Advanced UNIX® (System V, Release 4) Usage Workshop

Student Guide

Unisys Release 1.0.1

March 1992

Printed in U S America
UE 7417

AL 3823

Advanced UNIX® (System V, Release 4) Usage Workshop

Student Guide

Unisys Release 1.0.1

March 1991

Printed in U S America

AL 3823

UE 7417

Contents

Course Description

Audience

Application users, programmers, and system support personnel

Prerequisites

AL 3822	Basic UNIX (System V, Release 4) Usage Workshop or a working knowledge of basic UNIX usage concepts and commands

Objective

Upon successful completion of this course, the student should be able to perform advanced user functions using the UNIX operating system.

Description

This course teaches the skills needed to use UNIX to perform advanced user functions. It also satisfies prerequisites for advanced UNIX cousses.

This instructor-led course includes a review of basic UNIX concepts and commands, section summaries, and practical exercises to supplement structured discussions. Hands-on activities are provided throughout this course.

Topics

- Input/output redirection
- Command execution management
- UNIX shells
- Shell programming fundamental

Duration

2 days

Agenda

Day 1

- **Module 1. Basic UNIX Review**

- **Module 2. Command Input/Output Redirection**
  - Standard input, standard output, and standard error
  - Redirection
    - Symbols
    - Redirect standard input
    - Redirect standard output
    - Redirect standard error

- **Module 3. Command Execution Management**
  - Process structure overview
  - Display process status
  - Execute multiple commands
  - Group commands for execution
  - Execute commands in background
  - Terminate processes
  - Prevent command termination
  - Execute commands later

Day 2

- **Module 4. Shells**
  - Review of UNIX shell functions
  - UNIX shell types
  - Korn shell overview
    - Features
    - Setup
    - Command history
    - Line edit mode
    - Command alias
    - Job control
    - Options and variables

- **Module 5. Shell Programming Fundamentals**
  - Shell programming overview
  - Create shell program
  - Check program for errors
  - Use variables
  - Modify user environment
  - Control flow of program execution

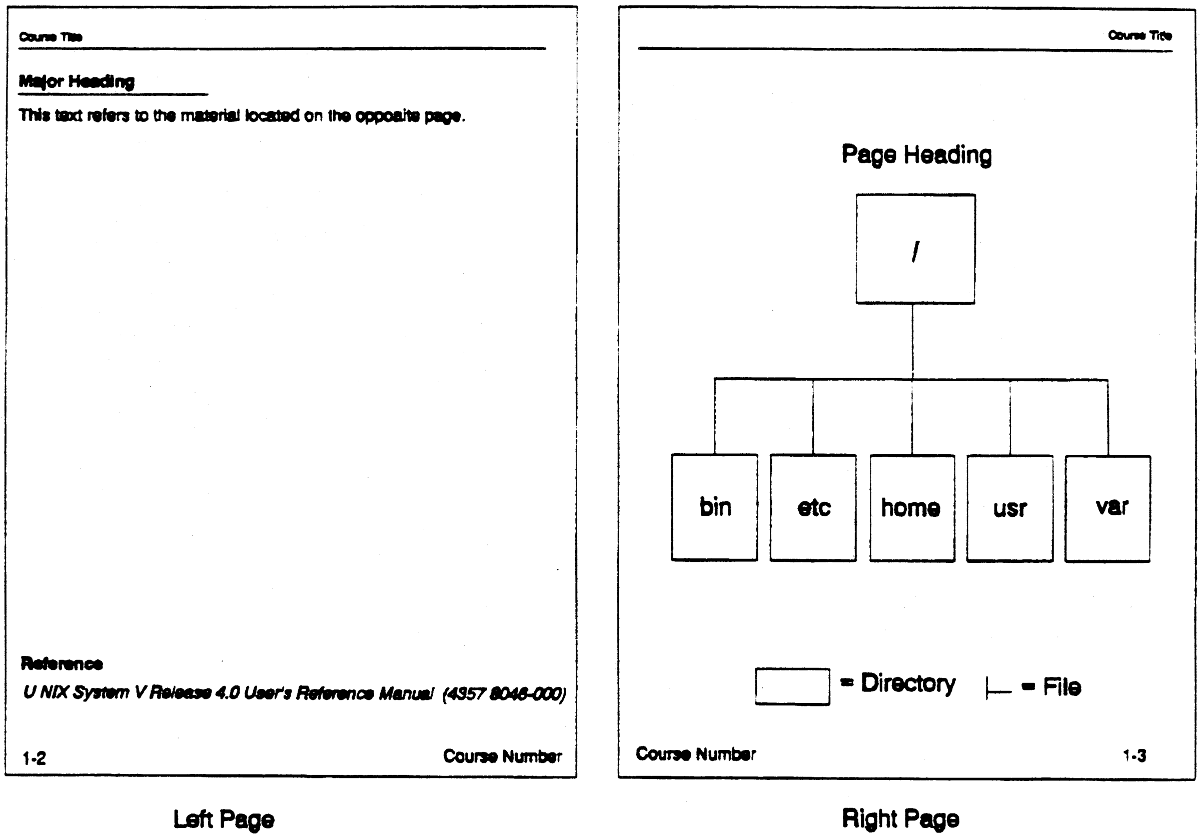
About This Course

Student Guide Organization

This course is directed towards individuals with diverse data processing experience. It presents additional fundamental UNIX principles and provides practical guidance to perform more advanced UNIX user functions. References to product documentation are provided to direct the student.

This student guide consists of five modules, or sections, and a command summary in the appendix. Each module begins with a set of learning objectives providing structure and purpose to the body of information contained within the module. A module summary and written and/or practical exercises are provided in each module. Longer modules contain multiple, staggered practical exercises providing hands-on activities on selected topics in addition to the module summary and the final module exercise.

The format for module pages in this student guide is shown below. The left page contains descriptive text pertaining to the information appearing on the right page. Product information documentation is listed at the bottom of the left page. The right page generally contains text, tables, or graphics.


Course Title

Major Heading

This text refers to the material located on the opposite page.

Reference

UNIX System V Release 4.0 User's Reference Manual (4357 8048-000)

1-2 Course Number

Figure 1. Page Layout

Typographic Conventions

The following conventions are used throughout this document.

Convention	Represents
Bold	Command name Command line entry
Italics	File name Directory name Variable information Documentation title
`Courier`	Terminal entry . Terminal display
< >	Input that does not appear on the screen when typed, such as TAB, ESCAPE, and RETURN keys
<^char>	Control characters that do not appear on the screen when typed. The circumflex (^) represents the control key, usually labeled Ctrl. To type a control character, hold down the control key while pressing the specified character. For example, to enter <^d>, hold down the control key while pressing the letter **d** key – the letter **d** does not appear on the screen.
[ ]	Command options and arguments considered optional are enclosed in brackets. Brackets should not be entered as part of the command line.

Unisys Reference Documentation

Abundant information on UNIX is available in Unisys publications and in commercial textbooks. It is important to be familiar with the available resources in order to locate desired information quickly and efficiently.

This section presents an orientation to Unisys reference documentation. Descriptions of key standard reference documentation are provided below.

• *Release Notes*	Describe the capabilities and features of the 4.0 UNIX operating system
• *Software Installation and Operations Guide*	Contains operating system and software product installation procedures, and basic startup and maintenance procedures
• *Administrator's Guide*	Describes system administration and maintenance functions such as system setup and configuration, user and device management, file system administration, backup and restore procedures, print service administration, and system security functions
• *Administrator's Reference Manual(s)*	Contains descriptions of administrative commands, file formats, and special files
• *Error Message Manual*	Provides assistance in interpreting error messages and determining probable causes and solutions for hardware and software problems; contains an index of alphabetic error messages to facilitate quick access
• *Programmers's Guide*	Presents an overview of the UNIX system programming environment and tutorials on various programming tools
• *Programmer's Reference Manual(s)*	Describes programming commands, system calls, library routines, formats, and miscellaneous utilities
• *User's Guide*	Presents an overview of the UNIX operating system and tutorials on user-related topics like text editing, print services, electronic mail, and network communication
• *User's Reference Manual*	Describes user commands
• *Network User's and Administrator's Guide*	Directed to users of remote services and to the system administrator setting up and maintaining file sharing capability

Of the documents listed above, the organization of the reference manuals containing command descriptions requires further examination.

Unisys Reference Documentation

Document Title	Number
Release Notes	4357 7592-000
UNIX System V Release 4 Installation and Operation Guide	3915 2483-000
UNIX System V Release 4 Administrator's Guide	3915 2442-000
UNIX System V Release 4 Network User's and Administrator's Guide	3915 2467-000
UNIX System V Release 4 Error Message Manual	3915 2624-000
UNIX System V Release 4 X Window Access Operation Guide	7431 1473-000
Commescial Secure: Security Features User's Guide	7431 2810-000
Commercial Secure: Trusted Facility Tuning Manual	7431 2802-000
UNIX System V Release 4 Tuning Guide	3915 2525-000
UNIX System V Release 4 Administrator's Reference Manual	V1 - 4357 7451-000 V2 - 4357 7469-000
UNIX System V Release 4 User's Reference Manual	4357 7444-000
UNIX System V Release 4 User's Reference Manual Supplement	3915 2962-000
UNIX System V Release 4 User's Guide	3914 9398-000
UNIX System V Release 4 Programmer's Guide Supplement	3915 2921-000
UNIX System V Release 4 Programmer's Guide: Support Services and Application Tools	3914 9463-000
UNIX System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools	3914 9414-000
UNIX System V Release 4 Programmer's Guide: Character User Interface: FMLI, ETI	3914 9406-000
UNIX System V Release 4 Programmes's Guide: STREAMS	3915 2608-000
UNIX System V Release 4 International Enhancements Guide	3915 2566-000
UNIX System V Release 4 Programmer's Guide: Networking Interfaces	3914 9422-100
UNIX System V Release 4 Device Driver Programmer's Guide	3915 2988-000
UNIX System V Release 4 BSD/XENIX Compatibility Guide	3915 2574-000
UNIX System V Release 4 ANSI C Transition Guide	3915 2590-000

Unisys Reference Manual Organization

Traditionally, UNIX reference manuals containing command descriptions adhere to the organization used by AT&T, the developer of UNIX. The AT&T organizational model consists of eight standard sections described below.

Section 1	Contains alphabetic descriptions of commands and utilities for administrators, users, and programmers
Section 2	Describes the system calls used to interact with the kernel
Section 3	Describes the library of subroutines available for programmers
Section 4	Describes main system files
Section 5	Contains various information and miscellaneous facilities, like character set tables and macro packages
Section 6	Games, not included in Unisys documentation
Section 7	Describes system special files, like device files
Section 8	System maintenance procedures, not included in Unisys documentation

Unisys has grouped the *Section 1* commands for users, administrators, and programmers into separate volumes containing commands and file format references extracted from the standard AT&T documents and Unisys customized commands.

Each set of reference manuals contains a table of contents listing the command entries in alphabetic order by section where multiple sections exists in the document, a permuted index to locate a command by topic followed by command entry descriptions.

Each command name appears in the format *command (AT&T_section_number)*, for example the **date**(1) command. The section number may also include a letter designating a grouping of like commands. For example, **C** refers to communication, **M** refers to maintenance, and **G** refers to graphics.

Unisys Reference Manual Organization

Unisys Manual	Extracted AT&T Section		Example
User's Reference Manual	1	User commands commands and utilities	**mandex(1)** **passwd(1)** **uucp(C)**
Administrator's Reference Manual	1	Administrator commands and utilities	**boot(1M)** **fsck(1M)** **mkfs(1M)**
	4	File formats	**inittab(4)** **passwd(4)** **profile(4)**
	5	Miscellaneous facilities	**regexp(5)** **signal(5)** **term(5)**
	7	Special files	**filesystem(7)** **termio(7)**
Programmer's Reference Manual(s)	1	Programmer commands and utilities	**cc(1)** **lint(1)** **sdb(1)**
	2	System calls	**sysfs(2)** **symlink(2)**
	3	Subroutines	**sleep(3C)** **basename(3G)** **trig(3M)**
	4	File formats	**a.out(4)** **core(4)**
	5	Miscellaneous facilities	**ascii(5)** **math(5)**

Reference Manual Entries

Each manual entry is formatted to include an appropriate subset of the following headings.

NAME	Entry name and function; may include related (secondary) entries
SYNOPSIS	Format of command, system call or library routine
DESCRIPTION	Overview of command usage or topic
EXAMPLES	Examples of command syntax or usage, where appropsiate
FILES	File names referenced by the command
EXIT STATUS	Value(s) set when command terminates
RETURN VALUES	Value(s) returned during command execution
NOTES	Helpful information or special considerations
SEE ALSO	Pointers to related information
DIAGNOSTICS	Diagnostic message interpretations
WARNINGS	Potential misuse, restrictions, limitations, or boundaries
BUGS or RESTRICTIONS	Known faults, deficiencies, or limitations.

Two sample manual pages are illustrated on the next page. The corresponding software utility package, Essential Utilities in this case, appears across from the command entry. Notice that the headings are not identical for each command description.

Reference Manual Entries

pwd

```
pwd(1)                                         Essential Utilities


NAME
     pwd - working directory name

SYNOPSIS
     pwd

DESCRIPTION
     pwd prints the path name of the working (current) directory.

SEE ALSO
     cd(1).

DIAGNOSTICS
     Cannot open .. and Read error in .. indicate possible file
     system trouble and should be referred to a UNIX system
     administrator.

NOTES
     If you move the current directory or one above it, pwd may
     not give the correct response.  Use the cd(1) command with a
     full path name to correct this situation.
```

cd

```
cd(1)                                          Essential Utilities


NAME
     cd - change working directory

SYNOPSIS
     cd [directory]

DESCRIPTION
     The cd command changes to a new working directory.  If
     directory is not specified, the value of shell parameter
     $HOME is used as the new working directory.  If directory
     specifies a complete path starting with /, ., or ..,
     directory becomes the new working directory.  If neither
     case applies, cd tries to find the designated directory
     relative to one of the paths specified by the $CDPATH shell
     variable.  $CDPATH has the same syntax as, and similar
     semantics to, the $PATH shell variable.  cd must have
     execute (search) permission in directory.

     Because a new process is created to execute each command, cd
     would be ineffective if it were written as a normal command;
          therefore, it is recognized by and is internal to the shell.

   SEE ALSO
          pwd(1), sh(1).
          chdir(2) in the Programmer's Reference Manual.
```

Online Reference Commands

The following UNIX commands provide online access to reference manual information.

man

This command accesses reference manual entries on the system. The information displayed is the same as the printed reference manuals described previously.

mandex

This command provides access to a menu-driven indexing system to search selected online manuals for a subject or command.

Online Reference Commands

- **man**

  Online reference manual

- **mandex**

  Menu-driven indexing system to online manuals

1

Basic UNIX Review

Module Objectives

The purpose of this module is to review fundamental UNIX terms and usage.

Reference

Documentation referenced in this module

- *UNIX System V Release 4 User's Guide* (3914 9398-000)
- *UNIX System V Release 4 User's Reference Manual* (4357 7444-000)
- *Basic UNIX (System V, Release 4) Usage Workshop* (UE-7415)

The following questionnaire consists of three sections (general, text editors, and commands).

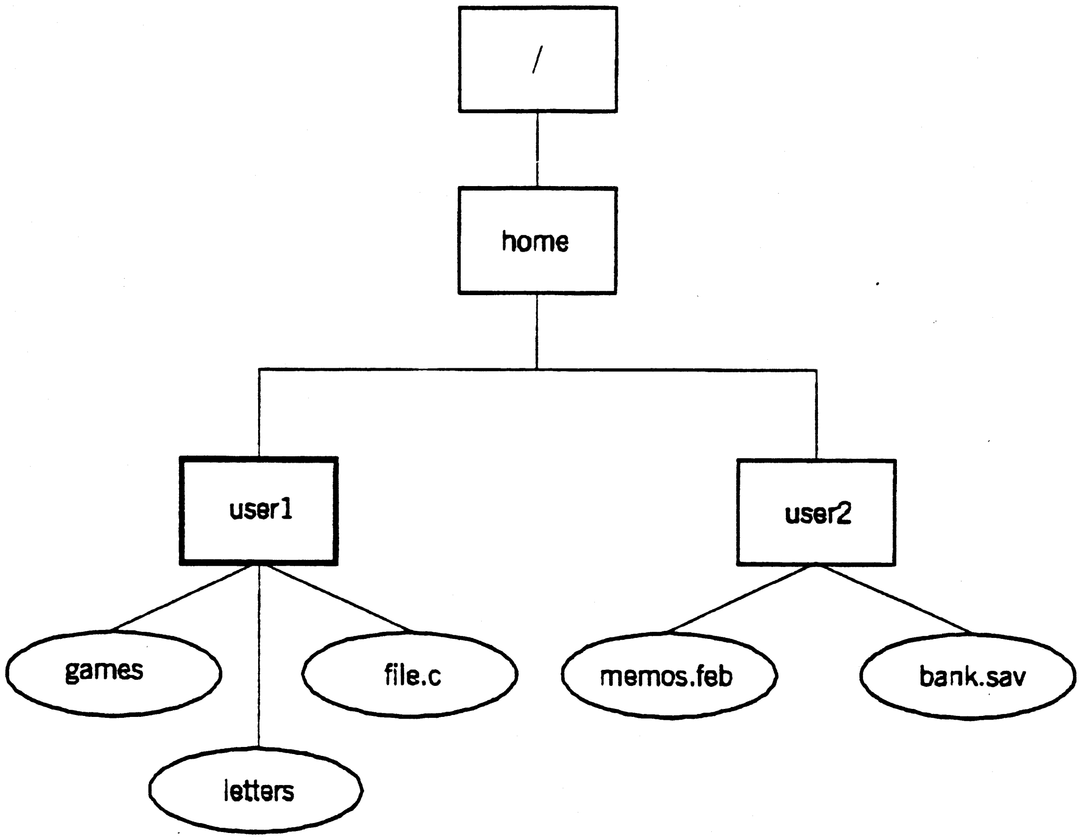
Complete the questionnaire according to the instructions provided by yous instructor. The questions will be reviewed in order to check and/or correct your understanding of basic UNIX terms and usage. You will be provided opportunity to ask questions on these and related topics.

General

1. Name the three layers of the UNIX operating system. Briefly describe each.

2. Describe the types of UNIX files.

3. Use the diagram below to answer the following questions pertaining to UNIX file hierarchy. Assume the current directory is **/home/user1** for all items.



14401-6

a. Name the full path name to user1's home directory.

b. Name the relative path name to user1's *file.c* file.

c. Name the reference to user1's parent directory.

d. Name the full path name to user2's *memos.feb* file.

e. Name the relative path name to user2's *memos.feb* file.

4. Describe the components of the UNIX command format.

5. Write the command to move to the parent directory.

6. List several file-naming conventions.

7. Name and describe three special characters used in file name expansion by the shell.

8. How can special characters be used literally?

Text Editors

1. Name and describe two UNIX text editors.

2. Distinguish between command and text input modes.

3. List the basic steps to create a file in ed, enter text, save text and exit the editor. Contrast with the steps to create a file through **vi**.

4. Write the **ed** command to display lines seven through nine with line numbers.

5. Describe three text input commands used in **vi**.

6. The command to undo the last change is ________.

7. Describe a method to copy text in **vi**. To move text.

8. Name two **vi** commands to delete text in command mode.

9. Which symbol indicates a forward search for a character string? A backward search?

10. Write the **vi** command to substitute all occurrences of *unix* with *UNIX* throughout a file.

11. Name the command that displays the file name with which the editing buffer is currently associated.

12. Write the command used in **vi** to replace the contents of the editing buffer with another file.

13. Which command reads the contents of another file into the current editing buffer?

14. Name the command to discard changes in the editing buffer without exiting the editor.

15. The option to invoke **vi** in read-only mode is ____. The command to override this option and save the contents of the editing buffer is _____.

16. Write the command that temporarily exits the editor to execute the shell command that will display your current directory.

Commands

1. Write the commands that perform the following functions:

   a. Assign or change a login password ________

   b. Print your working directory ________

   c. List the contents of a directory ________

   d. Display the contents of a file ________

   e. Determine who is logged into the system ________

   f. Display the current system date and time ________

   g. Get online assistance ________

   h. Copy a file ________

   i. Move or rename a file ________

   j. Create links to a file ________

   k. Create a directory ________

   l. Remove a directory ________

   m. Format a file for printing ________

   n. Print a file ________

   o. Display the current terminal settings ________

2. Which command searches selected online reference manuals and displays the pages containing the search string?

3. Name the command that prints status information for all printers on the system.

4. Which command is used to remove a print request?

5. Printer **ptr1** was disabled due to a paper jam which has been corrected. Write the command to restart printer **ptr1**.

6. Write the command that displays your user/group name and id.

7. Which command classifies files by their contents (for example, ASCII, executable, empty)?

8. Write the command to find your files that have not been accessed within the last 60 days and remove them.

9. Write the command to change access permission to the *salaries* file so that the owner can view the file and all others are completely restricted from accessing the file.

10. Write the command that sorts *fileX* numerically on the third field only and saves the sorted output to *fileX.srt*.

11. Write the command that searches all files for the pattern *UNIX* at the beginning of a line and displays the file names containing the pattern once (if multiple matches occur in the same file).

12. Write the command that changes the default permissions for new files and new directories to read and write access for the owner and read only for all other users.

13. Describe three UNIX commands that allow users to communicate with one another on the same system.

14. Which command displays information concerning a terminal's ability to receive messages from another terminal?

15. Distinguish between the **mail** and **mailx** utilities. What are tilde escape commands (**mailx**)? How are they used?

2

Command Input/Output Redirection

Module Objectives

Upon completion of this module, you should be able to redirect the flow of command input and output.

The supporting module objectives include the ability to

1. Redirect standard input from a file.
2. Redirect standard output to a file.
3. Redirect standard output to another command.
4. Redirect standard output to multiple destinations.
5. Redirect standard error to a file.

Reference

Documentation referenced in this module

- *UNIX System V Release 4 System User's Reference Manual* (4357 7444-000)

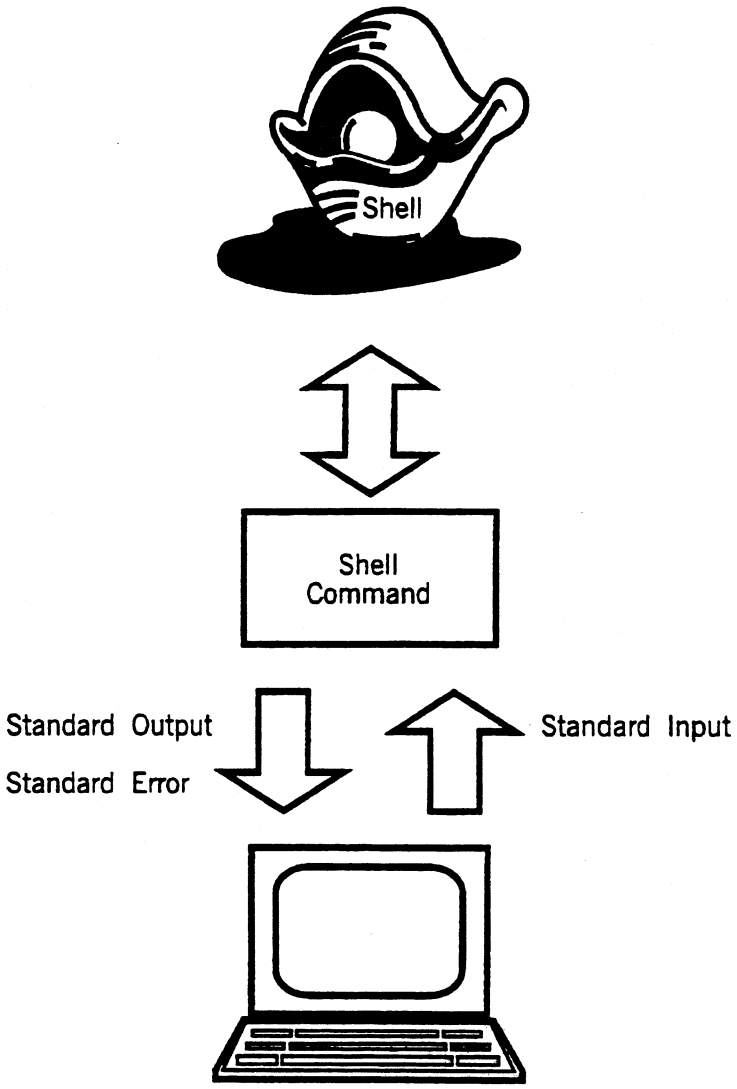
Standard Input, Standard Output, and Standard Error

Most commands accept input from a source called *standard input*. By default, this source is the terminal screen, unless otherwise designated. Similarly, a command sends its output to a destination called *standard output*, the terminal screen by default. Error messages generated by a command are directed to *standard error*, again the terminal screen by default.

The UNIX shell directs the source and destination of a command program's standard input, standard output, and standard error. This is illustrated in the next figure. In effect, the command program is "unaware" whether the standard input is actually from the terminal keyboard or from a file. Likewise, the command program is unaware whether the standard output is directed to the terminal screen, to a file, to another command, or to multiple destinations.

The shell can be instructed to *redirect* a command program's standard input, standard output, and standard error. This capability illustrates the shell's utility and flexibility.

Standard Input, Standard Output, and Standard Error



14401-1

Standard Input and Standard Output

The **cat** command illustrates how the terminal is used as standard input and standard output. **cat** accepts input from a file name (argument) designated on the command line and copies the file's contents to its standard output, directed by the shell to be the terminal screen. When a file name is not specified, **cat** takes input from standard input, the terminal keyboard.

Examples

In the first example, the **cat** command is executed without supplying a file name. After the <RETURN> key is pressed, the cursor (indicated by the filled square below the $ prompt) is stationary and no other activity is taking place. The shell is waiting for input from the terminal to pass to the **cat** command.

In the second example, each line of text that is entered and followed by a <RETURN> is passed to **cat**. This input is then copied by **cat** to standard output – back to the terminal screen – until <^d> is pressed to indicate the end of the file. **cat** finishes execution and returns control to the shell which displays another command prompt. The input is not stored (saved) since a destination was not specified other than standard out.

Standard Input and Standard Output

Example 1

```
$ cat<RETURN>
█
```

Example 2

```
$cat
Entering a line of text.
Entering a line of text.
cat continues to copy text
cat continues to copy text
until Ctrl-d is pressed
until Ctrl-d is pressed
on a line by itself.
on a line by itself.
<^d>
$
```

Redirection Symbols

The term *redirection* refers to the various ways the shell alters the source of standard input and the destination for standard output and standard error.

The special characters used in a command line to instruct the shell to redirect a program's input, output, or error are listed on the following page and described below. The shell interprets these characters before the command is executed.

Input Redirection

The < symbol instructs the shell to redirect the input to a program from the named file.

Output Redirection

Output redirection diverts the output from a command to a destination other than the terminal screen. A program's output can be directed to a file, to another command, or to both files and commands.

The > symbol instructs the shell to redirect the output of a program to the named file.

The >> symbols cause standard output from the command to be appended to the named file.

The | symbol allows two or more commands to be connected together. This is called a *pipe* because the output of one command is piped as input to the next command.

The **tee** command "splits" the output of a command and redirects it to multiple destinations, for example, to the named file and to the next command in the pipe.

Each form of redirection is presented separately.

Redirection Symbols

Redirection	Designation	Interpretation
Input	**<**	Redirects input from named file
Output	**>**	Redirects output to named file
	>>	Appends output to named file
	|	Redirects output to named command
	tee (command)	Redirects output to multiple destinations

Input Redirection

The redirect input symbol, <, tells the shell to take input for a command from a file instead of the terminal keyboard. Input redirection can be used with any command that accepts standard input from the terminal.

The format for a command line using input redirection is shown at the right. Spaces before and after the < symbol are not required, although they are generally used by convention.

Another form of input redirection is *in-line input redirection*, also referred to as a *here document*. It is useful in shell programs to specify standard input to a command directly in the program without creating a separate file first. The <<*label* designation, consisting of one or more characters following the << symbols, tells the shell to use the lines that follow up to the next *label* designation (at the beginning of a line) as input to the command. This is illustrated later in the shell programming module.

Examples

In the first example, the shell is instructed to take input for the **cat** command from the file called *test*. The **cat** command then displays this input on the standard output, the terminal screen. In this case the input redirection yields the same result as if the **cat** command were executed without the < symbol. Many commands, like **cat**, are already designed to accept input from a file.

The **mail** command allows a user to send or receive electronic mail. Often the message sent to a user is entered as standard input at the keyboard. In the second example, however, the message is already stored in a file called *reminder* which is redirected as input to the **mail** command.

Generally, input redirection is not used as frequently as output redirection.

Input Redirection

command *[-options] [arguments]* < *input_file*

<<label

Example 1

```
$ cat test
This is a sample text file.
$ cat < test
This is a sample text file.
```

Example 2

```
$ mail user1 < reminder
```

Redirecting Output to a File

The > symbol indicates output redirection to a file. This can be used with any command that provides standard output. The command structure including output redirection is depicted on the next page. Again, spaces around the > symbol are not significant, but generally they are used.

Examples

In Example 1, standard output is redirected to *text* file. Since an input file is not specified, the shell directs the **cat** command to accept input from the keyboard until a <^d> is encountered indicating the end of the file. Notice the text is not echoed (repeated) back to the screen when the <RETURN> key is pressed because it is diverted to *text* file. This use of the **cat** command provides a quick way to create short files without using a text editor. However, it does not allow corrections easily.

If the *text* file already exists containing data, as in Example 2, it is overwritten. The Bourne shell erases the contents of the existing file to prepare for the output of the **cat** command. To avoid inadvertently losing the contents of a file, use the >> symbols to append (add) the output to the end of the named file. This is illustrated in Example 3.

An existing file may also be overwritten by the Bourne shell, if the same file name is used for both input and output files. In the example **cat** *names morenames > names*, the *names* file would only contain the contents of the *morenames* file. The shell first empties the file name to the right of the redirection symbol. Since *names* file is now empty, the only input passed to the **cat** command is the contents of the *morenames* file. To avoid accidental loss of data when using redirection, use different input and output file names.

Note: The Korn shell's *noclobber* variable prevents accidentally overwriting a file when output redirection is used. It also prevents creating a file when output is appended to a nonexistent file.

Redirecting Output to a File

command *[-options] [arguments] > output_file*

>> output_file

Example 1

```
$ cat > text
Text is not echoed to the screen because
it is redirected to the named file.
<^d>
$ cat text
Text is not echoed to the screen because
it is redirected to the named file.
```

Example 2

```
$ cat > text
This text replaces the original text.
<^d>
$ cat text
This text replaces the original text.
```

Example 3

```
$ cat >> text
Adding more text to the file.
<^d>
$ cat  text
This text replaces the original text.
Adding more text to the file.
```

Redirecting Output to a Command

The UNIX pipe is a method to connect two or more commands. Although the same results can be achieved using output redirection to a file, this method is more efficient because it avoids the user's having to create extra files. The pipe symbol, | (vertical bar), indicates output redirection to another command. Any command that accepts standard input and produces standard output can be used in a pipeline.

The format for a pipeline is illustrated on the next page. Each additional command in the pipeline is preceded by the | symbol.

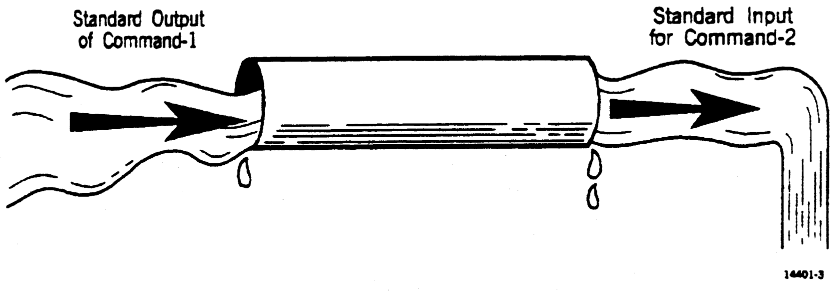
Examples

The first method in Example 1 uses file redirection to count the current number of users on the system. The output of the **who** command is saved to a file, *users.tmp*. Next, the **wc -l** command is used to count the number of lines in the *users.tmp* file. Finally, the temporary file is removed since the information it contains is variable. The second method shows how these three steps can be combined into one using a pipeline construction. The output of the **who** command is not displayed; it is directed by the shell as input to the next command, **wc -l** in this case. The terminal displays only the final output, that is, the number of current users.

Example 2, shows a more complex structure to combine the contents of two files into one using **cat**, to sort it, and to print the sorted version. In this pipeline construction, the sorted output is not displayed, nor is it captured in a file. This can be done, however, using the **tee** command, presented subsequently.

Some commands, like **wc** and **sort**, transform or change the input in some way and output the altered data. These commands are called *filters*; they are frequently used in pipelines.

Redirecting Output to a Command



command1 | command2 | command3 . . .

Example 1

File Redirection Method:

```
$ who > users.tmp
$ wc -l < users.tmp
$ rm users.tmp
```

Pipeline Method:

```
$ who | wc -l
```

Example 2

File Redirection Method:

```
$ cat names morenames > allnames
$ sort allnames > allnames.srt
$ lp allnames.srt
```

Pipeline Method:

```
$ cat names morenames | sort | lp
```

Redirecting Output to Multiple Destinations

The **tee** command captures the output of a command in a pipe by copying standard input to standard output, to another command in a pipeline, or to one or more files. This is illustrated on the next page.

The format of the **tee** command appears on the next page. The **tee** command overwrites the output file(s) if it exists, unless the **-a** option is used to append output to the named file(s).

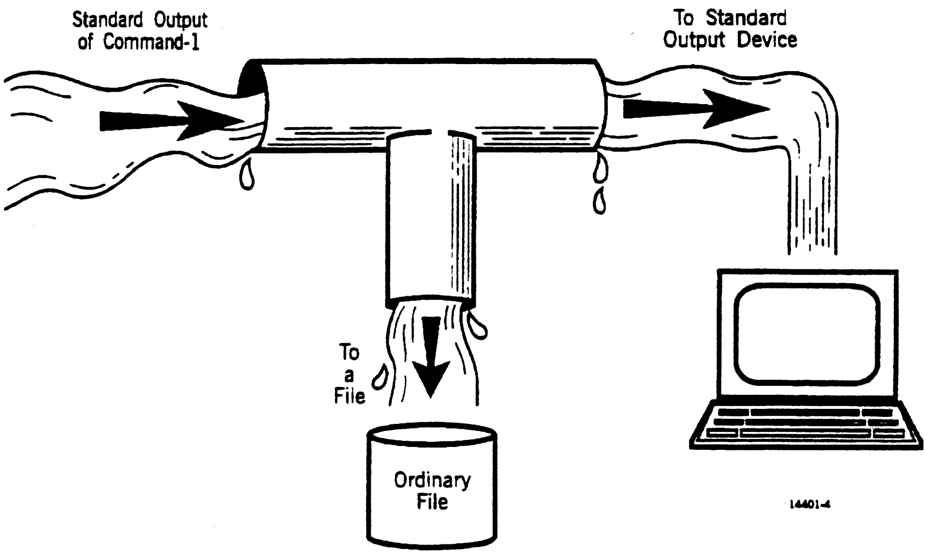
Examples

In Example 1, the *report1* file is formatted using the **pr** command. The formatted output is copied to the *report1.pr* file and to standard output, the screen since no other command is included in this pipeline.

The **tee -a** command prevents accidental overwriting of the existing file. The formatted output of the *report2* file, in Example 2, is appended to the *report.pr1* file. The output is not displayed as in the previous example because it is piped to the **lp** command for printing.

Example 3 copies the formatted output of the *inventory* file to the *inventory.pr* file, to the terminal (more correctly, to the device file */dev/term/14* representing the terminal), and pipes the formatted output to the default system printer.

Redirecting Output to Multiple Destinations



command *[-options] [arguments]* | **tee** *[-a] file(s)*

Example 1

```
$ pr report1 | tee report1.pr
```

Example 2

```
$ pr report2 | tee -a report1.pr | lp
```

Example 3

```
$ pr inventory | tee -a inventory.pr  /dev/term/14 | lp
```

Redirecting Standard Error

The error message produced by a command is normally directed by the shell to standard error, which is the same destination as standard output – the terminal. This can also be redirected to a file using the > symbol. Since this symbol is also used to redirect standard output, further distinction is required to avoid ambiguity.

The following file descriptors specify standard input, standard output, and standard error explicitly.

0	standard input
1	standard output
2	standard error

The file descriptor immediately precedes the redirection symbols. For example, *1>* refers to standard output, while *2>* refers to standard error. The **prgm** *2> errfile* command instructs the shell to direct any standard error to the file *errfile*. The explicit designations for standard input (*0<*) and standard output (*1>*) are required only to avoid ambiguity.

Examples

In Example 1, three files, one of which does not exist, are concatenated (serially appended). The *combo* file contains the contents of *file1* and *file2*. Since the error message was not redirected, it is displayed as standard output on the screen.

Example 2 illustrates standard error redirection using file descriptors to explicitly distinguish between standard output and standard error. The *combo* file contains the contents of *file1* and *file2*. The expected error message is not displayed. Instead, it has been captured and diverted to the file *oops*. To append standard error to an existing file, use the designation *2>> errfile*, where errfile is the name of the original file containing standard error.

Example 3 shows how to redirect standard error to the same file specified for standard output. The *1>* redirects standard output to the *combo* file. The notation *2>&1* declares file descriptor 2 to be a duplicate of file descriptor 1, thus redirecting standard error to the same destination as standard output. Again, the expected error message is not displayed since it has been redirected to the *combo* file.

Redirecting Standard Error

command *[-options] [arguments] 2> file*

Example 1

```
$ cat file1 file2 nofile > combo
cat:  cannot open nofile
$ cat  combo
This is file 1.
This is file 2.
```

Example 2

```
$ cat file1 file2 nofile  1> combo 2> oops
$ cat combo
This is file 1.
This is file 2.
$ cat oops
cat:  cannot open nofile
```

Example 3

```
$ cat file1 file2 nofile  1> combo 2>&1
$ cat combo
This is file 1.
This is file 2.
cat:  cannot open nofile
```

Summary

- *Standard input* is the input or information used by a command. By default, this is the terminal keyboard.
- *Standard output* is the output or result of a command. By default, this is also the terminal screen.
- Error messages generated by a command are directed to *standard error*, again, the terminal screen by default.
- The shell directs a program's standard input, standard output, and standard error.
- The shell can be instructed to *redirect* a program's standard input, standard output, and/or standard error.
- The *redirect input symbol (<)* instructs the shell to take input for a command from a file instead of the terminal keyboard.
- The *redirect output symbol (>)* tells the shell to redirect the standard output of a command from the terminal to a file.
- The *append output symbol (>>)* instructs the shell to add new information to the end of a file. Appending to output prevents accidentally overwriting data.
- A *pipe (|)* connects commands together so that the standard output of one command becomes the standard input to the next command in the pipeline.
- A *filter* is a command that takes its input from standard input, transforms it in some way, and produces its result on the standard output.
- The *tee* command splits the direction of the standard output and redirects it to multiple destinations. The -a option appends the modified data to the named file preventing accidental overwriting of data.
- *File descriptors* are used to explicitly distinguish between standard input, standard output, and standard error. By convention, 0 refers to standard input, 1 refers to standard output, and 2 refers to standard error.

Practical Exercise

1. Create a file named *format* using the **cat** command and output redirection. Enter the following text in the file:

```
I am creating this file using the cat command
and the redirect output symbol.  This text will
be stored in the format file.
```

2. Append the following text to the file named *format*:

```
This text will be added to the format file.
```

3. Using the editor of your choice, create two files in your home directory called *students1* and *students2* with the text provided below. These files contain the following student names:

```
students1                students2

Zollow, George           Adams, Mary
Brady, Mike              Green, Bob
Christensen, Alice       Thompson, Dave
Brenton, Rick
```

Write the command lines used to perform the following steps in the space provided below each item.

a. Use the **cat** command to concatenate the *students1* and *students2* files. Redirect the output to a file called *students.all*.

b. Remove the *students.all* file, and **cat** the files again, this time misspell the *students1* file name. What happened? What is the content of *students.all*?

c. Repeat Step b above, again misspelling the *students1* file. This time, redirect error messages to a file name *students.err*. What does each file contain?

d. Concatenate the *students1* and *students2* files, save the results in a file called *students.temp*, and simultaneously display the results on the terminal and send the output to the printer.

Optional Exercise

1. Define standard input, standard output, and standard error.

2. True or false. Redirection is performed by the command program. If false, explain.

   a. True

   b. False

3. Match the descriptions in column B with the corresponding redirection symbol in column A.

Column A		Column B
___ >	a.	Appends to output
___ <	b.	Redirects input
___ >>	c.	Appends to input
	d.	Redirects output

4. Select the answer(s) that complete(s) the following statement correctly.

   A filter is a command that

   a. Redirects input to a program

   b. Redirects output of a program

   c. Alters standard input

   d. Is used in UNIX pipelines

5. Write the designation that redirects standard error to the same file as the standard output.

3

Command Execution Management

Module Objectives

Upon completion of this module, you should be able to manage command execution.

The supporting module objectives include the ability to

1. Determine command process status.
2. Execute multiple commands in one command line.
3. Group commands for execution.
4. Execute commands in background.
5. Terminate processes.
6. Run command ignoring hangups and quits.
7. Schedule command execution for a later time.
8. List and remove jobs from schedule queue.

Reference

Documentation referenced in this module

- *UNIX System V Release 4 System User's Reference Manual* (4357 7444-000)

Processes

A *process* is the execution of a program. When a command line is executed, a process is initiated.

Process Structure

Like the UNIX file system, the organization of processes is hierarchical. It has parents and children, even a root. A *parent* process creates a *child* process, which can also create other processes. The term *spawn* is used to refer to the creation of processes. The first process started when the system is booted is *init*. Like the superuser root, init is the *grandparent* of all processes.

Command Execution

When a command line is executed, the shell usually spawns a child process to execute the command. While a child process is executing, the parent process is in an inactive state called *sleep*. A sleeping process does not use any computer time. When the child process completes execution, it dies. The parent process (running the shell) awakens and issues another command prompt.

Process Identification

When a process is spawned, UNIX assigns it a unique number. This number is called the *process identification,* also referred to as *PID*. A process keeps the same PID number as long as it exists. For example, during a given session, the same PID is associated with the login shell. The process identification of the *init* process is always PID 1.

Some commands display the PID as part of the output. The **ps** command provides status information about processes. The **kill** command terminates processes.

Processes

- **A program during execution**
- **Organized in a hierarchy**
  - Grandparent of all processes is *init*
  - Parent process
  - Child process
- **Identified by a unique number called *PID***
  - *init* has PID 1
  - PID remains the same while the process exists
- **Display status of processes**
  - **ps** command
- **Terminate processes**
  - **kill** command

Displaying Process Status

ps – Display information about active processes

Description

The **ps** command displays information about active processes according to the designated options. The column headings displayed depend on the options specified. For example, the **-f** option displays eight columns of information; the **-l** option displays 15 columns. Column!descriptions are provided below. Multiple options can be combined. Some!options accept lists of arguments.
If options are not specified, information is displayed about processes running under the user's ID and associated with the current terminal only. The output, in this case, contains the process ID, terminal identifier, cumulative execution time, and command name.

Options

-a	Prints information about processes owned by *others*
-f	Generates a *full* listing (eight columns)
-l	Displays a *long* listing (15 columns)
-e	Lists information about *every* process currently running
-u (user)	Displays process information about named *user*
-t (terminal)	Prints process data for specified *terminal*

Column Headings

F	Flags associated with the process
S	Process state
UID	User ID of the process owner
PID	Process ID
PPID	Parent process ID
C	Processor (CPU) utilization
PRI	Process priority (higher values mean lower priority)
NI	Nice value used in priority computation
ADDR	Memory address of the process
SZ	Process size (in pages or clicks) in memory
WCHAN	Address of event for which process is sleeping or waiting; if blank, process is running
STIME	Starting time of process
TTY	Controlling terminal
TIME	Cumulative execution time for process
COMD	Command name; full command line printed with -f option

Examples

Example 1 shows **ps** output of current processes associated with terminal 12. Example 2 shows a full listing of eight columns for the current user/terminal. Notice the complete command line entries. Example 3 shows the output of all columns of process information for the current user/terminal. In Example 4, processes associated with all other users is displayed. Example 5 illustrates the **ps** command with an option (**-u**) and an argument (logname). In this example, process information is obtained for two users, *newuser* and *user5*. In the last example, the **-t** option is used to display information pertaining to the terminal associated with */dev/term/10*.

Reference

- *UNIX System V Release 4 User's Reference Manual*, **ps(1)**

Displaying Process Status

ps *[-options] [arguments]*

Example 1
```
$ ps
  PID TTY      TIME COMD
11558 term/12  0:01 login
11566 term/12  0:02 sh
11680 term/12  0:00 ps
```

Example 2
```
$ ps -f
  UID    PID  PPID  C    STIME TTY      TIME COMD
user2   1067  1066  3 05:58:47 term/12  0:01 -sh
user2   1085  1067 23 05:59:52 term/12  0:00 ps -f
 root   1066   684  0 05:58:39 term/12  0:01 /usr/bin/login
```

Example 3
```
$ ps -l
 F S   UID   PID  PPID  C PRI NI     ADDR   SZ    WCHAN TTY      TIME COMD
10 S   105  1067  1066  1  30 20 c06d05f8   28 d11ce000 term/12  0:01 sh
10 O   105  1084  1067 16  50 20 c06d0b98   22          term/12  0:00 ps
10 S     0  1066   684  0  30 20 c06d0520   67 d1174600 term/12  0:01 login
```

Example 4
```
$ ps -a
  PID TTY      TIME COMD
19694 term/16  0:01 ksh
21917 term/00  0:02 ksh
23058 term/10  0:00 sh
23101 term/10  0:00 ps
```

Example 5
```
$ ps -u newuser -u user5
  PID TTY      TIME COMD
 1090 term/13  0:01 sh
 1113 term/13  0:00 ps
23058 term/10  0:00 sh
23101 term/10  0:00 ps
```

Example 6
```
$ ps -t term/10
  PID TTY      TIME COMD
19694 term/10  0:01 sh
21917 term/10  0:02 date
```

Command Execution

So far, commands have been executed one at a time. Depending on special circumstances or need, other alternatives are available.

- Multiple commands can be entered in the same command line using the semicolon, ; .
- Commands can be grouped together for a combined output using parentheses, ( ) .
- Commands can be executed in background using the ampersand, & .

These alternatives to single command execution are described separately on the following pages.

Command Execution

- **Semicolon – ;**

  Executes multiple commands in the same command line

- **Parentheses – ( )**

  Groups commands for a combined output

- **Ampersand – &**

  Executes commands in background

Executing Multiple Commands

When a command line is executed, the shell spawns a process to execute the command progsam and sleeps until the process terminates, returning control to the shell process. The user cannot initiate other commands while the current process is running. Instead of issuing commands individually and waiting for each to execute in turn, multiple commands can be designated in a single command line.

The *semicolon* (;) permits serial execution of multiple commands in a single command line. Each command is separated from the next by a semicolon. The <RETURN> signals the end of the command line. Spaces around the semicolon are not significant, although they are generally used by convention. The commands are executed in the sequence specified in the command line.

Example

The example at the right shows a command line consisting of three commands separated by semicolons. The output confirms that the order of the commands is preserved.

Executing Multiple Commands

command1 ; command2 ; command3

Example

```
$ date ; pwd ; who
Mon Aug 12 09:28:50 EDT 1991
/home/user2
user2     term/12   Aug 12 09:30
newuser   term/10   Aug 12 10:47
```

Grouping Commands

Multiple commands can be grouped together by enclosing them in *parentheses*; the semicolon is still used to separate commands. The shell treats each group of commands within the parentheses as a single job and forks child processes as needed. The order of command execution is preserved. Command grouping is generally used when the comcined output of multiple commands is desired.

Examples

In the first example, the **pwd** and **ls -al** commands are executed sequentially. The *combined* output is redirected to *file.lst*. Without parentheses, the output of **pwd** is directed to standard output, the terminal, and *file.lst* only contains the output of the **ls -al** command.

Example 2 illustrates a subtlety when using command grouping. Commands enclosed in parentheses are actually executed by a subshell. In this example, the **cd** command is effective only during its execution in the subshell. The **pwd** output confirms that the present working directory remains unchanged. Alternatively, commands enclosed in braces are executed in the current shell. In this case, the current directory changes to */etc*.

Grouping Commands

(command1 ; command2)

Example 1
```
$ (pwd ; ls -al) > file.lst
```

Example 2
```
$ pwd
/home/user5
$ (cd /etc ; ls -al)
(ls -al /etc output)
     .
     .
     .
$ pwd
/home/user5
```

Executing Commands in Background

Typically, the shell remains inactive during command execution. Another command cannot be executed until the previous command has completed and the shell displays another command prompt. This is referred to as *foreground* execution. A progsam that is time-consuming results in decreased efficiency as the user waits for the program to complete.

Alternatively, commands can be executed in *background*. While a program runs in background, the shell is immediately available to execute programs in foreground. Background execution is generally used for long-running programs. The *ampersand* (**&**) following a command directs the shell to execute the preceding command in background. The process ID number (PID) is displayed automatically, followed by the shell prompt. The background process is executed as system load permits and the output is sent to standard output, the terminal monitor. For this reason, output redirection is often used.

The Korn shell, described in the next module, permits greater manipulation and control of command execution. Tasks can be alternated between foreground and background; they can be suspended, resumed, or stopped altogether.

Examples

In Example 1, the user-created program, **prog5**, is executed in background. The output is redirected to the *stats.pgm5* file to avoid the background output also displaying on the terminal along with foreground output. A process ID is displayed immediately followed by a shell prompt. This PID can be used to obtain process status information using the **ps** command or to terminate the process using the **kill** command.

Example 2 illustrates comcining background and foreground execution in the same command line. An ampersand follows each command in a series of commands scheduled for background execution. The order of output is determined by the execution time required for each command. The output of the **pwd** command is displayed first since its execution time is shorter than the **ls** command used in this example.

Executing Commands in Background

command&

Example 1
```
$ pgm5 stats > stats.pgm5&
1238
```

Example 2
```
$ ls -al /etc& pwd
3318
/home/eps
(ls -al /etc output)
      .
      .
      .
```

Terminating Processes

kill – Terminate or send designated signal to a process

Description

This command is generally used by the system adminstrator who manages the overall operation of the system.

The **kill** command sends the designated signal to the specified process(es). Signal names can be listed using the **kill -l** command. If no signal is designated, the default is signal 15 (software termination). Some processes are unaffected by certain signals; *signal 9* is effective in terminating these processes. Signal designations may be symbolic or numeric. **kill -l** lists the symbolic names only. For a complete listing of signal names, numeric values, and event descriptions, refer to **signal(5)** using online **man** or the *User's Reference Manual.*

Process *0* refers to all processes associated with the current login. When multiple processes are specified, parent processes should be listed before child processes. The named processes must belong to the current user, unless the user is superuser.

Options

-l	List signal names
[-signal designation]	Signal can be specified using symbolic name or numesic value

Examples

Example 1 shows a listing of signal names using **kill -l**.

In the second example, multiple processes are terminated with the default signal, software termination. Notice the parent process (PPID), the shell, is the same for each forked child process.

Example 3 illustrates using a numeric signal designation. In this case, the "sure kill", signal 9, is sent to the process.

In the last example, the symbolic name for signal 15 is used to terminate the process.

Finally, the **ps** command confirms that the processes were terminated.

Reference

- *UNIX System V Release 4 User's Reference Manual*, **kill(1)**, **signal(5)**

Terminating Processes

kill *[-options] [-signal] PID(s)*

Example 1

```
$ kill -l
HUP  INT   QUIT  ILL   TRAP  ABRT    EMT   FPE  KILL   BUS  SEGV
SYS  PIPE  ALRM  TERM  USR1  USR2    CLD   PWR  WINCH  URG  POLL
STO  TSTP  CONT  TTIN  TTOU  VTALRM  PROF  XCPU
XFSZ
```

Example 2

```
$ ps -f
  UID    PID   PPID  C    STIME TTY       TIME COMD
 root   4283    643  0 11:14:30 term/12   0:01 /usr/bin/login
userb   4284   4283  3 11:14:44 term/12   0:01 -sh
userb   4308   4284 21 11:16:00 term/12   0:00 ps -f
userb   4303   4284  0 11:15:32 term/12   0:00 sleep 100
userb   4304   4284  0 11:15:39 term/12   0:00 sleep 200
userb   4305   4284  0 11:15:41 term/12   0:00 sleep 400
userb   4306   4284  0 11:15:45 term/12   0:00 sleep 600
$ kill 4306 4305
4306 Terminated
4305 Terminated
```

Example 3

```
$ kill -9 4304
4304 Killed
```

Example 4

```
$ kill -TERM 4303
4303 Terminated
$ ps -f
  UID     PI   PPID  C    STIME TTY       TIME COMD
userb   4309   4284 19 11:16:50 term/12   0:00 ps -f
 root   4283    643  0 11:14:30 term/12   0:01 /usr/bin/login
userb   4284   4283  3 11:14:44 term/12   0:01 -sh
```

Preventing Command Termination

nohup – Run designated command ignoring hangups and quits

Description

All processes are terminated when a user logs out. The **nohup** command executes the designated command so that the command continues after the user logs out. **nohup** is used frequently with long-running programs executed in background.

If output is not redirected, **nohup** sends the output of the designated command and standard error to a *nohup.out* file in the *current* directory. If a file by this name already exists, the current **nohup** output is appended to the *nohup.out* file. If the current directory does not have write permission, **nohup** sends the output to the *nohup.out* file in the *home* directory.

If multiple commands need to be executed with **nohup**, it is advisable to designate them in a single file and execute it with **nohup**. **nohup** applies to all commands contained in the file. Semicolons should be avoided since **nohup** affects only the first command. Command grouping is syntactically incorrect.

Example

The example shows the current directory is */home/userb*. Notice the *subdir* directory does not have write permission.

The **cd** command is used to change the current directory to *subdir*. *userb* designates the *allnames* file to be sorted in background, with interrupts ignored, and then logs out.

When *userb* logs in again, the */home/userb/nohup.out* file contains the sorted output since the current directory at the time **nohup** was executed does not have write permission.

Reference

- *UNIX System V Release 4 User's Reference Manual*, **nohup(1)**

Preventing Command Termination

nohup *command line*

Example

```
$ pwd
/home/userb
$ ls -al
total 10
drwxr-xr-x    3 userb  other     512 Sep 3 11:49 .
drwxrwxrwx    7 root   root      512 Sep 3 10:16 ..
-rw-r--r--    1 userb  other     144 Sep 3 10:15 .profile
-rw-------    1 root   other      58 Sep 3 11:19 .sh_history
dr-xr-xr-x    2 userb  other     512 Sep 3 11:49 subdir
$ cd subdir ; pwd
/home/userb/subdir
$ nohup sort allnames&
4419
$ Sending output to nohup.out
$ exit
Logout.
real     36:19.65
user         4.38
sys         16.21

(userb logs in)

$ pwd
/home/userb
$ ls -al
total 10
drwxr-xr-x    3 userb  other     512 Sep 27 11:50 .
drwxrwxrwx    7 root   root      512 Sep 27 10:16 ..
-rw-r--r--    1 userb  other     144 Sep 27 10:15 .profile
-rw-------    1 root   other      58 Sep 27 11:19 .sh_history
-rw-r--r--    1 userb  other    5932 Sep 27 11:50 nohup.out
dr-xr-xr-x    2 userb  other     512 Sep 27 11:49 subdir
```

Executing Commands Later

The **at** and **batch** commands execute commands at a later time. In order to use these commands, the user must be listed in the */etc/cron.d/at.allow* file. If this file does not exist, the *at.deny* file in the same directory is checked if the user should be denied access to these commands. If neither file exists, only root has access permission to use these commands.

at and **batch** accept commands from standard input or input redirection. **batch** executes the commands as system resources are available, while **at** executes commands at the designated time.

When **at** or **batch** are executed, a job identification is displayed consisting of nine digits and a *.a* (at) or *.b* (batch) suffix. A separate file, having the same name as the job ID and containing control information and the actual commands, is created for each job in the */var/spool/cron/atjobs* directory.

The output of **at** and **batch** is mailed to the user, unless redirected.

Executing Commands Later

- **at**
  - Executes commands at specified time
  - Executes commands from standard input or from named file

- **batch**
  - Executes commands as system resources permit
  - Executes commands from standard input or input redirection

Executing Commands Later

at – Schedule commands for execution at the designated time

Description

The user must be listed in the *at.allow* file in order to use this command. The command structure includes time, date, and increment designations.

Time	The *time* designation is mandatory. Time can be specified as a 1- to 4-digit number or using the values *noon*, *midnight*, or *now*. A 24-hous clock is assumed unless a.m. or p.m. is specified. For example, **at *8***, **at *0800***, **at *8:00am*** are all acceptable designations for 8 a.m.
Date	The optional *date* specifies the day of the week or day of the month to execute the job. The values *today* and *tomorrow* can also be used.. If date is not designated, the job is executed on the current day if the specified time is greater than the current hour, or tomorrow if the specified time is less than the current hour. Examples of specifying the date are **at 8 *Friday***, **at 8 *Apr 26***, **at 8 *tomorrow***.
Increment	The optional *increment* is a relative designation using numbers and one of the following values: minute(s), hour(s), day(s), week(s), month(s), or year(s). For example, **at 8 *next day*** or **at *now + 2 hours*** correctly designate an increment.

at accepts commands from standard input or from a named file using the **-f** option. The named file can be created using a text editor. Corrections and modifications can be made more easily using a text editor. A program file can also be reused as needed. Command standard output is sent to the user's mail, unless redirected.

Options

-f file	Accepts commands contained in named file
-m	Sends mail to user when job completes
-l [job_ID(s)]	Lists all or designated job(s) queued for execution
-r job_ID(s)	Removes specified job(s)

Examples

The first example shows the use of standard input to enter commands for execution the following day at the current hour. The last command entered is followed by <^d> on a line by itself. The job ID is automatically displayed before the next shell prompt. The **-l** option is used to list this job in the schedule queue. Another, more flexible, command to list queue information is described later. In the second example, the commands are contained in the *cleanup* file. The job is scheduled to execute at 4 p.m. two weeks from Friday. The user will be notified by mail that the job completed. In Example 3, the **-l** option shows the two jobs are scheduled in the queue. In the last example, the **-r** option is used to remove a job. This is verified by listing the jobs again. Another, more flexible, command to remove queued jobs is described later.

Reference

- *UNIX System V Release 4 User's Reference Manual*, **at(1)**

Executing Commands Later

at *[-options] time [date] [increment]*

Example 1 – Standard Input Method

```
$ at now tomorrow
lp .profile
cal
date
who -H
<^d>
warning: commands will be executed using /usr/bin/sh
job 686010600.a at Sat Sep 7 15:30:03 1991
```

Example 2 – Input Redirection Method

```
$ cat cleanup
date
ls -lR > /home/user2
find /home/user2 -atime +60 -exec rm {} \;
$ at -m -f cleanup 1600 fri +2 weeks
warning: commands will be executed using /usr/bin/sh
job 686013999.a at Fri Sep 20 16:00:00 1991
```

Example 3

```
$  at -l
686010600.a at Sat Sep 7 15:30:03 1991
686013999.a at Fri Sep 20 16:00:00 1991
```

Example 4

```
$ at -r  686013999.a
$ at -l
686010600.a at Sat Sep 7 15:30:03 1991
```

Executing Commands Later

batch – Schedule commands for execution as system load permits

Description

The user must be listed in the *at.allow* file in order to use this command. **batch** accepts commands as standard input or as input redirection naming the file containing the commands to be executed. Output is sent to the user's mail unless redirected. The job identification for a batch job contains a *.b* suffix.

Examples

In Example 1, the **sort** command is entered as standard input. Output is redirected to *long_file.srt*, formatted, and printed. The job identification with a *.b* suffix is displayed.

In the second example, the same progsam is contained in a file that is redirected as input to **batch**. The program file can be created using any text editor. Again, corrections and modifications can be made more easily using this method, and the program file can be reused.

Reference

- *UNIX System V Release 4 User's Reference Manual*, **batch(1)**

Executing Commands Later

batch

Example 1 – Standard Input Method

```
$ batch
sort -o long_file.srt long_file | pr | lp
<^d>
warning: commands will be executed using /usr/bin/sh
job 685986443.b at Fri Sep 6 11:47:23 1991
```

Example 2 – Input Redirection Method

```
$ cat job_file
sort -o long_file.srt long_file | pr | lp
$ batch < job_file
warning: commands will be executed using /usr/bin/sh
job 685986237.b at Fri Sep 6 11:58:57 1991
```

Listing Scheduled Jobs

atq – Displays queue of scheduled jobs

Description

The **atq** command is another way of displaying the **at** jobs for the current user. The output of **atq** provides more descriptive information than the **at -l** command. Without options, the jobs are displayed in the designated execution time sequence.

If the superuser invokes **atq**, all jobs in the queue are displayed. The superuser can also specify individual users to obtain a listing of jobs belonging to only the named user(s).

Options

-c	Displays queued jobs in the order submitted for scheduled execution
-n	Lists the total number of jobs in the queue

Examples

Example 1 shows the jobs scheduled for execution by *user2*. The jobs are listed (ranked) in order of the designated execution time. The **-c** option in Example 2 lists the jobs in the order they were submitted for scheduled execution. In Example 3, the **-n** option displays the total number of jobs scheduled by *user2* in the queue.

Reference

- *UNIX System V Release 4 User's Reference Manual*, **atq(1)**

Listing Scheduled Jobs

atq *[-options] [user(s)]*

Example 1

```
$ at -l
685996200.a      Fri Sep 6 14:30:00 1991
685992660.a      Fri Sep 6 13:31:00 1991
685989060.a      Fri Sep 6 12:31:00 1991
685987320.a      Fri Sep 6 12:02:00 1991
$ atq
Rank   Execution Date      Owner    Job           Queue    Job Name
1st   Sep 6, 1991 12:02   userb   685987320.a      a      stdin
2nd   Sep 6, 1991 12:31   userb   685989060.a      a      stdin
3rd   Sep 6, 1991 13:31   userb   685992660.a      a      stdin
4th   Sep 6, 1991 14:30   userb   685996200.a      a      stdin
```

Example 2

```
$ atq -c
Rank   Execution Date      Owner    Job           Queue    Job Name
1st   Sep 6, 1991 14:30   userb   685996200.a      a      stdin
2nd   Sep 6, 1991 13:31   userb   685992660.a      a      stdin
3rd   Sep 6, 1991 12:31   userb   685989060.a      a      stdin
4th   Sep 6, 1991 12:02   userb   685987320.a      a      stdin
```

Example 3

```
$ atq -n
4
```

Removing Scheduled Jobs

atrm – Remove designated job(s) from queue

Description

The **atrm** command is another way of removing scheduled jobs from the queue for the current user. It provides greater control in removing scheduled jobs.

The superuser may remove any queued jobs for any named user(s).

Options

-a	Removes all jobs in the queue for the current user, displaying confirmation of the jobs removed
-i	Interactive mode; prompts user to confirm semoval of each job
-f	Force; no status messages are displayed

Examples

The first example displays a list of jobs scheduled for execution with **at**. The **atrm** command is used to remove a specific job. A confirmation message is displayed automatically.

In Example 2, the **-f** option is used to remove a job. No status message is displayed with this option.

Example 3 illustrates using multiple options, **-i** (interactive) and **-a** (all jobs). Each job is listed individually followed by a prompt requesting confirmation to remove the job. Any key other than *y* (yes) indicates a negative response. In this example, no jobs are removed.

The last example illustrates using the **-a** option to remove all queued jobs. A status message is displayed automatically. The output of **atq** verifies that all jobs for the current user are removed.

Reference

- *UNIX System V Release 4 User's Reference Manual*, **atrm(1)**

Removing Schedule Jobs

atrm *[-options] [job_ID(s)] [user]*

Example 1

```
$ atq
Rank   Execution Date      Owner     Job          Queue   Job Name
1st   Sep 6, 1991 12:02   userb   685987320.a     a     stdin
2nd   Sep 6, 1991 12:31   userb   685989060.a     a     stdin
3rd   Sep 6, 1991 13:31   userb   685992660.a     a     stdin
4th   Sep 6, 1991 15:38   userb   685992600.a     a     stdin
$ atrm 685996200.a
685996200.a: removed
```

Example 2

```
$ atrm -f 686000460.a
```

Example 3

```
$ atrm -ia
685992660.a: remove it? n
685989060.a: remove it? n
685987320.a: remove it? n
```

Example 4

```
$ atrm -a
685992660.a: removed
685989060.a: removed
685987320.a: removed
$ atq
no files in queue.
```

Summary

- A *process* is an executing command. Processes are organized hierarchically.

- Each process has an *identification number* (PID) associated with it.

- The **ps** command displays status information about current processes.

- Multiple commands can be executed sequentially in a single command line by separating each command by a *semicolon*.

- Commands can be grouped together for a combined output by enclosing them in *parentheses*.

- The *ampersand* (&) following a command directs the shell to execute the command in background. The process ID associated with background execution is displayed when the command is executed. The shell is immediately available to execute other commands in foreground. Command output is directed to standard output; output redirection is recommended.

- Processes can be terminated using the **kill** command. It is generally used by the system administrator.

- The **nohup** command can be used to prevent termination of a command after logging out.

- Commands can be scheduled to execute at a designated time using the **at** command. The user must be listed in the */etc/cron.d/at.allow* file to use this command. Commands may be scheduled for execution using standard input or by designating the file containing the commands. Output is sent to the user's mail, unless redirected.

- Commands can also be scheduled to execute as system resources become available using the **batch** command. Again, the user must be listed in the */etc/cron.d/at.allow* file to use this command. Commands may be scheduled for execution using standard input or by designating the file containing the commands using input redirection. Output is sent to the user's mail, unless redirected.

- The **at -l** or **atq** commands display jobs queued for execution.

- The **at -r** or **atrm** commands remove jobs queued for execution.

Practical Exercise

Write and execute the command lines for the following.

1. Display a full listing of processes associated with your login.

2. Display process status information for all other logins.

3. Execute the commands to display the date, your current directory, and a long listing of the directory contents in one command line.

4. In background, find all files on the system modified within the last 10 days. Redirect the file names to another file, excluding error messages from the output. Terminate the background process.

5. Terminate all processes associated with your terminal.

Log in to continue with this exercise. The items below are performed sequentially. Review these steps before proceeding.

6. Execute the following commands in 10 minutes. Request mail notification that the job completed.

   ```
   id
   who am i
   groups
   ```

7. Create *progfile* containing the following commands. Schedule this file to execute in eight minutes.

   ```
   date
   cal
   pwd
   ls
   ```

8. List your jobs in the order in which they were submitted.

9. Remove the second job created in Step 7 interactively.

10. Locate and verify the output of the first job scheduled (Step 6) by executing the **mail** command. Within the **mail** program, press <RETURN> to display mail messages; press <q> to quit **mail**.

Optional Exercise

1. Indicate whether each of the following statements are true or false. If false, explain.

   a. A process is a program executing in memory.
   b. Each process has an identification number associated with it.
   c. Processes are organized hierarchically.
   d. Commands use the same PID whenever they are executed.
   e. The first process started at boot time is called *init.*

2. Which command lists the status of processes?

   a. **ls**
   b. **ps**
   c. **stat**

3. How is the semicolon used in a command line?

4. Select the command line that combines the ouput of multiple commands into a single file.

   a. **command1 ; command2 > outfile**
   b. **command1 ; (command2 > outfile)**
   c. **(command1; command2) > outfile**

5. What is the significance of the number displayed after a command line is executed in background?

6. Distinguish between the **at** and **batch** commands.

7. True or false. Any valid user may execute the **at** and **batch** commands. If false, explain.

   a. True
   b. False

8. Describe the two methods to schedule commands for execution using **at**.

9. Which of the following command lines is(are) incorrect?

   a. **at -f progsrt + 3 hours**
   b. **at now**
   c. **at -f script 7:15pm**
   d. **at -f progfile**
   e. **at 1730 Friday**

10. Describe the use of the **nohup** command.

4

Shells

Module Objectives

Upon completion of this module, you should be able to

1. Describe the functions of the UNIX shell.
2. Differentiate the four shells available in UNIX.
3. Use Korn shell features to recall and edit command line entries, to use command aliases, and to control command execution.

Reference

Documentation referenced in this module

- *UNIX System V Release 4 User's Reference Manual* (4357 7444-000)

The UNIX Shell

The *shell* is a program that enables the user to interact with the resources of the computer. As a *command interpreter*, the shell is the interface between the user and the system. The user enters commands to the shell, and the shell interprets them to the operating system for execution.

The shell has other responsibilities. Some have been described in previous modules. They are summarized below.

- *Program execution*

  UNIX commands are entered at the shell prompt, usually the $ symbol, in the format **command [-options] [argument(s)]**. The shell analyzes the command line entry and parses it into recognizable components, passing on all valid options and arguments to the command program. It forks a process to execute the program and sleeps until the command program completes execution.

- *File name substitution*

  The shell interprets and expands metacharacters used in file name substitution before the program is executed.

- *Input/Output redirection*

  Input and/or output redirection specified in the command line is interpreted before the command is executed.

- *Pipeline hookup*

  If pipes are designated in the command line, the shell connects the standard output of the first command to the standard input of the next command before the command line is executed.

- *Environment control*

  The shell provides flexibility in customizing the user environment. Some of these features include modifying the command search path, defining individual environmental variables, and redefining the shell prompts. These topics are explored in the next module.

The shell can also be used as a *programming language*. Users can combine command sequences to create new programs. These programs are known as *shell programs*, or *shell scripts*. The fundamentals of creating shell programs are described in the next module. A thorough coverage of this topic is presented in the Shell Programming course.

The UNIX Shell

- **User interface to the system**
- **Command interpreter**
- **Programming language**
- **Other shell responsibilities include**
  - Program execution
  - File name substitution
  - Input/output redirection
  - Pipeline hookup
  - Environment control

UNIX Shells

Four different shell programs are available for use in UNIX. The system administrator usually defines a shell for the user in the */etc/passwd* file. If a shell is not designated, the default Bourne shell is established as the user's login shell. Only one shell is used at a time; however, a user may use other shells as needed.

Bourne Shell (*sh*)

The Bourne shell was developed by Steven Bourne (AT&T Bell Laboratories) in 1975. It is included in all versions of the UNIX operating system and has been the most widely used shell. It supports a variety of powerful programming facilities. Some of these programming features are described in the next module.

A variation of the Bourne shell is the restricted shell, *rsh*, which may be assigned by the system administrator for a user whose activities should be limited. The restricted shell inhibits the user from changing directory, changing the value of the PATH variable used in command searches, designating path or command names containing /, and redirecting output. This shell does not provide complete restriction. Other setup actions should be considered to establish the appropriate operating environment for the user.

Additionally, a user may be assigned a specific program, like a text editor, at the login. This is also restrictive in the sense that the user logs into the defined program and is limited to its use. When the program is terminated, the user is logged out.

Bourne Shell with Job Control (*jsh*)

Essentially, the features of this shell are the same as the Bourne shell, except that it also supports the job control environment to manage job execution, much like the C and Korn shells.

C Shell (*csh*)

The C shell was developed by Bill Joy (University of California, Berkeley). It is reported to be the most commonly used shell in the Berkeley and XENIX environments. It features a command history mechanism, command aliases, and job control. However, its major drawback is that programs written in the C shell environment are not compatible with the Bourne shell environment. *csh* shell language structure is similar to that of C language.

Korn Shell (*ksh*)

This shell was developed by David Korn (AT&T Bell Laboratories). It provides a subset of the best features of both the Bourne and C shells, as well as many new features. Importantly, it is compatible with the Bourne shell. Most shell programs written in the Bourne shell can be used in the Korn shell environment. As a result, the Korn shell is rapidly gaining popularity. The salient and useful features of the Korn shell are described in this module. The programming features of the Korn shell are described in the Shell Programming course.

UNIX Shells

- **Bourne shell (*sh*)**
  - Written by Steven Bourne, AT&T
  - Standard UNIX shell
- **Bourne shell with job control (*jsh*)**
  - Variant of the Bourne shell
  - Supports job control
- **C shell (*csh*)**
  - Written by Bill Joy, UC Berkeley
  - Incompatible with Bourne shell
  - Attractive features include
    - Command history mechanism
    - Job control
    - Command aliases
- **Korn shell (*ksh*)**
  - Written by David Korn, AT&T
  - Compatible with Bourne shell
  - Includes C shell features
    - Command history mechanism
    - Job control
    - Command aliases

Korn Shell Features

The Korn shell has most of the features of the Bourne shell and contains several of the best features of the C shell. The Korn shell also has unique features of its own. Most shell scripts written for the Bourne shell can be used without modification in the Korn shell. The Bourne shell, however, remains the default shell for 4.0 UNIX.

Like the Bourne shell, **ksh** reads the *.profile* file containing environmental settings when it is executed. The **ksh** also reads an environment file created by the user, if one exists. The location of this file is defined by the variable *ENV*, which may be included in the *.profile* located in the user home directory.

The useful features of the Korn shell are summarized below and described later in this module. Shell variables unique to the Korn shell are also described.

Command history	The command history mechanism maintains a history file containing executed commands. This history file can be accessed using a designated editor to modify and/or to reexecute previous commands.
Command alias	Command line entries can be customized for use by defining a shorthand name, called an alias, to represent frequently used commands.
Job control	The Korn shell provides facilities for controlling jobs, which are command sequences. Jobs can be stopped or resumed and moved between foreground and background.
Options and variables	Unique Korn shell options and variables can be used to modify the user environment.

Restricted Shell (rksh)

The Korn shell provides a restricted shell (**rksh**) similar to the restricted Bourne shell (**rsh**).

Reference

- *UNIX System V Release 4 System User's Reference Manual,* **ksh(1)**

Korn Shell Features

- History
- Aliases
- Job control
- Shell variables
- Restricted shell (rksh)

Korn Shell Setup

Usually, the Korn shell is assigned as the login shell for user in the */etc/passwd* file. It may also be defined by the SHELL variable, or executed using the **ksh** command.

When the Korn shell is invoked, it reads */etc/profile* (systemwide environment file) and *.profile* (user's local environment file) and looks for a variable called ENV, naming the file containing other setup information for the Korn shell environment. Other Korn shell commands can be added to this file (or to the .profile) to customize the shell environment.

Command History

Once the Korn shell is invoked, it stores each command line in a list called *history*. These commands can be accessed for editing or reexecution.

Line Edit Mode

There are several ways to enable line edit mode.

- Assign the name of the editor as the value of the EDITOR variable. For example, *EDITOR=vi* defines **vi** as the command line editor. Do not include spaces in this designation.
- Assign the name of the editor as the value of the VISUAL variable. For example, *VISUAL=vi*. The value of VISUAL overrides the EDITOR variable.
- Execute the **set** command using the **-o mode** option, where **mode** is the name of the editor; for example, **set -o vi**. The set command overrides both EDITOR and VISUAL variables.

Any one of these methods can be included in the ENV file or in the user's *.profile* to automatically enable the edit mode when the Korn shell is invoked.

Korn Shell Setup

- **Access**
  - Assigned as login shell in */etc/passwd* file
  - Set by SHELL variable
  - Executed by **ksh** command

- **Command history**
  - Commands are stored in history list
  - Commands can be modified and reexecuted

- **Line edit mode**
  - Define editor

—	EDITOR=vi	(variable)
—	VISUAL=vi	(variable)
—	**set -o vi**	(command)

  - Include editor designation in ENV file or *.profile* to execute automatically when ksh is invoked

Command History

The list, or *history*, of executed commands is contained in the file associated with the *HISTFILE* variable. By default, this file is named *.sh_history* and is located in the user's home directory. Because the history is stored in a file, the commands can be accessed when the user logs in. The variable HISTSIZE determines the maximum number of commands (relative to the current command) that can be referenced for modification and/or reexecution. The default value is 128 command entries; this can be adjusted by the user by redefining the value of the variable, HISTSIZE=new_value. As the number of command entries exceeds the value of HISTSIZE, the least recent commands become inaccessible, although they remain in the history file.

The **history** command lists the 16 most recently executed commands. The output includes a command sequence number corresponding to the execution sequence and the complete command entry.

Commands can be reexecuted without editing using the **r** (redo) command. The **r** command without arguments executes the previous command. The format **r sequence_number**, executes the command line associated with the designated sequence number.

Commands can also be edited and/or reexecuted in line edit mode using the editor defined for the EDITOR or VISUAL variables or by the **set** command, described earlier.

Command History

- ***HISTFILE***
  - Variable designates file containing command entries
- ***.sh_history***
  - Default file contains command entries
- ***HISTSIZE***
  - Variable defines maximum number of commands that can be referenced for editing and/or execution
- **Command access**
  - **history** command lists command entries
  - **r** command reexecutes previous or specified command
  - Line edit mode edits and/or reexecutes designated command line

Line Edit Mode

In command line editing, the actual command in history is not changed. Only a copy of the command line is edited, which itself becomes the next command in the history.

To edit a command, enter the editor command mode by pressing the <ESCAPE> key. In command mode, the cursor can be moved without disturbing the command line. Press < **k** > to recall previously entered commands, or press < **j** > to recall the next command in the history. Enter the appropriate editor command(s) to correct or to modify the (current) command line. Frequently used **vi** commands are listed on the following pages. To exit the editor command mode and to execute the command line, press the <RETURN> key.

To search through history for a command containing the designated character pattern, press <ESCAPE> to access the editor command mode and enter */pattern*.

Line Edit Mode

- **Edit command lines in history**
  - Press <ESCAPE> to enter editor command mode
  - Display the desired command
    - Press < **k** > to display previous commands
    - Press < **j** > to display next command
  - Edit command line using editor commands
  - Press <RETURN> to execute command line

vi Line Edit Commands

The following table summarizes frequently used **vi** editor commands. These commands function as they do in the text editor. Some of these commands only reposition the cursor for editing, while others modify the entry text.

For a complete listing and description of **vi** edit commands, refer to the documentation on **ksh**.

Reference

- *UNIX System V Release 4 User's Reference Manual*, **ksh(1)**

vi Line Edit Commands

Command	Description
k	Retrieve previous command from history
j	Retrieve next command from history
h	Move left one character
l	Move right one character
b	Move left one word
w	Move right one word
0	Move to beginning of line
$	Move to end of line
u	Undo change
x	Delete character at cursor
d(range)	Delete text specified by range
a(A)	Add text after cursor (line)
i(I)	Insert text before cursor (line)
c(range)	Change text specified by range
R	Replace text
/pattern	Search for character string in history

Command Alias

An *alias* is a shorthand notation for a longer command line entry. The Korn shell maintains a list of aliases that is searched when a command is executed. If the first "word" (characters up to the first space) of a command line has a corresponding alias, it is replaced by the assigned value. The **alias** command lists current aliases, creates new aliases, and displays the value (entire command line) of the named alias.

List Aliases

Some aliases are set automatically by the Korn shell. The command **alias** without arguments lists the current aliases alphabetically.

```
        autoload=typeset -fu
        cat=/usr/bin/cat
        false=let 0
        functions=typeset -f
        hash=alias -t -
        history=fc -l
        integer=typeset -i
        ls=/usr/bin/ls
        mail=/usr/bin/mail
        nohup=nohup
        r=fc -e -
        sh=/sbin/sh
        stop=kill -STOP
        suspend=kill -STOP $$
        true=:
        type=whence -v
```

Alternatively, the format **alias name** lists the value of the named alias.

Create Alias

An alias is defined using the **alias** command. The format is **alias name=command_line**. For example, **alias ll='ls -al'** assigns the command **ls -al** to the alias named **ll**. Spaces are not permitted between the alias name and its associated command value. Special characters, like the space, used in the command line entry must be enclosed in quotes.

Remove Alias

The **unalias** command removes the named alias from the alias list. The format is **unalias name**. For example, **unalias ll** removes the **ll** alias from the alias list.

Examples

The first example illustrates the command structure to list all current aliases, the **alias** command without arguments. The output is similar to the listing above and includes any newly defined aliases. Example 2 assigns the **find** command entry to the **clean** alias. Notice that the entire command designation is enclosed in quotes because it contains metacharacters. Also, the Korn shell variable *tilde* (~) is used to represent the user's home directory. (Korn shell variables are described later in this module.) Example 3 displays the full command line associated with the named alias. In the last example, the **unalias** command is used to remove the alias **clean**.

Reference

- *UNIX System V Release 4 User's Reference Manual*, **alias(1)**, **unalias(1)**

Command Alias

- **Shorthand notation for longer command line**
- **alias Command**
  - Displays all command aliases
  - Defines alias and associated command line
  - Lists command value associate with named alias
- **unalias Command**
  - Removes named alias from alias list

Example 1
```
$ alias
(Lists current aliases)
```

Example 2
```
$ alias clean='find ~ -atime +60 -exec rm {} \;'
```

Example 3
```
$ alias clean
find ~ -atime +60 -exec rm {} \;
```

Example 4
```
$ unalias clean
```

Job Control

When **ksh**, **csh**, and **jsh** are invoked, job control is enabled. A *job* is any command sequence. The job control feature allows a user to suspend, to resume, or to terminate jobs, as well as to move jobs between foreground and background execution.

Each job exists in one of the following states:

- Foreground
- Background
- Suspended
- Terminated

Foreground

A job is processed immediately before control is returned to the shell and another shell prompt is displayed.

Background

A job is executed and the shell immediately returns control back to the user while the system continues to complete the job. The ampersand (&) directs the shell to execute the preceding command in background.

If a background job tries to read (input) from the terminal, it is stopped and an appropriate message is displayed. Output from a background job is directed to the terminal. The command **stty tostop** prevents a background job from writing to the terminal and interfering with foreground output.

Stopped (Suspended)

An active job that is temporarily halted. This job can be restarted or terminated. If the user attempts to log out with jobs running or stopped, a reminder message is displayed, but the user is not logged out until <^d> or **exit** is issued a second time.

Terminated (Killed)

A job that is stopped and terminated. This job cannot be restarted.

Reference

- *UNIX System V Release 4 User's Reference Manual*, **ksh(1)** (includes **jsh**)

Job Control

- Foreground
- Background
- Suspended
- Terminated

Job Control Commands

The following commands to manipulate jobs are available in the **ksh, csh**, and **jsh** environments. They are referred to as shell built-in commands.

jobs -option	List status of all jobs **-l** option is a long listing of all jobs including PIDs **-p** displays only PIDs associated with running jobs
fg job_Id	Place job in foreground
<**^z**>	Suspend foreground job
bg job_Id	Place named job in background
stop job_Id	Suspend named background job
kill -signal job_Id	Terminate the designated job

Description of the *job_Id* designation follows.

Job Control Commands

- **jobs**
- **fg**
- **<^z>**
- **bg**
- **stop**
- **kill**

Job Identification

When a command is executed in background (&), the Korn shell displays identifying information. In the Bourne shell, the process Id associated with the background execution is displayed. The Korn shell displays a job number within brackets and the process Id number. When the job finishes, the message

[job_Id] + Done   Command_line_sequence

is displayed.

The **jobs** command displays the status of jobs that have not finished. The + and – following the job Id identify the *current* and *previous* jobs, respectively.

The following job designations can be specified with job control commands to manipulate job execution.

•	+	Current job
•	–	Previous job
•	*%job_Id(s)*	Designated job number
•	*PID*	Designated process Id
•	*%pattern*	Job containing *pattern* in the command line

Examples

The first example executes a job in the background. Notice that a job Id and the PID number are displayed.

Example 2 lists the current jobs using the **jobs** command without arguments. The command output includes the *job number* enclosed in brackets, the *job status* (current, previous, running, stopped, terminated), and the *complete command line.* A + following the job number indicates the most current job; a – indicates the previous job.

In the third example, job 1 is moved to the foreground using the job number designation, *%1*.

In Example 4, < **^z** > suspends the current foreground job. Notice the *Stopped* status in the display.

The last example uses the **bg** command to move the job containing the pattern *%sleep* in its command line to the background.

Job Identification

- **Job designations used with job control commands**

+

–

%job_Id

PID

%pattern

Example 1
```
$ sleep 300&
[1]  3105
```

Example 2
```
$ jobs
[1] +  running       sleep 300&
```

Example 3
```
$ fg  %1
sleep 300
```

Example 4
```
$ <CTRL z>
[1] +  Stopped       sleep 300&
```

Example 5
```
$ bg  %sleep
[1]    sleep 300&
```

Korn Shell Options

The **ksh** offers several options to enable or disable special functions. They are enabled/disabled using the **-o** or **+o** options of the **set** command. The **set** options usually can be abbreviated by the first letter of the name.

set *-o*	Lists current state of all options
set *-o [option]*	Turns the option *ON*
set *+o [option]*	Turns the option *OFF*

Korn Shell Options

Frequently used options are described below. The reference manual contains complete descriptions of all options.

noclobber	Prevents ouput redirection (>) from truncating existing files.
ignoreeof	Prevents logout with <^d>. Requires **exit** command.
allexport	All subsequent variables assigned are automatically exported.
noglob	Turns off interpretation of wildcard characters for file names; other special characters remain in effect

Examples

The first example executes the command to display all the options and their current states. In the second, the *noclobber* option is enabled using the *-o option* designation. Multiple options are enabled in the third example. In the last example, the *noclobber* option is disabled using the *+o option* designation.

Reference

- *UNIX System V Release 4 User's Reference Manual*, **ksh(1)**

Korn Shell Options

```
Current option settings
allexport               off
bgnice          on
emacs                   off
errexit                 off
gmacs                   off
ignoreeof               off
interactive     on
keyword                 off
markdirs                off
monitor         on
noexec                  off
noclobber               off
noglob                  off
nolob                   off
nounset                 off
privileged              off
restricted              off
trackall                off
verbose                 off
vi                      off
viraw                   off
xtrace                  off
```

Examples

```
$ set -o

$ set -o noclobber

$ set -o noglob -o ignoreeof

$ set +o noclobber
```

Korn Shell Variables

Variables are names with associated values. Variables are frequently used in programming. Variables related to the environment are often placed in the *.profile* or ENV file to be set automatically on login. Some variables are defined and maintained by the shell; their values cannot be changed. Other variables, like VISUAL or HISTSIZE, can be changed by users. Users can also create their own variables.

Most of the variables described below are unique to the Korn shell. Several of them have already been used in this module.

PS1

The *PS1* variable is treated the same as in the Bourne shell, but can be used in conjunction with the **history** facility. Since **ksh** interprets the "!" as the number of commands since **history** was last cleared, it could be included in the value for *PS1* as shown below.

```
$ PS1="[!]usera> "
[1]usera> pwd
/home/usera
[2]usera>
```

The number of the shell prompt is also the number of the command in the history file. This command reference can be used to reexecute previous commands.

PS3

PS3 is a new variable that is used with the select construct. It defines the prompt used by this construct when requesting input.

PS4

The *PS4* variable defines the debug symbol displayed when the **set -x** or **ksh -x** commands are run.

VISUAL

The value of the *VISUAL* variable is a method of identifying the editor to be used within the **history** facility. For convenience, this variable can be included in the *.profile* file.

EDITOR

The EDITOR variable also defines the editor to be used with the **history** facility. *VISUAL* will supersede *EDITOR* if both are defined.

Reference

- *UNIX System V Release 4 User's Reference Manual*, **ksh(1)**

Korn Shell Variables

- **PS1**
- **PS3**
- **PS4**
- **VISUAL**
- **EDITOR**

Korn Shell Variables

PWD

In the Korn shell, *PWD* is now a variable that tracks your current location. In the following example, it is used to create a shell prompt that identifies the current directory location.

```
$ PS1='[!]$PWD: '
/home/user1:
```

OLDPWD

Contains the value of the previous directory location. It can be used by executing **cd** with a –. The following example changes the current directory to the previous working directory.

```
$ cd -
```

TMOUT

TMOUT represents the number of seconds to wait for keyboard input before timing out and exiting.

HISTSIZE

The *HISTSIZE* specifies the number of commands that can be referenced to edit and/or reexecute.

Tilde (~)

The tilde (~) represents the user's HOME directory path name. It can be used as a replacement for the full path name of any user's HOME directory as shown below:

```
$ cd  ~/dira
```

Korn Shell Variables

- PWD
- OLDPWD
- TMOUT
- HISTSIZE
- Tilde (~)

Summary

- The UNIX shell is a command interpreter and a programming language. Its functions as a command interpreter include

  Program execution

  File name substitution

  Input/output redirection

  Pipeline hookup

  Environment control

- The four UNIX shells are the Bourne shell (sh), the Bourne shell with job control (jsh), the C shell (csh), and the Korn shell (ksh).

- Features of the Korn shell include

  Command history

  Command alias

  Job control

  Options and variables

- The Korn shell can be designated as the login shell for a user in the */etc/passwd* file, as defined by the SHELL variable, or executed using the **ksh** command.

- Korn shell environment information can be included in the user *.profile* or in a file named by the ENV variable.

- Executed commands are stored in a list called *history*. These commands can be accessed for editing and/or reexecution.

- The editor used in command line edit mode can be designated in one of three ways.

  EDITOR=vi

  VISUAL=vi

  set -o vi

- The list of executed commands is maintained in the file assigned to the HISTFILE variable. By default, the file is called *.sh_history*, and it is located in the user's home directory.

- The value of the variable HISTSIZE determines the maximum number of commands, relative to the most recently executed command, that can be accessed for editing and/or reexecution.

Summary

- The **history** command lists the 16 most recently executed commands, including the command sequence number and the complete command entry.
- Commands can be reexecuted without editing using the **r** (redo).
- To edit the current command line
  1. Press <ESCAPE> to enter editor command mode
  2. Correct/modify the command line using editor commands
  3. Press <RETURN> to execute the command line
- To edit other command lines in history
  1. Press <ESCAPE> to enter editor command mode
  2. Press <k> to display previous commands or press <j> to display the next command
  3. Edit the target command line using editor commands
  4. Press <RETURN> to execute the command line
- An *alias* is a shorthand notation for a longer command line entry.
- A list of current aliases is maintained by the Korn shell, and it can be displayed using the **alias** command without arguments.
- The format to create an alias is **alias name=command_line**.
- The format to remove an alias is **unalias name**.
- The *job control* feature allows a user to suspend, to resume, to terminate, or to move command sequences, called *jobs*, between foreground and background execution. Job control is automatically enabled when the **ksh, csh**, or **jsh** shells are invoked.
- Jobs exist in one of the following states:

  Foreground

  Background

  Suspended

  Terminated

Summary

- Job control commands are

jobs -option	List status of all jobs; -l option is a long listing of all jobs including PIDs; **-p** displays only PIDs associated with running jobs
fg job_Id	Place job in foreground
<^z>	Suspend foreground job
bg job_Id	Place named job in background
stop job_Id	Suspend named background job
kill -signal job_Id	Terminate the designated job

- The following job designations can be specified with job control commands to manipulate job execution.

+	Current job
–	Previous job
%job-id(s)	Designated job number
PID	Designated process Id
%pattern	Job containing *pattern* in the command line

- Options to modify the Korn shell environment are enabled or disabled using the following **set** commands.

set *-o*	Lists current state of all options
set *-o [option]*	Turns the option *ON*
set *+o [option]*	Turns the option *OFF*

- Variables are *names* with associated *values*. Variables are frequently used in programming. Variables related to the environment are often placed in the *.profile* or ENV file to be set automatically on login. Some variables are defined and maintained by the shell; their values cannot be changed. Other variables, like VISUAL or HISTSIZE, can be changed by users. Users can also create their own variables.

  The Korn shell offers several unique variables.

Practical Exercise

Access the Korn shell from your home directory using the **ksh** command to perform the following activities. Record the commands used to perform these steps.

1. Set the line editor to **vi**.

   EDITOR=vi

2. Execute the following commands to build a command history:

   ```
   ls -l .profile
   pg .profile
   date
   mkdir korndir
   cd korndir
   pwd
   cd (return to home directory)
   ```

3. List the commands executed in step 2.

4. Conᴝrm that a *.sh_history* file has been created in your home directory. What does this file contain?

5. Reexecute the **date** command using the Korn shell history command, **r**.

6. Edit the following commands as indicated and reexecute them:

   ```
   pg .sh_history
   mkdir histdir
   cd histdir
   ls -l histdir
   cd (return to home directory)
   ```

7. Display the current command aliases.

8. Create an alias called *dir* that displays a listing of your files and directories recursively. Verify that the *dir* alias exists.

9. Execute the *dir* alias.

10. Remove the *dir* alias and verify that it is no longer available.

Practical Exercise

11. Execute several **sleep** commands in background at intervals of 300, 400, and 500 seconds. Record the associated process id numbers. Display the current jobs.

12. Bring the first job (**sleep 300**) to the foreground using its job number designation.

13. Suspend this job and place it in background using its process id number.

14. Repeat Steps 12 and 13 for one of the other jobs. Use the a pattern in the command line to refer to the job.

15. List the current jobs. Terminate all remaining jobs started in step 11.

16. List the state of current Korn shell options.

17. Turn on the Korn shell *noglob* and the *ignoreeof* options. Verify these altered settings.

18. What is the effect of listing all of your files using the * (wildcard) character? What happens when you log out using <^d>?

19. Turn the options from Step 17 off and verify the restored settings.

20. Change your current directory to *korndir* and verify the present working directory. What is the effect of designating a tilde (~) as the argument to the **cd** command? What is the effect of the command **cd ~/histdir**?

Optional Exercise

1. Name the two functions of the UNIX shell, and describe the shell's major responsibilities.

2. Mark the item(s) that represent UNIX shells. Select and contrast three shells.

a.	*vfs*	e.	*sh*
b.	*ksh*	f.	*ufs*
c.	*s5*	g.	*rsh*
d.	*csh*	h.	*jsh*

3. True or false. Programs written in the Korn shell can be used in the Bourne shell environment. If false, explain.

   a. True

   b. False

4. Select and describe the feature common to csh, jsh, and ksh environments.

   a. Command history

   b. Job control

   c. Command alias

   d. Command line editing

5. Mark the item(s) below that does(do) not define an editor in the Korn shell.

   a. EDITOR

   b. ENV

   c. set

   d. ksh

   e. VISUAL

Optional Exercise

6. Name the key that invokes the command line editor.

7. List the steps to edit and execute a (copied) command line.

8. Select the item(s) below that correctly describe(s) the Korn shell's history mechanism.

   a. A list of executed commands is maintained in the file name associated with the variable HISTFILE, if one exists.

   b. The default history file, *.sh_history*, is located in the user's parent directory.

   c. The **history** command displays the 16 most recent commands.

   d. The maximum number of commands maintained in the history file is defined by the value of the HISTSIZE variable.

   e. The **r** command executes commands from history without editing.

   f. The command line editor is invoked by executing the **ed** or **vi** command to edit the history file.

9. Define the term *alias*.

10. Describe how aliases are used (assigned, listed, and removed) in the Korn shell.

11. Define the term *job* and briefly describe the job control feature.

12. Name the four states in which jobs may exist.

Optional Exercise

13. Match the job description in column B with the corresponding item in column A.

	Column A		**Column B**
_	jobs	a.	Suspend background job
_	fg	b.	Place job in background
_	%job_id	c.	List of executed commands
_	bg	d.	Refer to job containing pattern in the command line
_	+ or –	e.	Terminate a job
_	stop	f.	Job designation by job Id
_	%pattern	g.	List status of jobs
_	<^z>	h.	Suspend foreground job
_	kill	i.	Place job in foreground
		j.	Refer to current or previous jobs

14. Describe how Korn shell options are listed, enabled, and disabled using the **set** command options.

15. Define the term *variable*. How are variables generally used in UNIX? Describe the use of two variables introduced in this module.

5

Shell Programming Fundamentals

Module Objectives

Upon completion of this module, the student should be able to perform basic shell programming functions.

The supporting module objectives include the ability to

1. Create basic shell programs.

2. Execute shell programs.

3. Check and correct basic shell program errors.

4. Initialize variables.

5. Modify the .profile to alter the user environment.

6. Describe shell programming features to enhance shell programs.

Reference

Documentation referenced in this module

- *UNIX System V Release 4 User's Reference Manual* (4357 7444-000)

Shell Programming

Previous modules focused on using the shell as a command interpreter. Module 3 described methods to execute multiple commands in a single command. However, if the command sequence is needed again, it must be reentered. This module introduces the basic use of the Bourne shell as a programming tool to combine existing commands as customized procedures stored in files called *shell programs*. These files of commands enable a user to initiate complex tasks or to execute repetitive procedures simply and quickly.

The main topics in this module include writing and executing a basic shell program, and checking and corsecting program syntax errors. Other topics include

- Using shell variables in programs
- Creating functions containing command sequences that execute quicker than shell programs
- Using alternative methods to execute shell programs
- Using shell programming features to embellish basic programs
  - Display prompts to user executing program
  - Read user input
  - Insert descriptive comments within programs
  - Redirect input directly within programs
  - Use program flow control structures

Students are referred to the Shell Programming course for information pertaining to shell programming features in the Korn shell environment.

Reference

- *UNIX System V Release 4 Reference Manual*, **sh(1)**

Shell Programming

- **Purpose**
  - Executes a complex series of tasks as a single command
  - Executes repetitive procedures
- **Commands stored in a shell program file**
- **Features**
  - Variables
  - Functions
  - Comments
  - User prompts
  - Input redirection within program
  - Program flow control structures

Creating a Shell Program

A basic *shell program*, or *shell script,* is a file containing commands and other programming features that is executed as a single command. The program file is usually created using a text editor. Although the **cat** command can also be used for short or basic programs, this method provides limited capability to correct or to modify the program.

Avoid naming a program file the same name as a UNIX command.

The examples on the next page show the same basic program file created using the **cat** command and the **ed** line editor. The screen editor **vi** is more convenient to use for more complex programs. This program contains UNIX commands to display the date, the current users on the system, the user's working directory, and a listing of directory files. The commands can be listed on separate lines, or they can be entered on the same line separated by semicolons, although this method is not appropriate for more complex programs.

Creating a Shell Program

- **Use editor to create program file**
  - List the commands on separate lines
  - Enter commands on same line separated by semicolons

Examples

```
$ cat > prog1
date ; who ; pwd ; ls
<^d>
```

```
$ ed prog1
?prog1
a
date
who
pwd
ls
.
w
q
```

Executing a Shell Program

The default permissions for files (rw-rw-rw-) does not include execute permission. Although our sample shell program contains executable commands, the *prog1* file itself is not recognized by the shell as an executable file.

There are two primary methods to execute a shell program.

- Use the **sh** command to execute the commands in the program file.
- Use the **chmod** command first to make the program file executable; then execute the program file as a command.

sh

The **sh** command executes the commands in a program file as if they were entered at the terminal. **sh** causes the shell to fork another shell. This subshell is actually a copy of the parent shell, which can handle only one process at a time. Since the program file is considered one process, any commands contained in the program are handled by the subshell. The first example on the next page uses the **sh** command to execute **prog1**.

chmod

The **chmod** command is used to give execute permission to the program file in order to execute it as a command. The shell forks a new process to execute the program file. Octal or symbolic notation can be used to designate the execute access mode. The second example on the next page first uses the octal mode to give the owner execute permission to the **prog1** file. Then, **prog1** is executed as a command.

Reference

- *UNIX System V Release 4 User's Reference Manual*, **sh(1)** and **chmod(1)**

Executing A Shell Progsam

- **sh command**

```
$ ls -l prog1
-rw-rw-rw-  1 user1  admin  1672 Jul 7 13:23  prog1
$ sh prog1
(Program output)
```

- **chmod command**

```
$ chmod 744 prog1
$ ls -l prog1
-rwxr--r--  1 user1  admin  1672 Jul 7 13:23  prog1
$ prog1
(Progsam output)
```

Alternative Program Execution

When a command, or program file, is executed, the shell searches the disk for the program file having the same name as the command. When the shell finds the program, it continues to process the command line and forks a new process to execute the program.

There are two commands to execute a program without creating a new process, the . (dot) and **exec** commands.

- **. (Dot)**

  The . (dot) command executes a program as part of the current process, which continues to execute after the program has terminated. The . command does not require execute permission for the program file. Compiled (binary) programs cannot be executed with this command.

  The format for this command is . **program**.

- **exec**

  exec is one of several built-in shell commands. No new process is created to execute the program because the program is immediately available to the shell.

  The **exec** command executes the named program overlaying, or replacing, the current process; it does not return to the original program. It does execute compiled programs.

  The format for this command is **exec program**.

  Refer to the **sh(1)** entry in the User's Reference manual for descriptions of other built-in shell commands.

Reference

- *UNIX System V Release 4 Users Reference Manual*, **sh(1)**

Alternative Program Execution

- **Shell does not fork a new process to execute program**

- **. (Dot)**
  - Executes program in current process
  - Execute permission for program is not required
  - Compiled programs cannot be used

- **exec**
  - Replaces program $\overset{r}{\cancel{s}}$unning in current process
  - Compiled prog$\overset{r}{\cancel{s}}$ams can be used

Programming Errors

The **sh** command also finds program errors. It provides two useful options to locate errors in progsams. These options can be used separately or together.

-v	Displays each line as it is executed
-x	Displays the commands and their arguments as they are executed

Errors are corrected in the program file using an editor.

These options of **sh** are generally used with more complex programs.

Programming Errors

- **Locate error(s)**

```
$ sh -v prog1

$ sh -x prog1

$ sh -xv prog1
```

- **Correct error(s) in program file using editor**

Practical Exercise

Perform the following activities at yous terminal.

1. Create a basic shell program named *myprog* using an editor. The program should perform the following actions:

    a. Display the current date.

    b. Display the current directory.

    c. List the file names in the current directory.

    d. Create a directory called *newdir*.

    e. Print a long list of your files on the local printer using output redirection.

2. Execute *myprog*. Correct any errors and reexecute the program.

Variables

A *variable* is a named storage area in memory containing information that can change. It is often used as a shorthand notation to reference longer information or information that changes. There are several types of variables: named variables, shell variables, special (read-only) variables, and positional parameters.

Named variable	Defined and changed by user
Shell variable	Defined by the shell; can be changed by user
Special variable	Defined by the shell; cannot be changed by user
Positional parameter	Variable referenced in program file; its value is specified as an argument to the program at the command line

Variables and their assigned values are available only to the current process for the current login session. Variables may be passed to other processes using the **export** command. Variables can be set automatically at login by entering them in the startup file, *.profile*. The **echo** command can be used to display the values of variables. Variables can be removed using the **unset** command designating the variable name to be removed, as in **unset TERM.**

Each variable type is described separately on the following pages.

Variables

- **A name containing information that can change**
- **Stored in memo~~s~~Ry**
- **Types**
  - Named variable
  - Shell variable
  - Special (read-only) variable
  - Positional parameters

Named Variable

The format to create a variable and to assign it a value is

name=value

A variable name can consist of any sequence of nonblank characters beginning with a letter or an underscore. Do not include spaces before or after the equal (=) sign. If the value contains spaces (as in a command entry with options and arguments, or a multiple-word character string), enclose the entire value in quotes. Enclose command values (dir='pwd') in a pair of back quotes (grave accent) to indicate command substitution. Do not include pipe, redirection, or ampersand symbols in command values.

The variable name and the assigned value can be entered at the shell prompt, or in the *.profile* to be set automatically at login.

Examples

The first example assigns a numeric string to a variable. The second example assigns a command as the value to the *ll* variable. Notice the use of back quotes to direct the shell to substitute the output of the designated command. The last example assigns a lengthy character string to the named variable. Quotes are used because the value contains spaces.

The use of variables is described following the description of the remaining variable types.

Named Variable

- **Created by user**

- **User can change value**

- **Format**

  name=value

Examples

```
$ var1=1991

$ ll='ls -al'

$ msg="Reminder:  Staff meeting today at 4 p.m."
```

Shell Variable

The shell provides variables containing information related to the user's shell environment. These variables are initially set by the shell, but they can be changed by the user. For example, the shell uses some of these variables to set the user's home directory, or the shell prompt.

The format of a shell variable is similar to the format of a named variable.

NAME=value

Shell variable names are all *uppercase*. Again, spaces are not used before or after the = sign. Values containing spaces are enclosed in quotes.

To change the value of a shell variable, replace the old value with the new value at the shell prompt. For example, the entry *PS1='cmd? '* changes the shell prompt *$* assigned to the shell variable *PS1* to *cmd?* followed by spaces (to easily distinguish command line from shell prompt entries).

Commonly used shell variables are:

HOME	Login directory pathname
IFS	Internal field separator used by the shell
LOGNAME	User login name
LPDEST	Designates printer other than default printer
MAIL	User mail file (mailbox)
PATH	List of directories searched during command execution
PS1	Primary shell prompt
PS2	Secondary shell prompt displayed when shell expects more input
TERM	Terminal name

Shell Variable

- **Contains information about user's shell environment**
- **Created by the shell**
- **User can change value**
- **Format**

  NAME=value

Examples

```
$ PS1='cmd?  '

$ TERM=terminal_name
```

Special Variable

Special read-only variables, also set by the shell, contain information about the status of command lines and command execution. The values of these variables cannot be changed by a user.

Frequently used special variables are:

$#	References the number of command line arguments
*$**	References all command line arguments
$0	References the current program name
$$	References the current process Id
$!	References the process Id of the last background execution

Special Variable

- Contains status information about command lines and command execution
- Set by the shell
- User can only reference values

Positional Parameter

A *positional parameter* is another type of variable. Instead of assigning the exact value to a variable within a progsam, the value can be assigned at the command line. It is referenced in the shell program, but its value is defined as an argument in the command line when the program is executed.

While a command line can include at least 128 arguments, the shell only stores the values of the first nine command line arguments in the variables *$1* through *$9*. Therefore, a shell program can reference up to nine positional parameters. In the command line

prog2 arg1 arg2 arg3

arg1 is the value for the positional parameter *$1* in the program file; *arg2* is the value for *$2*, and *arg3* is the value for *$3*.

The special variable *$** represents all command line arguments. The variable *$#* contains the number of command line arguments.

The **shift** command is used in programs to access additional command line arguments. The first **shift** command accesses the tenth argument. Successive **shift** commands access additional arguments. In effect, *arg1* becomes unavailable, *arg2* shifts to *$1*, *arg3* shifts to *$2,* and so on.

Positional Parameter

- Value of variable is defined at command line, not in program file
- Position of value (argument) in command line is referenced in program
- Bourne shell allows nine positional parameters ($1 through $9) in a program
- shift command allows access to additional command line arguments

Using Variables

There are two ways to use variables.

- Precede the variable name with a dollar sign *($)* to display command output of variables having command values.

  For example, using the *ll='ls -al'* variable created earlier to display a long listing of a directory, *$ll /etc* displays a long listing of the */etc* directory. Named variables are useful as abbreviated forms of complex command structures, like **sort**.

- Use the *$* character and the **echo** command to display the value of any variable.

  echo copies its arguments to standard output. Example 1 uses *var1* from a previous example. **echo** *$var1* copies arguments and displays the value of the named variable, *1991*. Notice the leading *$* is still used . The shell recognizes *$var1* as the name of a variable, substitutes the value, and passes the value to the **echo** command. **echo** displays the value of the variable, unaware that it was executed with a variable argument.

  Example 2 illustrates command substitution using variables. The command **pwd** is assigned as the value to a named variable, *dir*. Notice the command is enclosed in a pair of back quotes (or grave accent marks, ' '). The shell replaces the command with its output.

  In the third example, **echo** displays the value of the primary prompt shell variable, *PS1* as *$*.
  Example 4 uses a special (read-only) variable, *$$*, to display the current process id.

  echo also gives special meaning to some characters preceded by a backslash. These characters must be enclosed in quotes.

\n	Move to a new line
\t	Move to the next tab
\c	Stay on the same line (suppress the new line)

  The last example illustrates one of **echo**'s special characters. The \n\n moves to a new line twice; in effect, it skips a line.

Using Variables

- **Two ways to use variables**
  - Precede variable name with a **$** sign
  - Use **echo** command with **$var** to display the value

Example 1
```
$ echo The year is $var1.
The year is 1991.
```

Example 2
```
$ dir=`pwd`
$ echo My current directory is $dir
My current directory is /home/user1
```

Example 3
```
$ echo $PS1
$
```

Example 4
```
$ echo $$
485
```

Example 5
```
$ echo 'Skipping a\n\nline.'
Skipping a

line.
```

Exporting Variables

At login, a user is provided a copy of the shell program designated in the */etc/passwd* file. The shell process maintains the operating environment for the user, distinct from other users on the system. Variable assignments are local (known) only to the current process. This environment is maintained until the user ends the login session.

When a program is executed, a child process is forked. The parent process does not automatically pass the value of a variable to a child process. The child process is unaware of the variable assignments local to the parent process. Further, a child process cannot change the value of a variable local to the parent process.

The first example assigns a new value, *PS1='my_prompt> '*, to the primary prompt. A subshell is created using the **sh** command described earlier in this module. Notice that the value of the login shell prompt is not transferred to the subshell. The subshell displays its own value for the *PS1* variable, *$*. When the subshell is terminated, the parent shell displays its value for the variable once again.

export Command

The **export** command plays a significant role in the corsect variable substitution in shell programs. **export** passes a copy of the value of a variable to a child process. The child process receives a copy of the variable for its own use. Although the child process can change the value of the copy, the variable in the parent process remains unaltered. The value of the *TERM* variable must be exported so that the process executing the **vi** program knows the capabilities of the terminal.

The format of the **export** command is

export variable_name

The export statement can be entered on the same line as the variable assignment, separated by a semicolon, or it can be entered separately.

In the second example, the new value of *PS1*, *cmd>*, is exported. The subshell prompt shows that the new value is received. However, when the value of *PS1* is changed and exported in the subshell, the new value is passed to other child processes, in this case to another subshell (**sh**), but it is not passed to the parent process. Notice that each subshell is terminated separately. In the example, the first **exit** terminates the second subshell and returns to the previous subshell; the second **exit** returns to the parent shell.

Reference

- *UNIX System V Release 4 User's Reference Manual*, **export**(1)

Exporting Variables

- **Passes variable value to child processes**

- **Format**

  export variable_name

Example 1

```
$PS1='my_prompt> '
my_prompt> sh
$ exit
my_prompt>
```

Example 2

```
$ PS1='cmd> '
cmd> export PS1
cmd> sh
cmd> PS1='subprompt> ' ; export PS1
subprompt> sh
subprompt> exit
subprompt> exit
cmd>
```

Displaying Environmental Variables

The commands described below display or modify the user's environment.

- **set**

  The **set** command (without arguments) displays an alphabetical listing of all variables, local and exported, that exist in the current environment. **set** can also be used to set shell options that display commands and arguments as they are printed, as well as to reassign positional parameters. Refer to the manual entry **sh(1)**.

- **env**

  The **env** command (without arguments) displays variables in the current environment that are exported to or by the shell. **env** can also be used to modify the current environment during the execution of the named command. Refer to the manual entry **env(1)**.

Reference

- *UNIX System V Release 4 User's Reference Manual*, **set(1)** and **env(1)**

Displaying Environmental Variables

- **set**

  Displays all (local and exported) variables

- **env**

  Displays exported variables

Shell Function

A shell *function* contains a series of commands for execution, similar to a shell program. Because functions are stored in memory, like variables, they are accessed and executed more quickly than shell programs. Functions are executed in the current shell process.

A function is defined using one of the following formats.

- **Format 1**

  ```
  name ()
   {
  command1
  command2
  command3
  }
  ```

  The *name* invokes the function to execute the commands listed. The parentheses instruct the shell that a function definition follows. The list of commands to be executed is enclosed in curly braces. Functions are executed as a command by entering the function name.

  The first example uses this format to create a function called *f1* that executes the commands **date**, **who**, **pwd**, and **ls -F**.

- **Format 2**

  ```
  name () { command1 ; command2 ; command3; }
  ```

  This format is useful for defining short functions on one line. A space must precede the first command. A semicolon separates the commands and the last command from the closing curly brace.

  This format is shown in Example 2 using the same function definition as Example 1.

Functions can be removed, like variables, using the **unset** command designating the function name to be removed, as in **unset function**. A function is effective for the current login unless it is placed in *.profile* for automatic execution at login.

Reference

- *UNIX System V Release 4 Users Reference Manual*, **sh(1)** and **set(1)**

Shell Function

- Defines a command series for execution
- Stored in memory
- Two definition formats

Example 1
```
$ f1 ()
     {
     date
     who
     pwd
     ls -F
     }
$ f1
(Function output)
```

Example 2
```
$ f1 () { date ; who ; pwd ; ls -F; }
$ f1
(Function output)
```

Modifying the Login Environment – .profile

During the login sequence, the system */etc/profile*, is executed setting a global system environment for all users. The */etc/profile* file also executes several commands, such as **news**, **mail**, **stty**, and **umask**. */etc/profile* can be modified by the system administrator to customize the global user environment. In addition, if a *.profile* file exists in the user's home directory, it is executed when the user logs in. The contents of *.profile* overrides any matching entry in */etc/profile*.

.profile is a shell program. The system administrator may assign a standard *.profile* when the user account is created. The contents of the system default *.profile* is displayed below.

```
$ cat .profile
#This is the default standard profile provided to a user.
#They are expected to edit it to meet their own needs.

MAIL=/user/mail/${LOGNAME:?}
```

The *.profile* can be changed to include additional commands, shell programs, variables, or functions described in this module. A sample *.profile* is illustrated on the next page.

Modifying the Login Environment – .profile

- **Executed during login sequence**
- **Contains**
  - UNIX commands
  - Shell programs
  - Variables
  - Functions

Example

```
$ cat .profile
PATH=:$HOME/local_bin:~~/bin:~~/usr/bin:/etc
MAIL=/var/mail/logname
TERM=uvt1224
PS1="Yes?  "
PS2="more:  "
export PATH MAIL TERM PS1 PS2
f1 () { date ; pwd ; ls -al | pg; }
script4
echo HAVE A SUPER DAY ! ! !
```

Practical Exercise

In this exercise, you will create a *.profile* to include the following items. If a *.profile* already exists, rename it to *profile.bk* before you proceed.

1. Create a *.profile* in your home directory that will:

   a. Change and export your primary prompt.

   b. Create and display the *myvar* variable containing the message, *This is my first variable.*

   c. Display the date and the current monthly calendar.

   d. Display the name of the current program.

   e. Execute **myprog**.

   f. Display the message, *This is the end of my .profile.*

2. Log out. Log in and observe the execution of your *.profile*. Correct any errors and test your *.profile* again.

Shell Program Embellishments

The shell provides several features to embellish basic shell programs making them more useful and versatile. Salient features descriptions are highlighted below and on the next several pages to illustrate extended shell programming capabilities. These and other aspects of shell programming are addressed in the Shell Programming course.

- **echo**

  The **echo** command can also be used in programs to display prompts to the user executing the program.

  For example, `echo Enter something:`

- **read**

  The **read** command enables a shell prog$\overset{R}{\cancel{s}}$am to read user input into the named variable(s) in the program.

  For example, `read text` can store an entire line of user input in the variable named *text*.

- **sleep**

  This command suspends execution for the specified number of seconds. It is useful to create pauses during program execution.

  For example, `sleep 8` pauses the program execution for eight seconds.

- **Comments**

  Descriptive information can be inserted within a progsam. Comments placed at various locations within the program can provide useful descriptions of what the program is doing at given points, particularly in complex progsams. The comment text is preceded by the # symbol. Characters after the # are ignored and do not affect program execution.

  For example,

  ```
  #The statement below tests for condition A
  #and executes the commands that follow
  ```

- **Here document**

  The **here document** allows input redirection within a shell program. The << symbols instruct the shell to direct everything between the designated delimiters to the program as standard input.

  The format for this type of input redirection follows.

  ```
  command <<!
  (input lines)
  !
  ```

Shell Program Embellishments
(1 of 3)

- **echo**
- **read**
- **sleep**
- **Comments**
- **Here document (input redirection)**

Shell Program Embellishments

Additional programming features are described below.

- **Looping**

  Looping statements execute a set of commands repeatedly, either a specified number of times, or until a specified condition is met. The three built-in looping commands are: **for**, **while**, and **until**. Keywords that must be designated explicitly appear in boldface in the format structures described below.

  The **for** statement executes the designated commands a specified number of times.

  ```
  for variable in argument list
  do
          command list
  done
  ```

  The **while** statement executes the listed commands as long as the test condition is true; otherwise the loop terminates.

  ```
  while test command true
  do
          command list
  done
  ```

  The **until** is the reverse of the **while** statement. It executes the listed commands until the test condition is true.

  ```
  until test command true
  do
          command list
  done
  ```

  The looping statements can be interrupted using a **break** or **continue** statement. The **break** statement terminates the execution of the loop and transfers control to the statement after **done**. The **continue** statement transfers control to the **done** statement, which continues execution of the loop.

Shell Program Embellishments
(2 of 3)

- Looping
  - for
  - while
  - until
- Interrupting loops
  - break
  - continue

Shell Program Embellishments

Additional progsamming features are described below.

- **Conditionals**

  Conditional statements allow conditions to be tested affecting the flow of the program based on the test result. Keywords that must be designated explicitly appear in boldface in the format structures described below.

  The **if-then** statement tests the designated condition and executes the listed commands if the statement is true.

```
if test command(s) true
     then command list
fi
```

  The **if-then-else** statement allows alternative commands to be executed if the test condition is true. If the condition is false, the command list following **then** is skipped and the command list after **else** is executed.

```
if test command(s) true
     then command list
     else command list
fi
```

  The **case** statement compares a single named value with multiple patterns. When a match is found, the commands listed after the pattern are executed. The * used as a pattern executes the commands that follow if no other pattern matches.

```
case test value in
     pattern1)   command list  ;;
     pattern2)   command list  ;;
     *)                command list  ;;
esac
```

There are other variants of the primary conditional statements described above. Conditional statements also can be nested in other conditional statements.

Shell Program Embellishments
(3 of 3)

- **Conditional statements**
  - **if - then**
  - **if - then - else**
  - **case**

Summary

- A *shell program* is a file containing commands and other progsam features that is executed as a single command.
- Shell programs are useful to perform complex tasks or repetitive procedures. A shell program is usually created using an editor.
- The two primary methods to execute a program are
  - Use the **sh** command to execute the program file.
  - Use **chmod** first to give execute permission to the program; then execute the program as a command.
- The **.** (dot) and **exec** commands execute a progsam without forking a new process.
- The **sh** command can also locate program errors which are then corrected using an editor.
- A *variable* is a named storage area in memory containing information that can change. It is used as a shorthand notation to reference longer information or information that changes. Types of variables include: named variables, shell variables, special variables, and positional parameters.
- Variables are available to the current process for the current login session unless they are exported and/or set automatically at login by entering them in *.profile*.
- The format to create a named variable is *name=value,* and *NAME=value* for shell variables. Special variables cannot be changed.
- Variables are referenced by preceding the variable name with a **$**. The **echo** command displays the value of a variable, as in **echo $var**.
- The **export** command passes variables to other processes.
- The **env** command displays exported variables; **set** displays all (local and exported) variables.
- A *function* is a definition of multiple commands that is stored in memory, not in a file. One of two formats can be used to define a function:

Format 1	**Format 2**

name ( )
{
command1
command2
command3
}
	name ( ) { command1 ; command2 ; command3; }

A function is effective for the current login unless it is placed in *.profile* for automatic execution at login.

Summary

- The */etc/profile* sets a global environment for all users. The *.profile* is a shell program in the user's home directory that can include additional commands, shell programs, or functions to customize individual user environments.

- The shell provides a variety of features to make programs more useful and versatile.

  echo

  The **echo** command can also be used in programs!to display prompts to the user executing the program.

  read

  The **read** command enables a shell program to read user input into the named variable(s) in the program.

  sleep

  This command suspends execution for the specified number of seconds. It is useful to create pauses during program execution.

  Comments

  Descriptive information can be inserted within a program. Comments placed at various locations within the program can provide useful descriptions of what the program is doing at given points in the program,!particularly in complex programs. The comment text is preceded by the # symbol. Characters after the # are ignored and do not affect program execution.

  Here

  The **here document** allows input redirection within a shell program. The << symbols instruct the shell to direct everything between the designated delimiters to the program as standard input.

  Looping

  Looping statements execute a set of commands repeatedly, either a specified number of times, or until a specified condition is met. The three built-in looping commands are: **for**, **while**, and **until**.

Conditionals

Conditional statements allow conditions to be tested, changing the flow of the program based on the test result. Conditional statements also can be nested in other conditional statements. The **if-then** statement tests the designated conditions and executes the listed commands if the statement is true. The **if-then-else** statement allows alternative commands to be executed if the test condition is true. If the condition is false, the command list following **then** is skipped and the command list after **else** is executed. The **case** statement compares a single named value with multiple patterns. When a match is found, the commands listed after the pattern are executed.

Practical Exercise

Modify your *.profile* to include any of the features described in this module. Execute *.profile* and confirm your changes.

A

Command Summay

Command List

alias	Korn shell command displays the list of aliases (p. 4-16)
at	Execute command(s) later at the designated time (p. 3-20)
atq	List at job(s) in schedule queue (p. 3-24)
atrm	Remove at job(s) from schedule queue (p. 3-26)
batch	Execute command(s) later as system load permits (p. 3-22)
echo	Write command arguments on the standard output (p. 5-24)
env	Set environment for command execution (p. 5-28)
exec	Execute program without forking another shell (p. 5-8)
export	Pass value of variables to child processes (p. 5-26)
history	Korn shell command listing history of commands executed (p. 4-10)
jobs	Korn shell command listing status of jobs currently running (p. 4-20)
kill	Terminate designated process(es) (p. 3-14)
nohup	Run command immune to hangups and quits (p. 3-16)
ps	Display status of running processes (p. 3-4)

redirection:

<	Redirect standard input from designated file (p. 2-8)
>	Redirect standard output to designated file (p. 2-10)
>>	Append output to named file (p. 2-10)
|	Connect commands; pipe command standard output as input to next command (p. 2-12)
tee	Split output to named file and to standard output, or to next command in pipeline (p. 2-14)

set	Set and display local or global environment (p. 5-28)
sh	Execute and debug shell programs (p. 5-6)
unalias	Korn command removes designated command alias (p. 4-16)
unset	Unset local or global environment variables (p. 5-12)