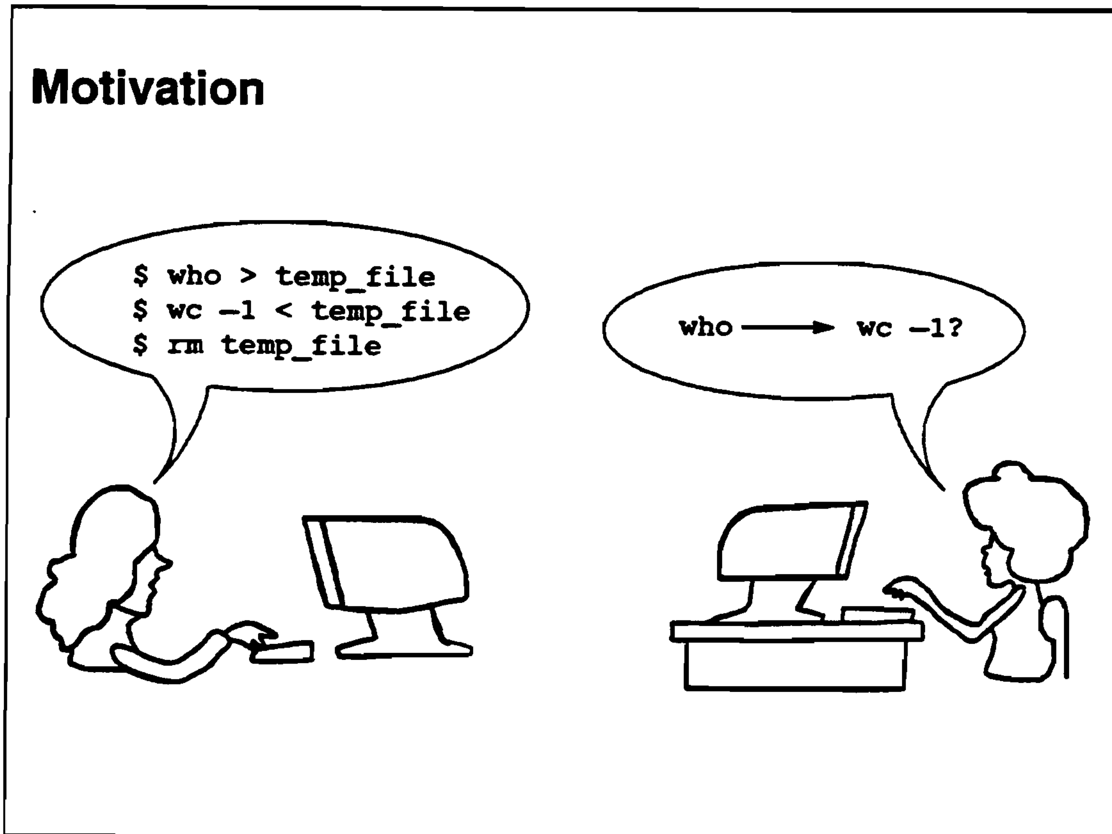
Objectives

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

- Construct a pipeline to take the output from one command and make it the input for another.
- Use the tee, cut, and pr filters.

Module 6 — Pipelines

6-1. SLIDE: Motivation



H2572 6-1. 57 © 1990 Hewlett-Packard Company

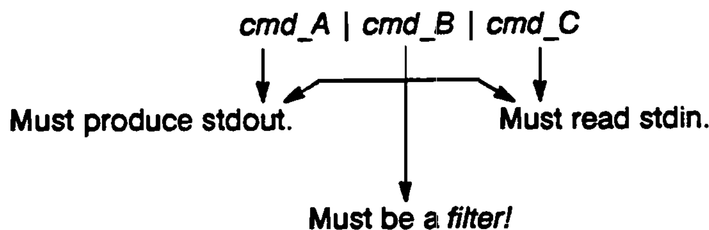
Student Notes

Very often we would like to take the output from one command and make it the input to another. Using I/O redirection, this can be accomplished rather simply. We redirect the output from the first command to a temporary file; then we redirect the input to the next command from the temporary file. Finally, we should remove the temporary file.

This three-step process is not too cumbersome, but it becomes much more so when the process is repeated with a third command. What we would like is a more sensible way to do this.

6-2. SLIDE: The | Symbol

The | Symbol



Example:

```
$ who | wc -l

$ who | sort | wc -l

$ who | grep tty | wc -l
```

H2572 6-2 58 © 1990 Hewlett-Packard Company

Student Notes

The | symbol (read as the **pipe** symbol) is used for linking two commands together. The standard output (stdout) of the command to the left of the | symbol will be used as the standard input (stdin) for the command to the right.

Therefore:

- Any command to the left of a | symbol must produce output to stdout.
- Any command to the right of a | symbol must read its input from stdin.
- Any command between two | symbols must be a filter.

6-3. SLIDE: Pipelines versus Input and Output Redirection

Pipelines versus Input and Output Redirection

Input and output redirection:

- Input redirection creates a linkage between a file and a process.
- Output redirection creates a linkage between a process and a file.

Pipelines:

- A pipeline creates a linkage between two processes.

H2572 6-3. 59 © 1990 Hewlett-Packard Company

Student Notes

Input and output redirection is used to direct information (data) between a process and a file. Pipelines are used to direct information between processes.

Pipelines are an elegant solution to the problem of having the output of one program become the input to another program without having to know at program design time all of the possible ways a creative user will want to use the program. Pipelines reinforce the HP-UX idea that a program should do one thing very well.

6-4. SLIDE: Some Useful Filters

Some Useful Filters

- tee
- cut
- pr

H2572 6-4. 60 © 1990 Hewlett-Packard Company

Student Notes

The tee command is used to tap a pipeline. At the point where it is inserted in a pipeline, tee passes the data, without modification, both to the next command along the pipeline and to the files named as its arguments.

The cut command is used to report selected fields from lines of input.

The pr command is used to format output.

See tee(1), cut(1), and pr(1) in the *HP-UX Reference Manual* for more information.

6-5. SLIDE: The tee Command

The tee Command

Tap the pipeline.

Syntax:

```
tee [-a] file [file ... ]
```

Example:

```
$ ls | tee unsorted.ls | sort

$ ls | tee -a unsorted.ls | sort

$ cat unsorted.ls
```

H2572 6-5. 61 © 1990 Hewlett-Packard Company

Student Notes

The tee command is used to tap a pipeline. Tee reads from standard input and writes its output to standard output *and* to the specified file. If the -a option is used, then tee appends its output to the file instead of overwriting it.

See tee(1) in the *HP-UX Reference Manual* for more details.

6-6. SLIDE: The cut Command

The cut Command

Cuts columns or fields from files or stdin.

Syntax:

```
cut -clist [file . . ]
cut -flist [ -dcharacter ] [-s] [file . . . ]
```

Examples:

```
$ cut -f3,7 -d: /etc/passwd | sort -nr

$ date | cut -c1-3

$ ps -ef | cut -c48- | sort -d
```

H2572 6-6. 62 © 1990 Hewlett-Packard Company

Student Notes

The cut command is used as a filter to extract (and pass to stdout) columns or fields from the standard input or specified file. The -c option is for cutting columns, and -f is for fields.

The syntax is described in the slide.

A *list* is a number sequence used to tell cut which fields or columns are desired. There are several permissible formats for this list:

A-B	Fields or columns *A* through *B* inclusive.
A-	Field or column *A* through the end of the line.
A,B	Fields or columns *A* and *B*.
-B	Fields or columns from the beginning of the line through B.

Any combination of the above is also permissible. For example:

```
cut -f1,4,6-9  /tmp/testfile
```

would cut fields 1, 4, and 6 through 9 from the file.

Field cutting has a special option, -d, for specifying the separating character between fields (the delimiter). The delimiter defaults to a [Tab] character unless otherwise specified.

Also, the -s option, when cutting fields, will discard any lines that do not have the delimiter. Usually, these lines are passed through with no changes.

The command "cut" starts numbering its fields with one, while "sort" starts with zero. One easy way to remember this is to note that "sort" has a zero (really "o") in it, while "cut" does not.

See cut(1) in the *HP-UX Reference Manual* for more details.

6-7. SLIDE: The pr Command

The pr Command

Syntax:

`pr [-option] [file]` Formats stdin and produces stdout.

Examples:

```
$ ls | pr -3 -t
 pr -n3 -15 -p /etc/passwd
```

H2572 6-7. 63 © 1990 Hewlett-Packard Company

Student Notes

The pr command is used to format output. Typically, this is done before sending the output to a line printer. Although there are many options to pr, we will discuss only a few:

-*k*	Produces *k*-column output.
-a	Produces multicolumn output.
-t	Removes the trailer and header.
-d	Double-spaces the output.
-w*N*	Sets the width of a line to *N* characters.
-l*N*	Sets the length of a page to *N* lines.

-n*CK*	Produces *K*-digit line numbering, separated from the line by the character *C*. *C* defaults to Tab.
-p	Pauses and waits for Return before each page.

For more details, see pr(1) in the *HP-UX Reference Manual*.

6-8. SLIDE: Paging from a Pipeline

Paging from a Pipeline

Intelligently display the contents of a file.

Syntax:

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

or

```
pg [filename . . . ]
```

Examples:

```
$ ls | sort | more

$ ls | sort | pg

$ primes 1 10000 | pr -5 | more
```

H2572 6-8. 64 © 1990 Hewlett-Packard Company

Student Notes

Very often, the output from a pipeline is more than one page long. This can be awkward to handle; however, there is an easy way to solve this dilemma. If we end the pipeline with one of HP-UX's paging programs (more or pg), then the generated output will be paged according to the rules of the program used. For example, if we end the pipeline with the more program, then the first page of the output will be displayed and successive pages can be seen by pressing the space bar.

6-9. SLIDE: Printing from a Pipeline

Printing from a Pipeline

lp located at end of pipeline sends output to printer.

Syntax:

```
. . .  | lp
```

Request id is lprinter-551 (standard input).

Example:

```
$ cut -f3,7 -d: /etc/passwd | sort -n | lp
```

H2572 6-9. 65 © 1990 Hewlett-Packard Company

Student Notes

Recall that the lp command can be used to send output to the printer. Now we will see another way of printing output.

When the desired output is generated from the end of a pipeline, we can print it by continuing the pipeline and ending it with the lp command.

The slide shows an example.

6-10. SLIDE: Redirecting in a Pipeline

Redirecting in a Pipeline

- Any command on the left of a pipe symbol can only redirect input and errors.
- Any command on the right of a pipe symbol can only redirect output and errors.
- Any command between two pipe symbols can only redirect errors.

H2572 6-10. 66 © 1990 Hewlett-Packard Company

Student Notes

If a command appears on the left of a pipe symbol, then its stdout is written to the pipe. Therefore, any command on the left of a pipe symbol can only redirect its stdin and stderr.

If a command appears on the right of a pipe symbol, then its stdin is read from the pipe. Therefore, any command on the right of a pipe symbol can only redirect its stdout and stderr.

However, any command that appears between two pipe symbols in a pipeline must read its stdin from the pipe and write its stdout to the pipe. Therefore, the only redirection that can be done in the middle of a pipeline is error redirection.

6-11. LAB: Pipelines

Lab Objective

To practice constructing pipelines and using some additional filters.

Directions

Complete the following exercises and answer the associated questions.

1. Construct a pipeline that counts the number of lines in /etc/passwd that contain the pattern "user[0-9][0-9]*". Now count the lines that do *not* contain the pattern.

2. Construct a pipeline that will count the number of users presently logged on.

3. Construct a pipeline to print a sorted list of files (and directories) in your current working directory. Print the list in a three-column format with no header or trailer. Is the sort command needed for this task or not?

4. Construct a single pipeline that delivers an unsorted version of new.jersey to a file called new.notsort and a sorted version to the file new.sort. Hint: Look up the cat command.

5. Construct a pipeline to pass to the printer a sorted list of files in your directory. The list should have line numbers and be printed 10 lines per page with a form feed after each page. Hint: Look up the options to the pr command.

6. Construct a pipeline to obtain a listing of just the user_ids of those presently logged into the system.

7. Construct a pipeline to obtain a listing of just the names and ownership of each process currently running on the system.

8. Construct a single pipeline that will create a hard copy of your directory structure as well as a file containing the same information.

Module 7 — An Introduction to Shell Programming

Objectives

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

- Describe the use of the positional parameters 1–9.
- Describe the use of the special shell variables *, #, and $.
- Describe the use of the shift and read commands.

7-1. SLIDE: Introduction to Shell Programming

Introduction to Shell Programming

- A shell program is a regular file containing HP-UX commands or shell intrinsics.
- The shell file's permissions should be at least "read" and "execute."
- To execute the shell program, type the name of the file at the shell prompt.

H2572 7-1. 67 © 1990 Hewlett-Packard Company

Student Notes

As we have seen, the shell is a command interpreter. It interprets the commands that you type at the keyboard. However, it is often desirable to execute a number of commands a number of times. For this reason, we have the ability to build shell programs. These shell programs can also be called command files, batch files, or shell scripts.

A shell program is, simply put, a list of shell commands in a file to execute. The commands in the shell program are executed just as if they had been typed at the keyboard. The shell simply reads the commands and executes them, one at a time, until the end of the file has been reached.

Because the shell must be able to read the shell program and execute each line, the shell program should have read and execute permission set.

To execute a shell program, simply type the name of the shell file at the shell prompt. (The shell program must be in one of the directories in your PATH.)

7-2. SLIDE: Arguments To Shell Programs

Arguments To Shell Programs

Command line:

```
$ sh_program arg1 arg2 arg3  . . .  arg9
```

Inside the shell program:

Shell Command	Output
echo $0	sh_program
echo $1	arg1
echo $2	arg2
.	.
.	.
.	.
echo $9	arg9

H2572 7-2 68 © 1990 Hewlett-Packard Company

Student Notes

We often want to transmit information from the command line into a shell program. That is, we want to write generic programs that perform specific functions when given specific information. As we will see, it is possible to write shell programs that are treated by the system just like any other command (program).

Any shell program can be invoked with a command line. Arguments on the command line can be referenced from within the shell program. The referencing scheme is based on the relative position of the arguments on the invoking command line. Such arguments are called **positional parameters.** Positional parameters get the value of the corresponding argument from the command line. Positional parameters can be used within a shell program just like other shell variables. That is, to dereference the variables, we use a $ symbol. We can use the positional parameters to reference up to nine command-line arguments ($1–$9).

The zero command-line argument is the command name. Thus, the positional parameter 0 refers to the command name.

Later, we will see a way to access more than nine arguments.

7-3. SLIDE: Some Special Shell Variables — # and *

Some Special Shell Variables — # and *

#	The number of command line arguments.
*	The entire argument string.
$	The process id of the shell program.

Example:

- Command line:

```
$ sh_program arg1 arg2 arg3
```

- Inside the shell program:

Shell Command	Output
echo $#	3
echo $*	arg1 arg2 arg3
echo $$	*process ID number*

H2572 7-3. 69 © 1990 Hewlett-Packard Company

Student Notes

There are several variables that the shell keeps current. We should not change these variables, but we can get their value.

The # variable holds the number of arguments on the command line. In the example, # would have the value 3.

The * variable holds the value of the entire command line except for the command itself. All command line arguments can be referenced with * whether there are less than, equal to, or more than nine arguments. In the example, * would contain arg1 arg2 arg3.

The $ variable holds the process identification number (PID) of the active shell program.

7-4. SLIDE: The shift Command

The shift Command

Syntax:

`shift [n]`

- Shifts all strings in * left *n* positions.
- Decrements # by *n*.
- *n* defaults to 1.

Example:

- Command line:

```
$ sh_program arg1 arg2 arg3
```

- Inside shell program:

Shell Command	Output
`echo $#`	3
`echo $*`	arg1 arg2 arg3
`shift 2`	
`echo $#`	1
`echo $*`	arg3

H2572 7-4. 70 © 1990 Hewlett-Packard Company

Student Notes

The shift command renames the positional parameters. For example, after a shift command is encountered, the leftmost parameter in $* can be dereferenced by accessing $1.

After a shift *n*, all parameters in * are moved to the left *n* positions and # is decremented by *n*. The default for *n* is 1.

The shift command does not affect the positional parameter 0.

7-5. SLIDE: The read Command

The read Command

Syntax:

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

Example:

Shell Program	Input	Output
echo enter a sentence		enter a sentence
read A	hi there	
echo $A		hi there
echo enter three words		enter three words
read A B C	a bee flys	
echo $C $A $B		flys a bee
echo enter three letters		enter three letters
read A B C	W X Y Z	
echo $C $B $A		Y Z X W

H2572 7-5. 71 © 1990 Hewlett-Packard Company

Student Notes

Sometimes, instead of specifying arguments on a command line, we want to prompt the user for information. The read command is used to read information typed at the terminal. Do not confuse positional parameters with variables read. Positional parameters are specified on the command line; variables that are read are specified as input to the executing program *after* a prompt has been issued by an echo command.

If there are more variables in the read command than there are (white space delimited) words of input, the leftover variables are assigned to NULL. If the user inputs more words than there are variables, all leftover data is assigned to the *last* variable in the list.

Once set, these variables are accessed just like all shell variables.

7-6. SLIDE: The expr Command

The expr Command

Syntax:

```
expr expression
```

Example A:

```
$ X=3
$ Z=`expr $X + 4`
$ echo $Z
7
```

Example B:

```
$ X=abcdef
$ Z=`expr $X : ".*"`
$ echo $Z
6
```

H2572 7-6. 72 © 1990 Hewlett-Packard Company

Student Notes

The arguments to the expr command are taken as an expression. The expr command evaluates its arguments and writes the result to standard output. This command is often used in conjunction with command substitution. We will not look at all of the expr command's properties.

The first use of the expr command is for simple arithmetic. The operators +, —, *, and / stand for integer addition, subtraction, multiplication, and division respectively. The slide shows a typical example.

The second use of the expr command is for string matching. This is not particularly useful except in one case: finding out the length of a string in a variable. The slide shows an example. In this example, the expr command makes use of Regular Expressions, a very powerful pattern matching concept. Other commmands such as grep, ed, and sed also use them. We will not, however, discuss them in this course.

7-7. SLIDE: Miscellaneous Techniques

Miscellaneous Techniques

- `$ ksh` *sh_program arguments*
  - sh_program does not have to be executable.
  - sh_program does have to be readable.
- `ksh -x` *sh_program arguments*
  - Each line of *sh_program* is interpreted and printed before being executed.
- Document shell programs by preceding a comment with a number sign (#).
  - The rest of the line after the # sign is ignored by the shell.

H2572 7-7. 73 © 1990 Hewlett-Packard Company

Student Notes

The # character is used to document shell programs. A # character, located at the beginning of the line or preceded by white space, indicates to the shell that whatever follows (up until the Return) is to be ignored.

An alternative way to execute a shell program is to use the following:

`ksh` *shell_program arguments*

This actually executes a subshell and tells the subshell to read its commands from the shell program specified. When executed this way, the shell program does *not* have to be executable.

Although there is no debugger for a shell program, the command

`ksh -x` *shell_program arguments*

will display each command in the shell program *before* executing it.

7-8. SLIDE: Interactive Shell Commands

Interactive Shell Commands

Interactive shell commands are multiple-line shell commands. They can be typed directly at the keyboard.

The secondary prompt (>) will appear until the command is finished.

Example:

```
$ if [ $X = 3 ]
> then
> echo hello
> else
> echo goodbye
> fi
```

H2572 7-8. 74 © 1990 Hewlett-Packard Company

Student Notes

Any command that can be typed at the keyboard in response to a prompt can be used in a shell program. Consequently, all of the commands that can be used in a shell program can be typed at the keyboard. This means that you can type fairly complex commands *directly* at the terminal in response to a shell prompt.

The slide shows an example of the multiple-line shell command if. We will be seeing the if command in the next module. Note the secondary > prompt. This secondary prompt will appear any time the user presses [Return] and the shell thinks that the user is not finished with the command.

7-9. SLIDE: .profile

.profile

The shell program .profile is executed every time you log into the Bourne (sh) or the Korn (ksh) shell.

- You typically own your .profile.
- .profile must be in your HOME directory.
- Usually you can customize your .profile.

H2572 7-9 75 © 1990 Hewlett-Packard Company

Student Notes

As we mentioned before, there is a file called .profile which gets executed when you log in. The dot (.) before the name is not a typographical error. This file can contain many useful commands and programs that you want executed before you start your session. Thus, if you want any shell commands to execute every time you log in, just include them in .profile.

The .profile file is a shell program that does *not* have to be *executable*.

7-10. LAB: An Introduction to Shell Programming

Lab Objective

To practice using some miscellaneous objects and commands used in shell programming.

Directions

Complete the following exercises and answer the associated questions.

1. If the command line for a shell program is:

```
$ myshellprog abc def -d -4 +900 xyz
```

   then what will be printed out from the shell program if it contains:

```
echo $#
echo $3
echo $7
echo $*
echo $0
```

2. If the shell program invoked by the command line in the previous exercise contained a shift 2 command as the first line, what would be the results of:

```
echo $#
echo $3
echo $7
echo $*
echo $0
```

3. What would be the output of the following shell program if, when prompted, the user typed in the input:

```
James A. Smith, Jr.
```

Shell program:

```
echo "Please type in your first, middle, and last names"
read first middle last
echo "$last, $first $middle"
```

4. Write a shell program called "reverse" that will receive up to nine arguments and list the arguments in reverse order.

5. Write a shell program called "myecho" that will do the following:

   a. Print the number of arguments passed to it. (Assume a maximum of nine arguments.)

   b. Print the first three arguments on separate lines.

What argument list will produce the following output from this shell program?

```
I cannot
seem to
find my KEYS.
```

6. Write a shell program called "alpha" that will display the first and last command-line arguments.

7. Write a shell program called "info" that will prompt the user for:

   a. Name.

   b. Street address.

   c. City, State, and Zip.

   The program should then store the replies in variables and display what the user entered.

8. Write a shell program called "double" that will prompt the user for a single integer and display twice its value.

9. Write a shell program called "home" that prompts for any user's login_id and returns the value of that user's HOME directory. Recall that the HOME directory is the sixth field in the /etc/passwd file.

10. Write a shell program called "clock" that will:

    a. Separate the hours, minutes, and seconds form the date command.

    b. Assign these values to variables.

    c. Output the following:

       "The time is now *hr* o'clock *min* minutes and *sec* seconds."

Module 8 — Shell Programming: Branches

Objectives

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

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

8-1. SLIDE: Return Codes

Return Codes

The shell variable ? holds the return code of the last command executed. A return code of 0 means true or no error. A return code of nonzero means false or that an error has occurred.

Examples:

```
$ ls
file1 file2 file3
$ echo $?
0
$ echo $?
0
$ cp
Usage: cp f1 f2
       cp [-r] f1  . . .  fn d1
$ echo $?
1
$ echo $?
0
```

H2572 8-1. 78 © 1990 Hewlett-Packard Company

Student Notes

To allow some limited communication between processes, HP-UX commands all generate return codes. At the termination of a command, the shell variable ? will be set to an integer and can be accessed by:

```
echo $?
```

In the shell, a return code of 0 is, by convention, a normal termination. A nonzero return code usually indicates that the command that just terminated failed in some way.

The command true sets the return code to the true value and the command false sets it to the false state.

The shell has a command called if which examines returns codes and makes a control-flow decision based upon them. We will see the if construct *after* we discuss the test command.

8-2. SLIDE: The test Command

The test Command

Syntax:

```
test expression
```

or

```
[ expression ]
```

The test command evaluates the expression and sets the return value to 0 if the expression is true and to nonzero if the expression is false.

We can test three types of things:

- Files.
- Strings.
- Integers.

H2572 8-2. 77 © 1990 Hewlett-Packard Company

Student Notes

The test command is used to evaluate expressions and generate a return code. It takes arguments that form logical expressions and evaluates the expressions. The test command writes nothing to the standard output, but it sets the return code to 0 if the expressions evaluate *true* and to 1 if the expressions evaluate to *false*.

The test command is most often used with the if and while constructs for conditional flow control.

We are using it here without any additional commands so that we can concentrate on the test command itself.

The test command can also be invoked as [*command*]. This is intended to assist readability. If the [ ] form is used, the "[" and "]" must have white space around them.

The test command can test three types of things:

- Files.
- Strings.
- Numbers.

8-3. SLIDE: The test Command — File Tests

The test Command — File Tests

Evaluates file name according to the option.

Syntax:

```
test -option filename
```

Examples:

```
$ test -f new.jersey
$ echo $?
0
$ test -d new.jersey
$ echo $?
1
```

H2572 8-3. 78 © 1990 Hewlett-Packard Company

Student Notes

One important use of the test command is to test the status of files. For instance:

`test -f` *filename*

will return true (0) if the file exists and is a regular (not directory or device) file.

`test -s` *filename*

will return true (0) if the file exists and has a size greater than 0.

There are many other file tests available. A partial list follows:

-r	True if the file exists and is readable.
-w	True if the file exists and is writable.
-x	True if the file exists and is executable.
-d	True if the file exists and is a directory.

See test(1) in the *HP-UX Reference Manual* for further details.

8-4. SLIDE: The test Command – String Tests

The test Command — String Tests

Compares strings.

Syntax:

[string1 = string2]

[string1 != string2]

Examples:

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

H2572 8-4. 79 © 1990 Hewlett-Packard Company

Student Notes

The test command can be used to test two strings for equality or inequality.

=	Is equal to.
!=	Is not equal to.

When you are testing a shell variable, it is wise to allow for the possibility that the variable may contain nothing. For example, look at the test statement in the following:

```
[ $XX = yes ]
```

If *XX* has not been set, that is, *XX* is null, the shell will perform the variable substitution and the command that the shell will attempt to execute will be:

```
[ = yes ]
```

which is malformed and guaranteed to cause a syntax error. A simple way around this is to quote the variable to be tested. For example:

```
[ "$X" = yes ]
```

This will assure that something will be generated, even if it is NULL.

Note also that if there is any possibility that a variable may get set to a string that contains blanks, it is wise to surround the variable name with double quotes.

Note that the [...] syntax is desirable for string tests.

8-5. SLIDE: The test Command – Numeric Tests

The test Command — Numeric Tests

Syntax:

[*number relation number*]   Compares numbers according to the given relation.

Relations:

`-lt`	Less than.
`-le`	Less than or equal to.
`-gt`	Greater than.
`-ge`	Greater than or equal to.
`-eq`	Equal to.
`-ne`	Not equal to.

Example:

```
$ X=3
$ [ "$X" -lt 7 ]
$ echo $?
0
```

H2572 8-5.   80   © 1990 Hewlett-Packard Company

Student Notes

The test command can also be used to perform numeric tests. In numeric testing, the command is only meaningful with integers.

The operators that are used to compare numbers are different than the operators that are used to compare strings.

These numeric operators are the following:

-lt	Is less than.
-le	Is less than or equal to.
-gt	Is greater than.
-ge	Is greater than or equal to.
-eq	Is equal to.
-ne	Is not equal to.

Again, the [...] syntax is standard for numeric tests.

8-6. SLIDE: The test Command – Other Operators

The test Command — Other Operators

-o	OR
-a	AND
!	NOT
\(\)	GROUPING

H2572 8-6. 81 © 1990 Hewlett-Packard Company

Student Notes

There are a few operators that are valid in a test command's expression whether we are testing files, strings, or numbers. These operators are:

-o	OR
-a	AND
!	NOT
\(\)	GROUPING

For the return code of an OR expression to be 0 (true), one or the other or both expressions must be true. For the return code of an AND expression to be 0 (true), both expressions must be true. The subexpressions can be

surrounded by parentheses for clarity. However, the parentheses must be escaped using a backslash to prevent the shell from interpreting them as special characters.

The NOT operator is not needed when you are doing string or number tests. Why bother to:

```
test ! a = b
```

when:

```
test a != b
```

works as well? However, the NOT operator is very useful with file tests. For example, it can be used effectively to:

test ! -d *filename*

when:

test -f *filename* -o -c *filename* -o -b *filename* . . .

is awkward and wasteful.

8-7. SLIDE: The If Construct

The if Construct

Used for single decision branch.

```
if
        list A
then
        list B
else
        list C
fi
```

1. list A is executed.

2. If the return value of the last command in list A is 0, execute list B, then jump to the first statement after fi.

3. If the return value of the last command in list A is not zero, execute list C, then jump to first statement after fi.

The else clause in the above example is optional.

H2572 8-7. 82 © 1990 Hewlett-Packard Company

Student Notes

The if construct is used to make decisions based on return codes. It sets up a control structure such that if a command executes successfully, a command list will be executed. The if construct provides a complete if/then/else control capability.

The slide shows the allowable if constructs. A **command list** is a list of shell commands separated by semicolons or Return.

The execution of the if construct is as follows:

1. Command list A is executed.
2. If the return code of the *last* command in command list A is a 0 (TRUE), we execute command list B, and then jump to the first statement following the fi.
3. If the return code of the *last* command in command list A is *not* 0 (FALSE), we execute command list C (if it exists), and then jump to the first statement following the fi.

8-8. SLIDE: The if Construct – Examples

The if Construct — Examples

Example A:

```
$ if
>   echo hello
>   grep x /etc/pas
swd
> then
>   echo "there"
> else
>   echo "gone"
> fi
hello
gone
$
```

Example B:

```
$ X=tree
$ if
>   test -d $X
> then
>   ls $X
> else
>   more $X
> fi
dir1 dir2 dir3
$
```

H2572 8-8. 83 © 1990 Hewlett-Packard Company

Student Notes

The slide shows two examples of using the if construct. Note that in the first example the grep command returns 0 if it finds the pattern in the file and 1 if it does not. It is common practice to redirect the output of grep to /dev/null if we do not wish to see the standard output of grep, but do wish to see if the pattern is in the file.

8-9. SLIDE: The case Construct

The case Construct

Used for multiway branching.

```
case word in
      pattern1) list A
        ;;
      pattern2) list B
        ;;
        .
        .
        .
      patternN) list N
        ;;
esac
```

1. word is compared to pattern1. If they match, then list A is executed and then we jump to the first statement following the esac.

2. if pattern1 does not match, pattern2 is compared, and so on. If there is no match, jump out.

H2572 8-9. 84 © 1990 Hewlett-Packard Company

Student Notes

The **case** construct is a control flow construct that provides multiway branching based on patterns.

The case construct controls program flow based on the word given. The word is compared, in order, against each pattern. When the first match is found, the associated list of commands is executed. The list of commands to be executed must end with two semicolons (;;). When the command list has been executed, we jump to the first statement after the esac.

A **word** is typically a dereferenced shell variable.

The **patterns** are built out of the shell file name generation characters even though we are not matching file names. There is also the addition of the | character which means OR.

Note that the close parenthesis and the semicolons are mandatory.

8-10. SLIDE: The case Construct – Pattern Examples

The case Construct — Pattern Examples

```
abc)

a??)

a*)

ab | cd)

a*b?[123])

a[!de]*)

[!a-z*)
```

H2572 8-10 85 © 1990 Hewlett-Packard Company

Student Notes

The slide shows several examples of case patterns. Try and determine what strings will match the patterns. Your instructor will give you the answers.

8-11. SLIDE: The case Construct – Examples

The case Construct — Examples

Example A:

```
$ X=abc
$ case $X in
> ?*a*) echo has an 'a' ;;
>   a*) echo starts with 'a' ;;
>    *) echo no 'a' ;;
> esac
starts with a
```

Example B:

```
$ X=xyz
$ case $X in
> abc|def) echo abc or def ;;
>   g*|*g) echo starts with a g
>          echo or ends with a g ;;
>       *) echo none of the above ;;
> esac
none of the above
$
```

H2572 8-11. 86 © 1990 Hewlett-Packard Company

Student Notes

The slide shows two examples of the case construct.

8-12. LAB: Shell Programming – Branches

Lab Objective

To practice using the test, if, and case commands for branching within shell programs.

Directions

Complete the following exercises and answer the associated questions.

1. Set a variable called *X* equal to the value abc. Now type an interactive test command that sets a return code of 0 if the value of *X* is abc and a value of nonzero if it is not. Check the value of the return code with an echo command.

2. Create an interactive if construct that will echo yes if *X* is equal to abc and no if it is not.

3. Write a shell program that counts the number of command-line arguments and echoes an error message if there are not exactly three arguments or echoes the arguments themselves if there are three.

4. Write a shell program that prompts the user for input and takes one of three possible actions:

   a. If the input is A, the program should echo "type A".

   b. If the input is B, the program should echo "OKAY".

   c. If the input is C, the program should terminate.

5. Write a shell program that responds to command-line arguments as follows:

   a. If the first argument on the command line is -a, the user is prompted for a string. If the string is D, run the date command; otherwise, run the ls command.

   b. If the first argument on the command line is -b, the user is prompted for a number. The program echoes an appropriate message if the number is greater than 100 or less than or equal to 100.

   c. If the first argument on the command line is -c, the user is prompted for a file name. The program tells whether that file name is a directory or not.

   d. If there are not any command line arguments, or if the option is not -a, -b, or -c, echo an error message to indicate that condition.

6. Modify myecho from the previous lab so that if any of the first three arguments are missing, myecho will print an error message instead of the argument.

   For example:

```
Argument 1 is missing!
Argument 2 is missing!
Argument 3 is missing!
```

7. Modify home from the previous lab so that an error message is displayed if the user_id is not a valid user_id.

8. Modify clock from the previous lab, so that the value for hrs will be displayed as a 12 hour, AM/PM clock.

9. (Advanced) Modify double from the previous lab so that the shell program will check for:

   a. A null entered.

   b. A noninteger value.

   c. More than one value entered.

10. Write a shell program called mycp that will copy one file to another. Be sure to check for the following:

    a. Is the mycp command being used properly?

    b. Are the source files and destination files the same?

    c. Does the source file exist?

    d. Does the target file exist, and if it does, should it be overwritten?

Module 9 — Shell Programming: Loops

Objectives

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

- Use the while construct to repeat a section of code while some condition remains true.
- Use the until construct to repeat a section of code while some condition remains false.
- Use the iterative for construct to repeat a sequence of commands a known number of times (based on the number of input strings).

9-1. SLIDE: Loops – An Introduction

Loops — An Introduction

A loop causes repeated execution of a list of commands.

Branches		Loops		
if	case	for	while	until
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
fi	esac	do	do	do
		.	.	.
		.	.	.
		.	.	.
		done	done	done

H2572 9-1. 87 © 1990 Hewlett-Packard Company

Student Notes

Unlike branches, which start with a keyword and end with the keyword in reverse (if/fi and case/esac), loops have do/done surrounding the body of the loop.

Whereas a branch was meant for flow control, a loop is designed for repeated execution of the same commands.

9-2. SLIDE: The while Construct

The while Construct

Syntax:

```
while
        list A
do
        list B
done
```

Execution:

1. *list A* is executed.
2. If the return value of the last command in *list A* is 0, execute *list B* and go to step 1.
3. If the return value of the last command in *list A* is not 0, jump to the first command following done.

H2572 9-2. 88 © 1990 Hewlett-Packard Company

Student Notes

The while construct is a looping mechanism provided by the shell.

The execution is as follows:

4. Command list A is executed.
5. If the return code of the *last* command in list A is 0 (TRUE), we execute list B. Then we go back to step 1.
6. If the return code of the *last* command in list A is *not* 0 (FALSE), we skip to the first command following the done keyword.

The looping continues until the last command in list A returns nonzero, at which point processing continues with the first command after the done.

9-3. SLIDE: The while Construct – Examples

The while Construct — Examples

Example A:

```
$ X=1
$ while [ "$X" -le 10 ]
> do
> echo hello
> X=`expr $X + 1`
> done
hello
  .
  .
  .
hello
```

Example B – In a shell program:

```
while [ $# -ne 0 ]
do
        if test -d $1
        then
                ls $1
        else
                cat $1
        fi
        shift
done
```

H2572 9-3. 89 © 1990 Hewlett-Packard Company

Student Notes

The slide shows some examples of the while construct. In example A, the expr command is used as a loop counter.

9-4. SLIDE: The until Construct

The until Construct

Syntax:

```
until
        list A
do
        list B
done
```

Execution:

1. *list A* is executed.
2. If the return value of the last command in *list A* is not 0, execute *list B* and go to step 1.
3. If the return value of the last command in *list A* is 0, jump to the first command after done.

H2572 9-4. 90 © 1990 Hewlett-Packard Company

Student Notes

The until construct is a looping mechanism similar to the while construct.

The execution is as follows:

1. Command list A is executed.
2. If the return code of the *last* command in list A is *not* 0 (FALSE), we execute list B. Then we go back to step 1.
3. If the return code of the *last* command in list A is 0 (TRUE), we skip to the first command following the done keyword.

The looping continues until the last command in list A returns zero, at which point processing continues with the first command after the done.

9-5. SLIDE: The until Construct — Examples

The until Construct — Examples

Example A:

```
$ X=0
$ until [ "$X" -eq 10 ]
> do
> echo hello
> X=`expr $X + 1`
> done
hello
  .
  .
  .
hello
```

Example B – In a shell program:

```
until [ $# -eq 0 ]
do
     if test -d $1
     then
          ls $1
     else
          cat $1
     fi
     shift
done
```

H2572 9-5. 91 © 1990 Hewlett-Packard Company

Student Notes

The slide shows some examples of the while construct.

9-6. SLIDE: The for Construct

The for Construct

Syntax:

```
for var in list
do
        list A
done
```

var is any shell variable, and *list* is a space-delimited list of strings. The number of strings in a list determines the number of iterations of the loop.

Execution:

1. *var* is set to the first string in *list*.
2. *list A* is executed.
3. *var* is set to the next string in *list*. Go to step 2.
4. Repeat until all strings have been used.

H2572 9-6. 92 © 1990 Hewlett-Packard Company

Student Notes

In the slide, the keywords are for, in, do, and done; var can be any shell variable (not dereferenced); and list is a string comprised of strings separated by blanks, new-lines, or tabs (IFS shell variable is used).

The construct works as follows:

1. The shell variable var is set equal to the first string in list.
2. Command list A is executed.
3. The shell variable var is set equal to the next string in list.
4. Return to step 2.

This sequence continues until all of the strings in list have been assigned once to var. After the loop is finished, execution resumes at the first command line following the done.

9-7. SLIDE: The for Construct – Examples

The for Construct — Examples

Example A:

```
$ for i in 1 2 3 4
5
> do
>    echo hello $i
> done
hello 1
hello 2
hello 3
hello 4
hello 5
```

Example B:

```
$ for X in *
> do
> if test -d $X
> then
>    ls $X
> else
>    cat $X
> fi
> done
```

H2572 9-7. 93 © 1990 Hewlett-Packard Company

Student Notes

The slide shows two examples of the for construct. Example B has a subtle potential problem in it. Can you find it?

9-8. SLIDE: The break, continue, and exit Commands

The break, continue, and exit Commands

break *[n]*	Causes any loop to terminate and passes control to the next command after the [the *nth*] done.
continue *[n]*	Stops the current iteration of the loop and begins execution with the next iteration [of the *nth* enclosing loop].
exit *[n]*	Stops the execution of the shell program and sets the return code to *n*.

H2572 9-8. 94 © 1990 Hewlett-Packard Company

Student Notes

Within any of the three loops that we have discussed — while, until, and for — you may use the break and continue commands to affect the regular flow.

The break command will cause the loop to terminate and control to be passed to the command immediately following the done keyword.

The continue command is slightly different. When encountered, the continue command will skip the remaining commands in the body of the loop and transfer control to the beginning of the initialization list (in the case of the while and until loops). If used in a for loop, the continue will skip the remaining commands in the loop and cause the variable to be set to the next item in the list.

The exit command will stop the execution of a shell program and set the return value for the shell program to the argument, if specified. If no argument is supplied, the return value of the shell program is set to the return value of the command that executed immediately prior to the exit.

9-9. LAB: Shell Programming – Loops

Lab Objective

To practice the looping constructs available in the shell.

Directions

Complete the following exercises and answer the associated questions.

1. Write an interactive for loop to mail the file called mailtest, in your HOME directory, to everyone who is logged on.

2. Write a shell program to display the command line arguments, one argument to be displayed on each output line.

3. Write a shell program to display the phrase "hello world" 100 times. Do not use a for loop!

4. Write a shell program named ison that will run in the background and check every 60 seconds whether a particular user has logged into the system. When the user logs in, print a message on your terminal flagging the login, and report what terminal the user logged into. (Hint: Look into the sleep command.)

5. Create a directory called $HOME/.waste. Write a shell program called myrm that will move all of the files you delete into the .waste directory, your wastebasket. This is a useful tool that will allow restoration of files after they have been removed. Remember, HP-UX has no "unremove" capability.

   Include options to myrm:

-l	List contents of the wastebasket.
-r	Empty out wastebasket.

6. Write a shell program that reads the command-line arguments, counts the number of options (those arguments that start with a "-"), and counts the number of arguments (those arguments that do not start with a "-").

Advanced	Output all of the options on one line and all of the arguments with their counts on another line.
Ultra-advanced	Sort the options and arguments before printing them out.

Module 10 — Shell Programming: Signals and Traps

Objectives

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

- Describe the use of the kill and trap commands.
- Use the trap command for producing interrupt-driven code.

10-1. SLIDE: What Is a Signal?

What Is a Signal?

Certain events cause signals to be sent to executing processes.

For example:

- Logout sends signal 1 to background processes.
- Pressing DEL sends signal 2 to foreground processes.
- kill PID sends signal 15 to process number PID.

H2572 10-1. 95 © 1990 Hewlett-Packard Company

Student Notes

Certain events are important to a process. For example, hanging up or pressing the DEL or Break keys has an effect on a process. When an event such as these occurs, HP-UX sends a **signal** to the process. By default, most signals will cause a process to terminate.

Each signal has a unique number. For example, hangup has signal number 1, and delete has signal number 2. The kill command allows users to send specific signals to processes. The kill command defaults to sending a signal 15.

Two other signals that can be sent to a shell program are 3 and 9. Signal 3 is the quit signal, and it generates a core dump as well as aborting the process. On most systems, typing a CTRL-\ will cause a quit signal to be sent to the foreground process.

Signal 9 is the absolute kill signal. It cannot be caught or ignored.

10-2. SLIDE: What Is a Trap?

What Is a Trap?

A trap is a way to catch a signal and possibly perform some action.

H2572 10-2. 96 © 1990 Hewlett-Packard Company

Student Notes

A **trap** is a way of interrupting current processing in response to a signal so that a predefined routine will run. This predefined routine is sometimes called an **interrupt service routine.** It is executed only when a specific interrupt (signal) occurs. We can have several different routines for several different specific signals.

10-3. SLIDE: The kill Command, Revisited

The kill Command, Revisited

Recall:

kill [-*signo*] *PID* [*PID* . . .]

Example:

```
$ myprog &
[1] 7467

$ kill -9 7467
[1] +killed    myprog &

$
```

H2572 10-3. 97 © 1990 Hewlett-Packard Company

Student Notes

Recall the kill command. This command is used to send signals to processes. It is called the kill command because the *default* action for most signals is for the process to die.

10-4. SLIDE: The trap Command

The trap Command

Perform *cmds* upon receipt of *signo*.

Syntax:

```
trap 'cmds' signo [signo]
```

Example:

```
$ cat myprog
trap 'echo bye; exit' 2
while true
do
     echo hello
done
$ myprog
hello
hello
hello
                                    Press[ DEL ]
bye
$
```

H2572 10-4. 98 © 1990 Hewlett-Packard Company

Student Notes

Normally, the receipt of any signal will cause the receiving process to abort. The trap command can be used in shell programs to "trap" signals before they can kill the process. This allows the programmer to change the action that is taken upon receipt of signals.

The trap commands are usually placed at the beginning of shell programs, but they can go anywhere. Traps are set by the trap command when the shell reads the trap command; the traps are sprung when a signal is received.

When the trap command is used, signals are specified by their signal number ("signo" in the slide).

There are three things the trap command can do with signals:

- The trap command can "catch" signals. Instead of aborting the process, the signal will trigger the execution of specific shell commands (cmds). For example, if we have a shell program that creates temporary files, we probably want to remove those files before exiting when someone types a "delete". That way, we will not leave files scattered around if the process is aborted.
- The trap command can "ignore" signals. Instead of aborting the process, received signals can be ignored. This is useful in a shell program that accesses an important file. For example, if we have a shell program that modifies a log file of some sort, we do not want the file corrupted because the program was aborted in the middle of some data modification. In this case, we would use trap to ignore signals.
- The trap command can "reset" signals. After we have "caught" or "ignored" a signal, the trap command can be used to restore the default action. The default action is usually to terminate the process.

Signal 0 (zero) has a meaning only in shell programming. It is the signal that a process sends itself upon normal termination. This is sometimes used to specify what actions a shell program should take upon normal completion. It is not a practice that we recommend. It makes shell programs difficult to read because the trap command is placed at the beginning of the program and yet has an effect at the end of the program.

10-5. SLIDE: Ignoring Signals

Ignoring Signals

Syntax:

```
trap ' ' signo [signo]
```

Example:

```
$ cat myprog2
trap ' ' 2
while true
do
     echo hello
done
$ myprog2
hello
hello                       Press [DEL] ; ignored
hello

hello
hello
hello                       Press [Break]
hello

$
```

H2572 10-5. 99 © 1990 Hewlett-Packard Company

Student Notes

Instead of performing an interrupt service routine upon receipt of a signal, we can choose to do nothing. This is called ignoring a signal. There is an example in the slide.

10-6. SLIDE: Placement of the trap Command

Placement of the trap Command

Place trap before the use of temporary files.

Example: `trap 'rm /tmp/tempfile;exit' 1 2 3 15`

Place trap before the critical sections of code.

Example: `trap '' 1 2 3 15`

Reset trap to default action when done.

Example: `trap 1 2 3 15`

H2572 10-6. 100 © 1990 Hewlett-Packard Company

Student Notes

The trap command can be placed anywhere in a shell program. However, there are certain placements that will greatly improve your satisfaction with the results.

For example, if you have a section of your shell program that uses a temporary file, you may wish to set a trap that "cleans up" the temporary file and *then* exits upon receipt of a signal. This trap should be placed immediately before the section of code that uses the temporary file.

If you have a section of code that processes a particularly delicate file (such as one with read-only permissions) and you wish to edit that file, you would not want a signal to interrupt your editing and leave that file writable to everyone. Thus, you may place a trap that would ignore all signals (that you can ignore) *before* that section of code.

After that section of code, you can reset the signals to their default actions by using the trap command again, and this time you specify no interrupt service routine.

10-7. LAB: Shell Programming -- Signals and Traps

Lab Objective

To allow the student to practice using signals and the trap command.

Directions

Complete the following exercises and answer the associated questions.

1. Write a shell program that will write a message to everyone logged on until the [DEL] key is pressed. The [DEL] key sends signal number 2. When the [DEL] key is pressed, the program should exit and set the return code of the shell program to 255. You will need to look up the use of "< <" in the manual under ksh(1).

2. Write a shell program that will display a message and erase it every 3 seconds. (Hint: Check out the sleep(1) command). Run this program in the background and make it immune to signal numbers 1, 2, 3, and 15. How could you kill this program?

Module 11 — Offline File Storage

Objectives

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

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

11-1. SLIDE: Tape Usage

Tape Usage

- Must know the device file name for your tape device.
- Typical names might be:

```
/dev/rct/c1d0s2
/dev/rmt/0h
/dev/update.src
```

- Check with your system administrator.

H2572 11-1. 101 © 1990 Hewlett-Packard Company

Student Notes

There are many times when the average user of an HP-UX system will want to make copies of files onto some removable media. The most popular media available is cartridge or 9-track tape. This module is designed to give you the basics of making and restoring tape copies of files. Keep in mind that your system administrator is usually responsible for backing up the entire system; you should coordinate all tape backups through him or her.

11-2. SLIDE: The tar Command

The tar Command

Syntax:

```
tar key [ f device_file ] [file . . . ]
```

Examples:

- Create an archive:

```
tar cf /dev/rct/c1d0s2 new.jersey myfile
```

- Add to an archive:

```
tar rf /dev/rct/c1d0s2 herfile
```

- Get a table of contents:

```
tar tf /dev/rct/c1d0s2
```

- Extract a file from the archive:

```
tar xf /dev/rct/c1d0s2
```

H2572 11-2. 102 © 1990 Hewlett-Packard Company

Student Notes

The tar command is the tape file archiver. It saves and restores files onto magnetic tape or diskettes. Its function is controlled by its first argument, called the **key argument.**

Examples of valid key arguments are as follows:

r	Files are added to the end of the archive.
x	Files are extracted from the archive.
t	A table of contents of the archive is printed.
u	Files are added to the archive if they are new or modified.
c	A new archive is created.

Modifiers can be added to these keys. We will discuss only one of these modifiers, f, which causes tar to use the next argument as the name of the device file on which the archive should be created.

Some examples are shown in the slide.

11-3. SLIDE: The find Command, Revisited

The find Command, Revisited

```
$ cd /users/bwinkle
$ find . -print | (rest of the process)
```

or

```
$ find /users/bwinkle -print | (rest of the process)
```

H2572 11-3. 103 © 1990 Hewlett-Packard Company

Student Notes

As we saw in module 3, the find command is the only command that performs an automated search through the file system. It is very slow and uses a lot of CPU capacity. It should be used sparingly.

To back up files, we will use a three-step process. The steps are as follows:

1. Create a list of files to be backed up.
2. For each file, copy the contents of that file to standard output.
3. Redirect the output directly to the tape drive (9-track tape), or first pipe the output through a buffering mechanism and then to the tape drive (cartridge tape).

The find command in the slide shows two ways to set up the list of files.

11-4. SLIDE: The cpio Command

The cpio Command

Syntax:

```
cpio -o[cxv]
cpio -i[cxuvdm]
```

Examples (using 9-track device):

- Create an archive:

```
$ find . -print | cpio -ocvx > /dev/rmt/0h
```

- Restore an archive:

```
$ cpio -icudmvx < /dev/rmt/0h
```

- Get table of contents:

```
$ cpio -ict < /dev/rmt/0h
```

- Restore a single file:

```
$ cpio -icudm "filename" < /dev/rmt/0h
```

- Restore all file names matching pattern:

```
$ cpio -icudm '*filename*' < /dev/rmt/0h
```

H2572 11-4. 104 © 1990 Hewlett-Packard Company

Student Notes

This command makes archive copies of files and directories in HP-UX. It has two modes that we will discuss:

- For making backups:

```
cpio -o
```

In this mode, cpio reads the standard input for a list of file names and copies those files to the standard output along with path name and file attribute information.

- For restoring backups:

```
cpio -i
```

In this mode, cpio reads the standard input, which is assumed to be the product of a previous cpio -o backup, and recreates the files indicated by the input.

There are several options that we will use with the major options -o and -i.

-o	-i	Option Function
-c	-c	Writes header in ASCII format. (If used with -o, it must be used with -i.)
-x	-x	Handles special (device) files.
—	-u	Unconditionally restores. (If the file already exists, this option overwrites the file.)
—	-m	Retains current modification date. (Important for version control.)
—	-d	Recreates directory structure as needed.
-v	-v	Displays a list of files copied.
—	-t	Displays a table of contents.

Some examples are in the slide.

11-5. SLIDE: The tcio Command

The tcio Command

Syntax:

```
tcio -o device_for_cartridge_tape
tcio -i device_for_cartridge_tape
```

Examples (using cartridge device):

- Create an archive:

```
$ find . -print | cpio -ocvx | tcio -o /dev/rct/c2d0s2
```

- Restore an archive:

```
$ tcio -i /dev/rct/c1d0s2 | cpio -icudmvx
```

- Get table of contents:

```
$  tcio -i /dev/rct/c1d0s2 | cpio -ict
```

- Restore a single file:

```
$ tcio -i /dev/rct/c1d0s2 | cpio -icudm "filename"
```

- Restore all file names matching pattern:

```
$  tcio -i /dev/rct/c1d0s2 | cpio -icudm '*filename*'
```

H2572 11-5. 105 © 1990 Hewlett-Packard Company

Student Notes

Using redirection causes excess wear and tear on the tape drive because the data-transfer rates between the host computer and the cartridge tape drive are not in sync. The tcio command was written to *buffer* the data transfer between cpio and the cartridge tape drive. Instead of redirecting the output of cpio straight to the device, we pipe it through tcio to enable this streaming to take place.

The tcio command, like cpio, has two major options:

-o	Go out to a device.
-i	Come in from a device.

The -o and -i options to tcio correspond to those used with cpio.

11-6. LAB: Offline File Storage

Lab Objective

To practice using the tar and cpio commands to produce backups of user files.

Directions

Complete the following exercises. Record the commands you would use to perform the following tasks.

1. Using tar, create an archive of all files in your HOME directory that start with abc.

2. Using find, cpio, and tcio, make a backup of your whole directory structure from your HOME directory on down.

3. Remove the file backup from your current directory. Then restore the file from tape using the tcio and cpio commands

4. Obtain a table of contents listing of this tape archive.

5. Create the directory $HOME/tree.cp. Look up the *pass* mode of the cpio command in cpio(1). Using the cpio command in the pass mode, recreate the directory structure $HOME/tree under the directory $HOME/tree.cp.

Course Review

HP-UX BASICS II

As you complete this course, please indicate below whether you feel extremely confident (EC), confident (C), fairly sure (FS), not sure (NS), or unable (U) to perform the following.

1.	Logging in and out of HP-UX.	EC C FS NS U
2.	Using shell variables.	EC C FS NS U
3.	Navigating the file system tree.	EC C FS NS U
4.	Understanding permissions and ownership of files.	EC C FS NS U
5.	Using simple file manipulation commands such as cp, ln, and mv.	EC C FS NS U
6.	Understanding metacharacters for file name generation.	EC C FS NS U
7.	Understanding the uses of some of the many directories under HP-UX.	EC C FS NS U
8.	Understanding command execution.	EC C FS NS U
9.	Using vi.	EC C FS NS U
10.	Using the advanced features of the Korn shell.	EC C FS NS U
11.	Using input/output redirection.	EC C FS NS U
12.	Using pipelines.	EC C FS NS U
13.	Using quoting mechanisms to escape the meaning of special characters.	EC C FS NS U
14.	Running multiple processes at one time.	EC C FS NS U
15.	Writing nontrivial shell programs.	EC C FS NS U
16.	Making copies of files to tape.	EC C FS NS U

Appendix A — Additional Features of the Korn Shell

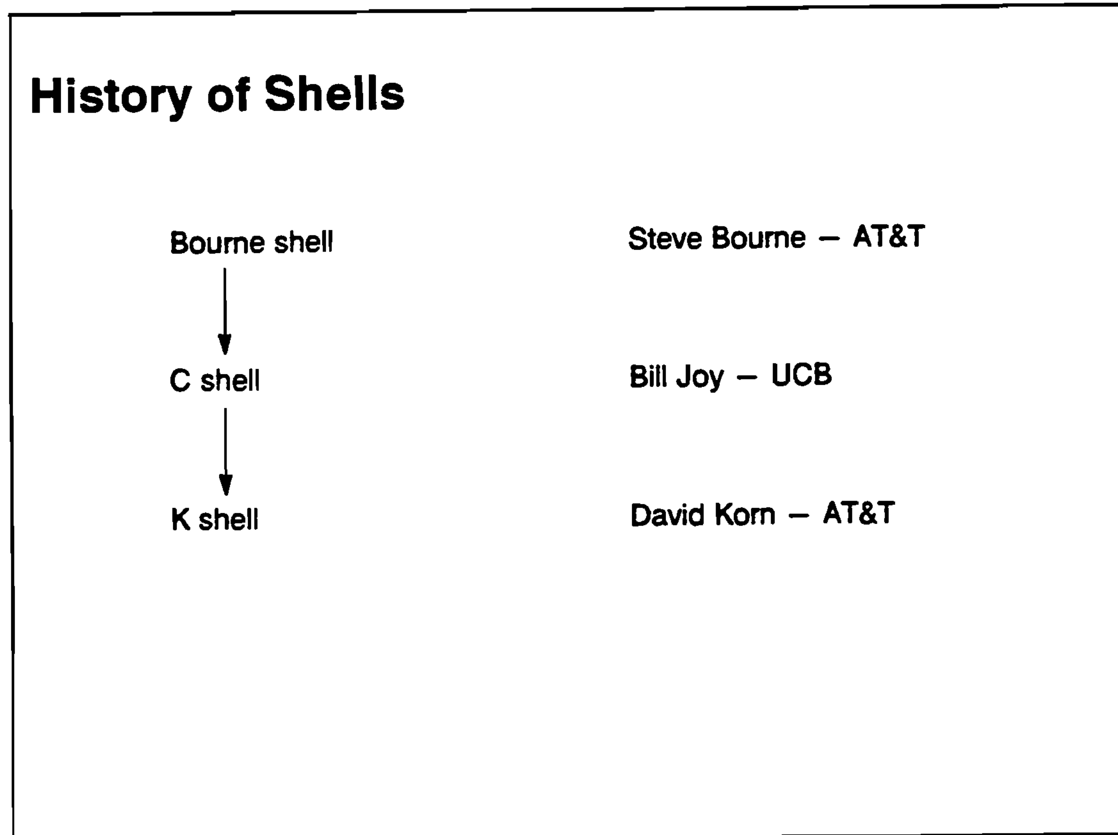
Objectives

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

- Set up aliases to save typing.
- Use the file name completion feature of the K shell.
- Recall and edit commands that have already been typed.

Appendix A — Additional Features of the Korn Shell

A-1. SLIDE: History of Shells



H2572 A-1. . 106 © 1990 Hewlett-Packard Company

Student Notes

The first shell developed for the UNIX operating system was called the Bourne shell after the man who led the team that wrote it, Steve Bourne.

The people at the University of California at Berkeley found the Bourne shell to be lacking in some key areas. For instance, they found that they had to do too much typing. Therefore, Bill Joy set about writing a new shell. He called his shell the C shell. The C stands for California.

The C shell offered new features that the Bourne shell did not have. First of all, the user could recall past commands and edit them. This editing process was tedious, but it worked. Secondly, the C shell offered the ability to set up different names for a command. These different names were called **aliases.** Thirdly, the C shell gave the user a mechanism for file name completion. That is, the shell would finish the file name that the user had started. Finally, the C shell offered a directory stack which very few people ever used. There were many other advantages to the C shell, but there were two major disadvantages: It was not standard, and it had trouble with Bourne shell programs.

Appendix A — Additional Features of the Korn Shell

Finally, David Korn of Bell Laboratories got it right! He wrote the Korn shell, which users call the K shell. It has the best features of the Bourne shell and C shell combined into one. It offers a command history, command editing, file name completion, and aliasing, while being perfectly compatible with the Bourne shell. In fact, the K shell is a superset of the Bourne shell.

This module discusses the best features of the K shell.

A-2. SLIDE: Features of the K Shell

Student Notes

The Korn shell (or K shell) has many advanced features. The features that we will discuss in this module are those that will make your usage of the HP-UX system more productive. Basically, the features that we will discuss are designed to decrease typing time and increase the accuracy of what is typed.

The four features that we will discuss are the following:

- Aliasing.
- Command-line editing.
- File name completion.
- Command history.

We will discuss each of these topics separately.

A-3. SLIDE: Aliasing

Aliasing

Syntax:

```
alias [new_word[='string']]
```

Examples:

```
$ alias go='cd /users/user3/tree/subdir1'
$ go
$ pwd
/users/user3/tree/subdir1

$ alias
date=/bin/date
false=let0
functions=typeset -f
h=history
hash=alias -t
history=fc -l
.
.
.
```

H2572 A-3. 108 © 1990 Hewlett-Packard Company

Student Notes

Aliasing is a method by which you can abbreviate long command lines, create new commands, or cause standard commands to perform differently by replacing the original command line with a new command called an alias. The new command can be a letter or short word when typed, but will expand to the complete command line when used. Aliasing can provide easier typing due to abbreviation of long command lines and automatic replacement of long path names.

The syntax is as follows:

```
alias [new_word[='string']]
```

For example:

```
alias go='cd /users/user3/courses/fundamentals/sources'
```

Now suppose you type:

```
go
pwd
CTRL-d
```

The result would be:

```
/users/user3/courses/fundamentals/sources
```

With no arguments, the alias command reports all aliases.

Aliases can be turned off with the unalias command. The syntax is the following:

unalias *aliasname*

Aliases can also be exported to child shells by putting them in a file (usually $HOME/.kshrc) and setting the variable ENV equal to the file name. ENV should be exported as follows:

```
ENV=$HOME/.kshrc
export ENV
```

A-4. SLIDE: Command-Line Editing

Command-Line Editing

```
$ more /users/user3/myfile [ESC]
```

Use vi commands to edit, then press [Return] to execute.

H2572 A-4. 109 © 1990 Hewlett-Packard Company

Student Notes

One of the problems that users had with the Bourne and C shells was that when they were typing a very long command, they might get almost to the end of it and realize that they had made a mistake near the beginning. The only way to correct the error was to erase all characters back to the mistake and retype the command. The K shell was written to help solve this problem.

While typing in a command, you can press [ESC] to put the command line in a one-line version of vi. Now, you can use standard vi commands to edit the command and change what you will. The command can then be executed by pressing [Return]. The choice of editor can be changed by setting the environment variable EDITOR.

There is an additional feature of command-line editing which, at first, may not seem obvious. Using the vi commands for moving around in a file, one can recall previous commands, edit, and execute them. This is a way to do what is known as command history. We will see another form of command history later in this module.

A-5. SLIDE: File Name Completion

File Name Completion

```
$ more ne [ESC] [ESC]
$ more new.jersey [Return]
    .
    .
    .
$ more abc [ESC] [ESC]
$ more abcdef [ESC] =
1)abcdefXlmnop
2)abcdefYlmnop
$ more abcdef
```

Edit with vi commands and press [ESC] [ESC] . Press [Return] to execute or press [ESC] again and continue typing.

H2572 A-5. 110 © 1990 Hewlett-Packard Company

Student Notes

The K shell has implemented many features from the C shell, not the least of which is file name completion.

Instead of typing the entire file name, you may type a part of the file name and allow the shell to complete it by typing [ESC] [ESC]. If the shell responds by "beeping," then it is unable to complete the file name based on what you gave it.

At this point, the shell has completed the file name as far as it can without a conflict. You may list the possible choices at this time by typing [ESC] =.

Using the command line editor, you can now complete the file name yourself or add sufficient characters so the shell can complete the file name. Type [ESC] [ESC] again. This process can continue until you are satisfied with the file name, and then press [Return].

The slide shows an example.

A-6. SLIDE: Command History – Part I

Command History — Part I

```
$ fc -l
7 more new.jersey
8 lsf
9 vi filename
.
.
.
$ fc -e - 8
lsf
dir1/ file1 file2 file3
$ fc -e - m
more new.jersey
.
.
.
```

H2572 A-6. 111 © 1990 Hewlett-Packard Company

Student Notes

A list of commands that have been previously typed can be obtained with the fc -l command. This command is used so often that there is a default alias for this command: history.

To reexecute a previous command, type:

`fc -e -` *X*

where *X* is either:

A number — indicating the number of the command to be executed.
A string — indicating a command that begins with that string.

For example, `fc -e - 217` will reexecute command number 217, and `fc -e - mo` will reexecute the last command starting with "mo".

The string fc -e - is used so often that there is a default alias, r, that can be used in its place.

Thus, `r mo` is the same as `fc -e - mo`, and `r 217` is the same as `fc -e - 217`.

A-7. SLIDE: Command History – Part II

Command History — Part II

$ [ESC] k [kkk . . .] Recalls previous commands.

Use vi commands (not arrow keys!) to edit the line, and press [Return] to execute.

H2572 A-7. 112 © 1990 Hewlett-Packard Company

Student Notes

Another way to reexecute previous commands is to enter the vi mode. This is done by pressing the [ESC] key. Then you can move "up" (toward older commands) by pressing the [k] key, and "down" (toward newer commands) with the [|] key.

Once the appropriate command is displayed, you can edit it using standard vi commands and execute it by pressing the [Return] key.

One note: Do not try to use the arrow keys for cursor movement because they will not work!

For this to work, the variable EDITOR must be set to "vi" in the shell's environment.

A-8. LAB: Additional Features of the Korn Shell

Lab Objective

To practice some of the more advanced features of the K shell.

Directions

Complete the following exercises and answer the associated questions.

1. Set up an alias called "go" to change your working directory to "tree" and do an ls. Now type the string go on the command line. What happens? Type pwd and see where you are. Now change back to your home directory.

2. Recall the alias command on your terminal and modify it (using vi commands) to reset the alias to "echo hello". Execute this new command by pressing [Return]. What happens? Then type go. What happens?

3. Get a list of the last several commands that you have typed with the fc -l command. Note that the alias "history" does the same thing. You can execute one of these previous commands with fc -e - *X*, where *X* is the number of the command or the string with which the command line started. Execute the alias command again. Then execute the go command. Note that r is the default alias to fc -e - .

4. We wish to perform the more command on a file, but we can only remember the first few letters of the file: "abc". Use the file name completion (ESC ESC) to complete the file name for you. What happens? How can you get a list of all possible files that match that pattern? Do it. Now add one additional letter and use ESC ESC again to complete the file name. Press Return to execute the command.

5. Recall the variable "whoson". Set up an alias called "whoson" to get a sorted list of just the user_ids of those presently on the system.