CRAY® COMPUTER SYSTEMS

UNICOS PRIMER

SG-2010

Copyright© 1986, 1987 by Cray Research, Inc. This manual or parts thereof may not be reproduced in any form unless permitted by contract or by written permission of Cray Research, Inc.

RECORD OF REVISION

PUBLICATION NUMBER SG-2010

Requests for copies of Cray Research, Inc. publications should be directed to the Distribution Center and comments about these publications should be directed to:

CRAY RESEARCH, INC.
Technical Publications
1345 Northland Drive
Mendota Heights, Minnesota 55120

Revision	Description
	February 1986 – First printing. Documentation to support the Cray operating system UNICOS, release 1.0. This documentation is derived from UNIX System V under license from AT&T Technologies, Inc.
A	October 1986 – Rewrite incorporating many editorial changes made in response to a usability study. This version of the manual is issued with the Cray operating system UNICOS, release 2.0. All trademarks are now documented in the record of revision. All previous versions of this manual are obsolete.
B	July 1987 – Complete rewrite and reorganization to support UNICOS release 3.0. This printing includes new features of UNICOS, many new examples, and documentation for the C shell as well as the Bourne shell. This printing obsoletes previous versions of this manual.

The UNICOS operating system is derived from the AT&T UNIX System V operating system. UNICOS is also based in part on the Fourth Berkeley Software Distribution under license from the Regents of The University of California.

The TCP/IP documentation is copyrighted by The Wollongong Group and may not be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, except as provided in the license agreement governing the documentation or by written permission of The Wollongong Group, Inc., 1129 San Antonio Road, Palo Alto, California 94303. The Wollongong software and documentation is based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California. © The Wollongong Group 1985.

CRAY, CRAY-1, UNICOS, and SSD are registered trademarks and APML, CFT, CFT77, CFT2, COS, CRAY-2, CRAY X-MP, CSIM, IOS, SEGLDR, SID, and SUPERLINK are trademarks of Cray Research, Inc.

3B20 is a trademark of AT&T. Apollo and DOMAIN are registered trademarks of Apollo Computer Inc. CDC is a registered trademark of Control Data Corporation. DEC, VAX, and VMS are trademarks of Digital Equipment Corporation. IBM is a registered trademark of International Business Machines Corporation. IRIS is a trademark of Silicon Graphics, Inc. Pyramid is a trademark of Pyramid Technology Corporation. Sun Microsystems is a trademark of Sun Microsystems, Inc. UNIX is a registered trademark of AT&T.

PREFACE

This manual provides introductory information on the Cray operating system UNICOS running on Cray computer systems.

Other Cray Research, Inc. (CRI) publications detail specific aspects of the operating system. The following CRI manuals relate to UNICOS on any Cray computer system:

Publication	Title/Description
SR-0066	Segment Loader (SEGLDR) Reference Manual; describes SEGLDR, an automatic loader for overlaid and nonoverlaid programs.
SR-2011	UNICOS User Commands Reference Manual; describes UNICOS user commands.
SR-2012	UNICOS System Calls Reference Manual; describes all UNICOS system calls and the error returns for these calls.
SR-2014	UNICOS File Formats and Special Files Reference Manual; describes file formats and devices used by UNICOS.
SG-2016	UNICOS Support Tools Guide; describes software tools available to aid the UNICOS user.
SG-2050	UNICOS Editors Primer; describes the three editors available under UNICOS, *vi*, *ed*, and *ex*.

The following CRI manuals relate to UNICOS on CRAY-2 computer systems only:

Publication	Title/Description
SR-2013	CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual; describes all library routines available to users of the CRAY-2 computer system. The manual also describes macros and opdefs for UNICOS and associated programming languages and contains calling sequence documentation.

The following CRI manuals relate to UNICOS on CRAY X-MP and CRAY-1 computer systems:

Publication	Title/Description
SR-0012	Macros and Opdefs Reference Manual; contains macro and opdef instructions for the Cray operating systems UNICOS and COS.
SR-0113	Programmer's Library Reference Manual; describes the Cray Fortran routines available to users of the CRAY X-MP and CRAY-1 computer systems.

Publication	Title/Description
SR-0136	CRAY X-MP and CRAY-1 C Library Reference Manual; describes the Cray C routines available to users of the CRAY X-MP and CRAY-1 computer systems.

CONTENTS

1. INTRODUCTION

This publication provides the following general information about UNICOS:

Section	Description
2 Basics for Beginners	Introduces basic information for using UNICOS, including logging on, creating, searching, and finding files, and the UNICOS file system structure
3 Beyond the Basics	Describes intermediate-level information about UNICOS, including redirecting the input and output of commands, creating command programs, communicating with other users, and working with applications programs
4 The Bourne Shell	Describes more advanced information specific to the Bourne shell is presented in this section, including shell variables, programming constructs for shell programs, changing the shell environment, and debugging shell programs
5 The C Shell	Discusses more advanced features specific to the C shell, including shell variables, programming constructs for shell programs, changing the shell environment, and debugging shell programs

Using the primer as a tutorial will give you a cursory operational knowledge of UNICOS. Depending on your previous experience with operating systems and the thoroughness with which you approach the exercises suggested, you can expect to complete each section of this manual in approximately 2 hours.

This primer is not intended to be a detailed description of UNICOS, nor does it describe all of the operating system's capabilities. Many of the topics described are discussed in detail in other CRI publications, which are listed in the preface.

1.1 USING THIS MANUAL: WHAT YOU SHOULD KNOW

It is assumed that you, as a reader of this manual, have some experience in programming and are familiar with general programming concepts such as looping, files, file editing and editors, and conditional branches.

The manual is organized into three levels of difficulty. Section 2, Basics for Beginners, provides the fundamentals that a novice must learn to be able to use (UNICOS). Section 3, Beyond the Basics, covers intermediate-level skills for using the system and contains most of the information that general users need to know about UNICOS to use it for submitting programming jobs and to do common operating system tasks, such as file management.

Section 4, The Bourne Shell, and section 5, The C Shell, discuss more advanced material about the UNICOS command interpreters.

NOTE

Interruption of command input, program execution, or output printing is controlled by the front-end computer system and differs from system to system. Before you begin working with UNICOS, it is a good idea to ask your system administrator how you can interrupt a process. Keys commonly used include CONTROL-c, BREAK, DELETE, and RUBOUT. This primer uses "interrupt key" to refer to whichever key is appropriate in your case.

1.2 CONVENTIONS

This primer uses the following typographic and lexical conventions:

Convention	Description
Bold	Indicates file names, including path names and directory names, when used in text.
Italic	Indicates UNICOS commands that are used within text, indicates specific values that you supply for general terms, indicates the system's responses to commands you type, and highlights terminology that is being defined.
blanks	Separate arguments and options to a command in a command line. Blanks may not always be required, but they are used here for readability.

1.3 DEFINITIONS

This subsection defines a few of the more commonly used terms in this primer. See the glossary for a more complete list of definitions.

Term	Definition
Command	The name of a UNICOS command (an executable file or the action of the command)
Command line	A command along with option and arguments to it
Null string	A string of nothing, specified with empty double or single quotes: "" or '' (the use of which becomes obvious later)

Term	Definition
Process	A program that is currently executing
The system	The UNICOS operating system, unless explicitly defined otherwise

1.4 ON-LINE MANUALS

The following UNICOS user documentation is available on-line so that you can display manual pages on your terminal:

Publication	Title
SR-0136	CRAY X-MP and CRAY-1 C Library Reference Manual
SR-2011	UNICOS User Commands Reference Manual
SR-2012	UNICOS System Calls Reference Manual
SR-2013	CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual
SR-2014	UNICOS File Format and Special Files Reference Manual

To display a manual page on your terminal, type the following (where *name* is the name of a command, system call, or library call):

man *name* (followed by a carriage return)

If there is more information than will fit on your terminal screen, the last line of the screen shows a colon in the left corner. Press the space bar or RETURN (depending on your terminal) to display the next screen of information. For example, to display the manual page for the *who* command, type the following:

man who

To display the manual page for the *man* command, type the following:

man man

In some instances, a command has the same name as a system call or library routine. In such a case, issuing the *man* command displays manual pages for all entities with the specified name, in the following order: user command, administrator command, system call, and library routine (as applicable). If you want to see only one of these entries (probably the command), use the following format of the *man* command, where *section* is one of the abbreviations in the table in appendix D, On-line Manual Section Abbreviations:

man *section name*

1.5 READER COMMENTS

If you have any comments about the technical accuracy, content, or organization of this manual, we urge you to share them with us. You can contact us in any of the following ways:

- Call our Technical Publications department directly at (612) 681-5729 during normal business hours.
- Send us electronic mail from a UNICOS or UNIX system at:

  ihnp4!cray!publications
  or
  sun!tundra!hall!publications

- Use the postage-paid Reader Comment form at the back of this manual.
- Write us at the following address:

  Cray Research, Inc.
  Technical Publications Department
  1345 Northland Drive
  Mendota Heights, MN 55120

We value your comments and assure a prompt response.

2. BASICS FOR BEGINNERS

This section explains the basics of using UNICOS.

- Accessing UNICOS
- File operations, which includes creating and displaying files and directories; moving, copying, and renaming files; and searching for text in files.
- Learning the structure of the UNICOS filing system
- Using the UNICOS file structure
- Changing the file and directory structure

2.1 ACCESSING UNICOS

To access UNICOS you must first establish a connection between your terminal and the Cray mainframe. Because establishing this connection can be different at each site, ask your system administrator for the procedure. It might involve a network or station command that brings you to the UNICOS login prompt. You then access UNICOS with a login (supplied by your system administrator), which identifies you to the system. You must know the following two components of your login:

- Login name
- Password

A *login name* is a string of lowercase letters, numbers, or both that you use to identify yourself to the system. The login name must begin with a letter and cannot be more than 8 characters. Any string of letters, digits, or both can be a login name as long as it is unique, that is, different from all other login names on the system.

The *password* is a string of uppercase letters, lowercase letters, numbers, punctuation, or a combination of these that you designate to control access to a login. The password string must have at least 6 characters; however, the system uses only the first 8.

The password for a login is a UNICOS security feature. Usually, every login is assigned a password. When you log in to the system, it requests a password. You must enter the password that corresponds to your login name (your system administrator assigns your initial password, until you change it). The system does not allow you access until you have entered the correct password. Once logged in, you can change your password as often as needed (if periodically required on your system) to ensure that other users are not accessing your login and, consequently, your data (subsection 2.1.2, Changing Your Password, explains how to do this).

2.1.1 LOGGING IN

To log in, first type the appropriate command on your terminal to establish communication between the front-end computer and the Cray mainframe (see your system administrator for this). After you log on, UNICOS responds with the following prompt:

```
login:
```

Type in your login name followed by a RETURN (ASCII keyboard) or new-line character (non-ASCII keyboard). (You must always press RETURN after your commands and responses.) UNICOS then prompts for your password, as follows:

```
Password:
```

Type in your password. If you have entered your login name and password correctly, the system may display one or more "messages of the day" (see subsection 3.3.1, The mail Command). UNICOS next displays the primary prompt string, which is usually the dollar sign ($), followed by a space (though your system administrator may have changed it to something else). If you make a mistake while logging in, or if the system administrator has not set up your login on the system, the system displays the following error message:

```
login incorrect
```

This error message is followed by the login prompt, indicating that you should attempt to log in again.

2.1.2 CHANGING YOUR PASSWORD

Your initial password is usually assigned by the system administrator and may be something fairly obvious, such as your last name. To protect your information, it is wise to change your password as the first thing you do after logging on. At the system prompt ($), type the following command:

```
passwd
```

UNICOS responds as follows:

```
Changing password for login name
Old password:
```

Enter your current password after the colon. UNICOS responds with the following:

```
New password:
```

Enter your new password, which can be 6 or more characters in length (only the first 8 are used). It is recommended that you avoid common words, names, and so on). UNICOS responds with the following:

```
Retype new password:
```

Reenter your new password. This is done in case you made a typographical error the first time, so that your password will not be changed to a mystery word you do not know because it has a typo in it. If you do make a typing mistake, so that the two versions of your new password are not the same, the system starts over, prompting you for your new password. When you have entered the same new password the second time, the system assigns it to your login name, and you have a new password.

2.1.3 LOGGING OFF

Before you try to log off, your terminal should be displaying the $ system prompt. This means that the system is ready for you to enter a command (such as to log off). To log off of UNICOS, press an ASCII End-Of-Transmission (EOT) character. On most terminals, the EOT character is CONTROL-d. Alternatively, you can use the *exit* command (if that does not work, try *logout*), which returns you to the login screen. Then, to disconnect the connection between the Cray mainframe and your terminal, follow the procedure provided by the system administrator.

2.1.4 COMMAND LINE SYNTAX

The UNICOS *shell* is a command interpreter. It is the interface between users and the system, interpreting their typed requests and then initiating the appropriate action. Most actions that you request of UNICOS are performed by programs; most commands initiate programs to complete the requested action. You send requests to the shell in the form of a single line, that is, a string of one or more words, followed by a RETURN. This single line that you enter after the $ prompt is called a *command line*. The shell invokes the appropriate program to complete the command and prompts you again with the $ when it is ready to accept another request.

The first word of a command line is the *command*: This is the name of the program to be executed. All subsequent words on the line are *arguments* to the command. Arguments provide information required by the command program.

The syntax for a command line is as follows:

```
command argument argument . . . RETURN
```

The spaces are required to separate the command and its arguments. Generally, commands accept two types of arguments:

- Options
- File-name arguments

Options consist of a minus sign (–), followed by an alphanumeric character and, in some cases, a value, with no intervening space. File-name arguments specify the file(s) that the command is to process.

The following are examples of command line syntax. (Do not try these right now; they are just to show possible formats):

```
ls -l /bin          (Command, option, file-name argument)

cat /etc/passwd     (Command and a file-name argument)
```

Spaces and tabs, known as *delimiters*, separate the words on a command line. The command is the first set of characters on a command line, up to the first space (or tab). The first argument is the second set of characters up to the next space and so on for successive arguments.

The *return* indicates that you should press the RETURN key on an ASCII keyboard (new-line character on other keyboards). Although subsequent examples do not specify a return, remember to end every command line by pressing RETURN.

When you need or want spaces or tabs within a single argument, enclose the argument in quotation marks. Generally either double or single quotes may be used; for an explanation of the differences between double and single quotes, see either subsection 4.2.3, How Variables, Command Arguments, and Quoting Metacharacters are Processed (Bourne shell), or subsection 5.2.2, How Variables, Command Arguments, and Quoting Metacharacters are Processed (C shell).

For example, to execute a program that requires two arguments such as *john r* and *doe*, the first argument should be *john* and the initial *r*, that is, "john r". The second argument should be *doe*. The required command line in this case would be as follows:

command "john r" doe

2.2 FILES

This subsection discusses the following skills and topics:

- Naming files
- Creating and saving files using the *ed* editor
- List the names of files
- Displaying files
- Renaming, copying, and removing files
- Using the metacharacters * [ ] ? ' with file names
- Searching files for text patterns
- Interpretation of metacharacters
- Using metacharacters within files

2.2.1 NAMING FILES

File names are limited to 14 characters. Although any character can be used in a file name, some characters called *metacharacters* have special meaning; therefore, use only letters, numbers, periods, and the underscore in file names. (Subsection 2.2.6, Using Metacharacters in File Names, defines the metacharacters and their functions.)

By convention, certain suffixes indicate specific file types to some UNICOS utilities; therefore, end file names with these characters only when you want to identify the files as follows:

Suffix	File Type
.a	A file created by *ar* or *bld*
.c	A file containing a C language source program
.f	A file containing a Fortran source program
.h	A C language include file with header data
.l	A *lex* source file or a Fortran listing file
.o	Object code (output from a compiler)
.p	A Pascal source file
.s	An assembly language file
.y	A *yacc* source file

2.2.2 CREATING AND SAVING FILES USING THE ed EDITOR

This subsection briefly covers one way in which you can create files under UNICOS and then save them in non-volatile storage. You can create files with the UNICOS line editor *ed*. Complete tutorials for the *ed* and *ex* line editors, and the *vi* screen editor are in the UNICOS Text Editors Primer, publication SG-2050. The *ed*, *ex*, and *vi* entries in the UNICOS User Commands Reference Manual, publication SR-2011, contain more brief reference material on the two editors.

2.2.2.1 Creating files with ed

Create a file named **doc**, and type at least 10 lines in the file so you have enough text with which to practice the next commands. Type the following lines at your terminal, substituting any content you like for *text*; be sure to read the parenthetical explanations for each step:

```
ed doc      (Invokes the ed text editor)
            (Because this is the first time doc has been named, ed responds with "cannot open
            input file." This sounds bad, but it is perfectly all right.)
a           (Instructs ed to append text)
text
text        (Enter at least 10 lines of text)
text
            (The period signals the end of adding text)
```

The period character (.), which signals the end of the added text, must be on a line by itself. Until it is typed, no other *ed* commands are recognized; everything you type is treated as text to be added. The $ does NOT appear when you are typing in the text editor. From the time you type *ed* at the command line, until you exit the editor with the *q* (quit) command, you will not see the $ prompt. After typing the period in the preceding example, you can make various editing changes to the file, as the next subsection explains.

2.2.2.2 Editing files with ed commands

The *ed* commands all work on the idea of line addressing; before a command letter, you specify the lines on which it is to operate. To specify one line, use the line number. For example, to display the fifth line of your file **doc**, use the *p* (print) command with the number 5 before it, as follows:

```
5p
```

To specify a range of lines, type the first line number of the range, a comma, and then last line number of the range. For example, the following command displays the first 8 lines of a file:

```
1,8p
```

The *ed* editor may respond to the preceding command with a ?, which indicates that there are fewer than the specified number of lines (8) in your file.

The $ character in *ed* is a special line address, specifying the last line of a file. The following command line displays an entire file in *ed*:

```
1,$p
```

Use the following command as a more brief way of displaying your entire file (this may not work on all systems):

```
,p
```

When you first enter *ed*, you are in *command mode*, and all keys that you type are interpreted as *ed* commands. To insert text on a new line before the specified one, use the *i* (insert) command. This puts you in *insert mode* and lets you add lines of text *before* the line you specified with the *i* command. To add text on a new line following the specified one, use the *a* (append) command. This puts you in *append mode* and lets you add lines of text *after* the line you specified with the *a* command. To escape either text mode (a or i) and return to command mode, type a period on a line by itself. After that, the next line that you type is interpreted as a command.

Try inserting the following new line 4 into your file doc:

```
4i
This is a new line 4
```

Try appending the following new line 5 to your file doc:

```
4a
This is a new line 5
```

Now display your file to see how it looks.

To delete lines, use the *d* (delete) command. Delete the two previously added lines with the following command:

```
4,5d
```

The *s* (substitute) command lets you substitute a new string for an old string. To substitute *new* for the first occurrence of *old* on the third line of a file, you would type the following:

```
3s/old/new
```

To substitute all occurrences of *old* with *new* on the third line, you would type the following:

```
3s/old/new/g
```

The *g* indicates *global*; all occurrences on the specified line.

To replace all occurrences of *old* with *new* on lines 4 through 8 of the file, type the following:

```
4,8s/old/new/g
```

If you accidentally make an incorrect substitution, the *u* (undo) command undoes the most recent substitution command. The *u* command also undoes other actions such as deletions, insertions, and so on.

Try the various substitution commands on your file doc to become familiar with them.

The *t* command copies lines. The following command line copies the fourth line of your file doc and places it after the seventh line in the file.

```
4t7
```

The *m* (move) command moves lines. Move the first, second, and third lines of your file **doc** and place them after the sixth line with the following command:

```
1,3m6
```

The last command this subsection covers is the *escape* command. Typing the ! character in an editing session lets you temporarily escape the edit and perform UNICOS shell commands. While editing your file **doc**, get a listing of users on the system with the following command:

```
!who
```

Subsection 2.2.2.4, Saving Files in ed, covers saving files, and subsection 2.2.2.5, Exiting ed, covers exiting the editor.

The *ed* editor has many more commands than those shown here. See the UNICOS Text Editors Primer, publication SG-2050, for a complete description of the *ed*, *ex*, and *vi* editors.

2.2.2.3 Error messages and explanations in ed

If at any time you make an error with commands in *ed*, *ed* responds with the ? character, followed by a terse explanation of the error. Whenever you wish to see a very brief explanation for the most recent error, type the *h* (hint) command as follows:

```
h
```

2.2.2.4 Saving files in ed

To *save* (write to permanent storage) information typed into a file, use the following *ed* command:

```
w
```

The editor responds with the number of characters it wrote into the file. You do not see any prompt because you are still in *ed*. None of the text in your file is stored permanently until you use the *w* (write) command. It is a good safety measure, therefore, to periodically use the *w* command to save information while you are editing a file.

2.2.2.5 Exiting ed

To quit the editor, type the *q* (quit) command as follows:

```
q
```

If you try to exit without first saving your text with the *w* command, the editor displays a question mark (?), which is its shorthand way of asking you if you want to save the text before you exit and lose it. If you want to quit without saving your changes, typing a second *q* gets you out of the text editor; otherwise, a *w* saves the changes. Save your file now and then quit *ed*.

For practice, use the information of the preceding five subsections to create a second file called **temp**, typing the text that follows this paragraph into it. You will need the text for later exercises, so please copy it exactly (punctuation and all). Before you begin, carefully reread the preceding subsections for the sequence of steps you need to follow to create this new file. As you work, refer back to those subsections or the UNICOS Text Editors Primer, publication SG-2050, if you don't understand how to do something.

```
This is a test line of only alphabetic characters
This line contains some numbers, 2 and 19, as well
On this line are the characters % @ !
The numbers 2 and 19, and odd characters % @ ! + are on this line
This line contains the UNICOS metacharacters $ > ' and * as text
All character types are mixed here: $45.00 * 8/carton 'units'
```

2.2.3 LISTING NAMES OF FILES

The *ls* (list) command lists the names (not contents) of your files. Type *ls* at your terminal now. If you have been given an unused login, you should see only the two files (**doc** and **temp**) that you have just created.

If you would like to have the list of names neatly ordered in columns, the -C option (uppercase C) will format the listing into several columns. The following example shows the command and sample output:

```
ls -C

calendar        finances        new_letters     project
do_today        memos           old_letters
```

The names are listed in alphabetical order, but other variations are possible. For example, the following *ls* command, using the -t option, lists files in the order in which they were last changed, with the most recently changed first (the C serves only to format the output in columns):

```
ls -tC

do_today        calendar        project         old_letters
new_letters     memos           finances
```

The -r option reverses the order in which files are listed. This is used along with the regular *ls* (reverse the alphabetical listing of the files) or with the -t option (reverse the last-time-modified order). Compare the following two examples with their counterparts (ls -rC with ls -C, and ls -rtC with ls -tC):

```
ls -rC

project         new_letters     finances        calendar
old_letters     memos           do_today


ls -rtC

old_letters     project         calendar        do_today
finances        memos           new_letters
```

There are a number of other options to the *ls* command that give you additional information about files. Some of those options are covered later in this section. See the UNICOS User Commands Reference Manual, publication SR-2011, for more complete reference information about the *ls* command.

2.2.4 DISPLAYING FILES

There are three commands you can use to display files on your terminal. These three commands, *pg*, *cat*, and *pr* are described in the following three subsections.

2.2.4.1 Displaying files using the pg command

The simplest way to display files is with the *pg* command, using it as follows:

```
pg  filename(s)
```

This shows approximately 20 lines of a file, (the exact number depends on your terminal) then stops to let you read it on the screen. Pressing RETURN displays the next 20 lines of the file, and so on. You can list more than one file name after the command and the files will be displayed, 20 lines at a time, in the order specified.

2.2.4.2 Displaying files using the cat command

Another easy way to display files is with the *cat* command (from *concatenate*). The *cat* command displays the contents of all the files you specify, in the order you name them, with no breaks between the listings. Thus, the files are concatenated and displayed. For example, to show one of the files you have created, type the following:

```
cat doc
```

To show both of your files, type the following:

```
cat doc temp
```

The two files are displayed one after the other on the terminal with no break between them; they are concatenated for display, though the files themselves are not actually merged. To slow down the display of these files to see them one screen at a time, use the following command line:

```
cat doc temp | pg
```

The vertical bar is the *pipe* metacharacter and is explained in Subsection 3.2.2, Combining Commands into One: Pipes. The *pg* is the *page* command.

2.2.4.3 Displaying files using the pr command

The *pr* (*print*) command produces formatted displays of files. As with the *cat* command, *pr* shows all the files named in a list. The difference is that *pr* breaks the file display into pages, with a heading on each page that includes the date and time when the file was last modified, the file's name, and the page number. The *pr* command also adds extra lines to skip over the folds in tractor-feed paper. Thus, the following command line displays file **doc** and then skips to the top of a new page and displays file **temp**:

pr doc temp

The *pr* command can also produce multicolumn output, though this only works if the lines in the file are short enough. If your terminal displays 80 characters per line and you specify an output of 2 columns, the lines in your file cannot be longer than 39 or 40 characters. For example, to display file **doc** in three-column format (you may have to shorten the lines), specify the following command:

pr -3 doc

The *pr* command has several other capabilities; see the *pr* entry in the UNICOS User Commands Reference Manual, publication SR-2011.

2.2.5 RENAMING, COPYING, AND REMOVING FILES

To learn how to rename, copy, and remove files, you will use the two files you have created, **doc** and **temp**. First, verify that these files exist using the *ls* command.

2.2.5.1 Renaming a file

This subsection discusses how to use the *mv* (move) command to rename files. The full function of the *mv* command is to move files from one location to another in the UNICOS filing system, and that use is explained in subsection 2.4, Using the UNICOS File System.

****

CAUTION

If you rename a file with the same name as that of a file that already exists, the existing file's contents are destroyed.

****

The generic format of the *mv* command is as follows:

mv *oldname newname*

Type the following command now:

mv doc newdoc

The contents of **doc** move to the new file named **newdoc**, and **doc** disappears. If you like, use the *ls* command to confirm that the file named **doc** no longer exists and the new one, **newdoc**, has taken its place.

2.2.5.2 Copying a file

To copy a file, use the *cp* command, followed by the existing file's name and then the new file's name.

****

CAUTION

As with renaming a file, copying to an existing file overwrites the existing file.

****

Type this on your terminal:

```
cp temp copytemp
```

This creates a new file, **copytemp**, and duplicates the contents of the file named **temp** into it. If you like, use the *ls* command to confirm that both **temp** and **copytemp** exist. You can also use the *cat* or *pr* command to verify that their contents are the same.

The generic format of the *cp* command is as follows:

```
cp oldname newname
```

Try this command again, making a copy of **newdoc** with its former name **doc**, verifying the process with the *ls* command:

```
cp newdoc doc
```

2.2.5.3 Removing a file

The *rm* command permanently deletes files from the UNICOS filing system. It is used as follows:

```
rm -i filename
```

The **-i** option stands for *interactive* and is an option that makes UNICOS prompt you for a yes or no response (y or n) before it actually deletes the file. This is a safe way of removing files, and it is recommended for new users.

You can delete more than one file, as follows:

```
rm -i filename1 filename2 ...
```

You get a warning message if a named file does not exist or if you are not allowed to write to the file; otherwise, there is no prompt or response. (Subsection 2.5.3, Permissions, covers file permissions.)

Try removing the two files that you have just made with the rename and copy commands. Type the following:

```
rm newdoc copytemp
```

Now use the *ls* command to see which of your files remain.

2.2.6 USING METACHARACTERS IN FILE NAMES

The metacharacters *, [,], and ? are a shorthand notation for identifying file names when you want to do operations such as displaying, listing, or removing certain files (this is known as *pattern matching* in UNICOS and UNIX documentation). The metacharacters ' and \ suppress the special meaning of metacharacters, letting you use metacharacters normally without their special meanings.

Remember, the meanings of these metacharacters as given here are in the context of file and directory names in the shell. These metacharacters have different meanings when used to match text patterns within the text of files (as explained in subsection 2.2.9, Metacharacters Within Files).

2.2.6.1 The * metacharacter

The * means "any characters," matching 0 or more characters of any kind. It can save you repetitive typing of similar file names in a command line. For example, suppose you are typing a large document such as a book. Logically, it divides into small pieces like chapters and sections, so you can type the document as a series of files. One method is to have a separate file for each section of each chapter, as follows:

```
chap1.1
chap1.2
chap1.3
chap1.4
chap2.1
chap2.2
.
.
.
```

Create a series of practice files now, using the *cp* command repeatedly to copy one of your files into files with related names like the ones just listed.

To display the whole book, you could enter the following:

```
pr chap1.1 chap1.2 chap1.3 ...
```

Using the *pr* command like this would be tiresome and could lead to errors from typing so many names. Instead, you can enter the following:

```
pr chap*
```

Because the * means "any characters, including none," this translates into "display (in ascending ASCII order) all files whose names begin with **chap**".

This shorthand notation is not a characteristic of the *pr* command; it is a service of the shell and can be used with almost any shell command. For example, the names of the files composing the book can be listed by using:

```
ls chap*
```

This command line produces the following list of file names:

```
chap1.1
chap1.2
chap1.3
chap1.4
chap2.1
chap2.2
.
.
.
```

The * is not limited to the last position in a file name; it can be used anywhere and can occur several times. The following command line displays all files that contain **memo** or **cft** as any part of their names:

```
cat *memo* *cft*
```

The preceding command line displays all of the following files:

```
cft.old        cft.fourier        memofile        memold        memory        new.cft
```

Further, * by itself matches every file name except names beginning with a . (period), so the following command line following displays the contents of all your files (in alphabetical order):

```
cat *
```

****

CAUTION

Because a * by itself matches every file name not beginning with a . (period), the *rm* * command line deletes all files listed by the *ls* command. Before using *rm* * make sure that none of the files is needed.

Whenever you use the metacharacters ? * or [ ] with the *rm* command, it is strongly recommended that you use the *rm* command's **-i** option.

****

2.2.6.2 The [ ] metacharacters

The * is not the only pattern-matching feature available. The metacharacters [ ] direct the system to match any one of the characters inside the brackets. The following command displays only **chap1.1**, **chap1.3**, and **chap1.4**, out of all the chap1 subsections:

```
cat  chap1.[134]
```

This command line does NOT display **chap1.34** or **chap1.134**, if they exist, because only one of the characters within brackets is matched at a time. The pattern *chap1.[134]* tells the system to match the string, "chap1.", followed by a 1 or a 3 or a 4, and nothing else after that.

To indicate a combination of 1-digit and 2-digit numbers such as 1, 3, 4, 34, and 49, specify the files as follows:

```
cat chap[134] chap34 chap49
```

If you have broken the chapters into subsections such as **chap1.1, chap1.2, chap2.four, chap9.5,** and so on, you could use the * metacharacter as follows to display all the subsections of the chapters 1, 2, and 9:

```
cat chap[129]*
```

The preceding command means "Display the files whose names consist of the combination of **chap** followed by **1, 2, or 9,** followed by any characters." For example, all of the following files would be displayed:

```
chap1.4    chap1.axe    chap2.99    chap9    chap955
```

A range of consecutive letters or digits can be abbreviated by placing the first and last characters of the consecutive range, separated with a hyphen, between the [ ] metacharacters as follows:

```
cat chap[1-49].*
```

The preceding command line displays all existing subsections of chapters 1 through 49 (for example, chap1.3, chap4.6, chap18.one, and so on). Letters can also be used within brackets. The [a-z] pattern-matching feature matches any character in the range a through z (lowercase only). To match all letters, both lowercase and uppercase, use [a-zA-Z]. There can be no space between the two ranges, a-z and A-Z.

2.2.6.3 The ? metacharacter

The ? metacharacter matches any single character, so the following command line lists all files that have single-character names (for instance, **b, x,** or **z**):

```
ls ?
```

The following command lists all files that are the first subsections of all chapters with single-characters after the *chap* prefix (**chap1.1, chap2.1, chapY.1, chapd.1,** and so on):

```
pr chap?.1
```

2.2.6.4 The ' " and \ metacharacters

There are two ways to suppress the special meaning of the metacharacters *, ?, [, and] so that you can use them as ordinary characters. One method uses either the single- or double-quote metacharacters and suppresses the meaning of several metacharacters at once. Place quotes around the argument that contains any of the metacharacters *, ?, , [, or]. For example, if you have a file named **what?**, you could display it by typing either of the following command lines:

```
cat 'what?'
cat "what?"
```

The ' and " metacharacters cannot protect themselves, so if you had files named **don't** and **can"t** you would have to use the following command lines to display them:

```
cat "don't"
cat 'can"t'
```

The second method of suppressing metacharacter meaning uses the \ metacharacter. \ suppresses the special meaning of any single metacharacter that follows it. Place the \ immediately before another metacharacter (including itself) to suppress the meaning of that one character. For example, another way to display the two files, **don't** and **can"t** is with the following command lines:

```
cat don\'t
cat can\"t
```

The \ protects all the other metacharacters, too. The following command displays the contents of a file named **this*dragon:**

```
cat this\*dragon
```

Without the \, any file beginning with "this" and ending with "dragon", with any characters in between, would be displayed, because the * would be treated as meaning "any characters, including none," rather than as simply an asterisk. For example, all of the following would match the pattern *this*dragon*:

```
thisdragon   this99thdragon   thisolddragon   thisisadragon
```

The \ metacharacter can also protect itself against interpretation. To display a file named six\pack, use this command line:

```
cat six\\pack
```

The first \ protects the second \ against interpretation, letting the *cat* command take the second \ as literal input (part of a file name).

The file names given in the preceding examples are merely to illustrate a point. It is poor practice to name files using anything other than letters, numerals, and the underscore, and it can lead to problems in finding or referencing the file.

Another use of the \ is to let you enter input more than one line at a time. If you want to enter a command line and it is more than one screen line in length, you must put a \ at the very end of the first (2nd, 3rd,...) line, just before pressing RETURN. UNICOS responds with another prompt, and you can continue to type the remainder of the command line. When you want to send the command line to be processed, just press RETURN with no \ before it as you normally would. Try it now:

```
pg \
temp
```

This has the same effect as typing *pg temp* on one line. This may not work with some terminal connections to UNICOS. If you have difficulties, try just typing one line that wraps down to the next line on your terminal.

2.2.7 SEARCHING FILES FOR TEXT PATTERNS: THE grep COMMAND

The *grep* (global regular expression printer) command locates a string (also known as a regular expression in UNICOS terminology) in files that you specify. The term *regular expression* indicates that you can search for more than just ordinary strings; metacharacters can be included, letting you search for whole classes of patterns, though metacharacters are interpreted differently in text files than in the shell. Subsection 2.2.8, Interpretation of Metacharacters, discusses these differences.

The simplest format of the *grep* command is as follows (italics represent arguments you supply):

grep *string filename*

The *grep* command searches for *string* in file *filename* and displays all of the lines from the file that contain this regular expression. You can search your file, **temp**, for all lines containing numerals, using metacharacters in the search pattern, as in either of the following command lines:

```
grep [0123456789] temp
grep [0-9] temp
```

This search function can be used in many ways. Try the following examples on your file **temp**, which you created in subsection 2.2.2.5. The file should look like this:

```
This is a test line of only alphabetic characters
This line contains some numbers, 2 and 19, as well
On this line are the characters % @ !
The numbers 2 and 19, and odd characters % @ ! + are on this line
This line contains the UNICOS metacharacters $ > ' and * as text
All character types are mixed here: $45.00 * 8/carton 'units'
```

Type the following command:

```
grep odd temp
```

The result is as follows:

```
The numbers 2 and 19, and odd characters % @ ! + are on this line.
```

The *grep* command returns all of the lines in a file that contain the specified pattern; *odd*, in this case.

Type the following command line:

```
grep This temp
```

You get only the first, second, and fifth lines of the file returned, because *grep* distinguishes between uppercase and lowercase. To make *grep* ignore differences in case, use the -i (ignore case) option, as follows:

```
grep -i This temp
```

From this command line, you would get all but the last line of the file returned, because this pattern matches any cases of the string *this* (which is on all lines but the last).

If you want to find a string, but do not remember in which file it is, type *grep*, followed by the string, followed by a pattern that will match all files in which the string may be. For example, suppose you want to locate the string *illustration* in a book that is on the system. Suppose also that the book is divided up into chapters and subsections, as discussed in subsection 2.2.6.1, The * Metacharacter. If you know that the string *illustration* occurs only in the second subsection of any chapter, you would use the following command:

```
grep illustration chap*.2
```

This command line returns one line for each occurrence of *illustration*, listing first the name of the file, then a colon and the line containing the string. *illustration*. The response could look like this:

```
chap1.2: as shown in the following illustration.
chap1.2: Our discussion begins with an analysis of Nabokov's illustration of
chap3.2: as previous evidence has shown, few illustrations have the impact
```

If you add the **-n** option to the *grep* command, line numbers are also displayed, showing you exactly where in each file the string occurs. Try the **-n** option now, searching for a string in your file **temp**, to see how the output looks.

Suppressing the meaning of metacharacters is particularly useful in conjunction with the *grep* command. Often, in files, metacharacters are used as characters with no special meaning intended. For example, the ? character may frequently be used as punctuation, and the [ ] or ( ) characters may be used to enclose notes, equations, or asides. To search for any strings containing these characters, you must use either the single quotes, double quotes, or backslash to treat the metacharacters as ordinary characters for which to search.

See the UNICOS User Commands Reference Manual, publication SR-2011, for more information about the *grep* command and its many options.

2.2.8 INTERPRETATION OF METACHARACTERS

Using metacharacters in search or substitution patterns can be very tricky because there are two levels at which metacharacters can be interpreted. The first level at which this happens is in the shell. When you enter a command line, the shell looks for metacharacters in it. If you do not want the shell to interpret those metacharacters, you must use the ',", or \ metacharacters to protect them by providing an outer layer for the shell to interpret.

When the shell has finished interpreting metacharacters, the resulting command line, with any unprotected metacharacters already interpreted and substituted, is sent to the command program. The command program can then interpret any metacharacters remaining after the shell's interpretation pass.

To see how this works, read over the following paragraphs that step through the shell's interpretation of this command line:

```
ls -l  ch\$p*
```

This command line displays a long listing for all files in the current directory that have names beginning with the four characters *ch$p*. The command line is sent to the shell, which first looks for metacharacters. The first metacharacter that the shell finds is the \. The shell removes it and ignores the character following it ($). The shell next finds the * and then searches the current directory for all possible matches to the pattern ch$p*. All of these matches (for example, **chp, chp2,** and **ch$pfive**) are then substituted for the pattern ch$p*. Next, the shell searches for the *ls* command program. When it finds *ls*, the shell invokes the program, sending it the **-l** option and the list of file arguments: chp, chp2, and ch$pfive. What the *ls* command program therefore actually sees as input is as follows:

```
-l ch$p ch$p2 ch$pfive
```

This is just as if you had typed the command line:

```
ls -l ch$p ch$p2 ch$pfive
```

This type of substitution occurs for all commands; therefore, if you want a command to interpret metacharacters, you must prevent the shell from interpreting them first. The protective metacharacters ', ", and \ serve this function by providing another layer of metacharacters for the shell to interpret.

2.2.9 USING METACHARACTERS WITHIN FILES

When used to match patterns within files, metacharacters are somewhat different than when used at the shell level to match file or directory names. When you use metacharacters to search for patterns within files, you must usually protect them from interpretation by the shell with the ', ", and \ metacharacters.

2.2.9.1 The ^ metacharacter

The ^ matches the beginning of a line, unless it is within the [ ] metacharacters. The command line that follows searches for lines in file **temp** that begin with the string *on*, but it ignores lines that have the string *on* in any other position. The **-i** (ignore case) option to *grep* makes it match uppercase, lowercase, or mixed case instances of the string.

```
grep -i '^on' temp
```

Try this command two ways on your file **temp**, noting that the string of two letters, *on*, is matched, not just the two-letter word, *on*:

```
grep -i on temp
grep -i '^on' temp
```

When the ^ is the first character within brackets, it acts as a NOT operator, matching any one character not in the set. The following command line searches for lines in **temp** that do not have any numerals in them:

```
grep '[^0-9]' temp
```

Try the following command lines, comparing their results:

```
grep '[0-9]' temp
grep '[^0-9]' temp
```

2.2.9.2 The $ metacharacter

The $ is the opposite of the caret's first meaning; it matches the end of a line. To see how this works, try the following two command lines, remembering that you must protect the $ metacharacter:

```
grep line$ temp
grep line temp
```

To match only empty (blank) lines, use both the ^ and the $, as follows:

```
grep '^$' filename
```

2.2.9.3 The . metacharacter

In a search pattern within a file, the . (period) metacharacter has the same meaning as the shell's ? metacharacter; it matches any single character. The following command line searches for lines in **temp** containing the words *tent, test, text,* and so on. Try it now to see the results.

```
grep 'te.t' temp
```

2.2.9.4 The * metacharacter

When searching for patterns within files, the * metacharacter has a different meaning than it does at the shell level, matching file or directory names. The asterisk matches zero or more occurrences of the single preceding character; therefore, to match one or more (not zero or more) consecutive occurrences of the letter A, use the following command line:

```
grep 'AA*' temp
```

Try the command line to see the results. To match any character(s), just as the asterisk does at the shell level, use the following line:

```
grep '.*' filename
```

The . specifies any one character and the * specifies zero or more occurrences of that character. Combined, therefore, they match anything, just as the * metacharacter does at the shell level (in file name substitution). Try the following two command lines:

```
grep '.*char' temp
grep '*char' temp
```

The second command line produces no output, because the search does not find any strings in **temp** to match the string *char. With no character specified before it, the asterisk is taken literally as a character for which to search, with no special meaning.

2.2.9.5 The protective metacharacters ' " \

The three protective metacharacters have the same meaning within files as they do at the shell level, as described in subsection 2.2.6.4, The ' " and \ Metacharacters.

2.3 STRUCTURE OF THE UNICOS FILE SYSTEM

The UNICOS operating system organizes files into a *hierarchical tree structure.* This is accomplished by grouping related files into *directories.* A *directory* or *directory file* is a file that contains information about the files grouped immediately beneath it on the tree. It is an index to those files, holding information about their size, location, and attributes (such as who owns them, and who can manipulate them and how).

Figure 2-1 shows a sample tree structure.

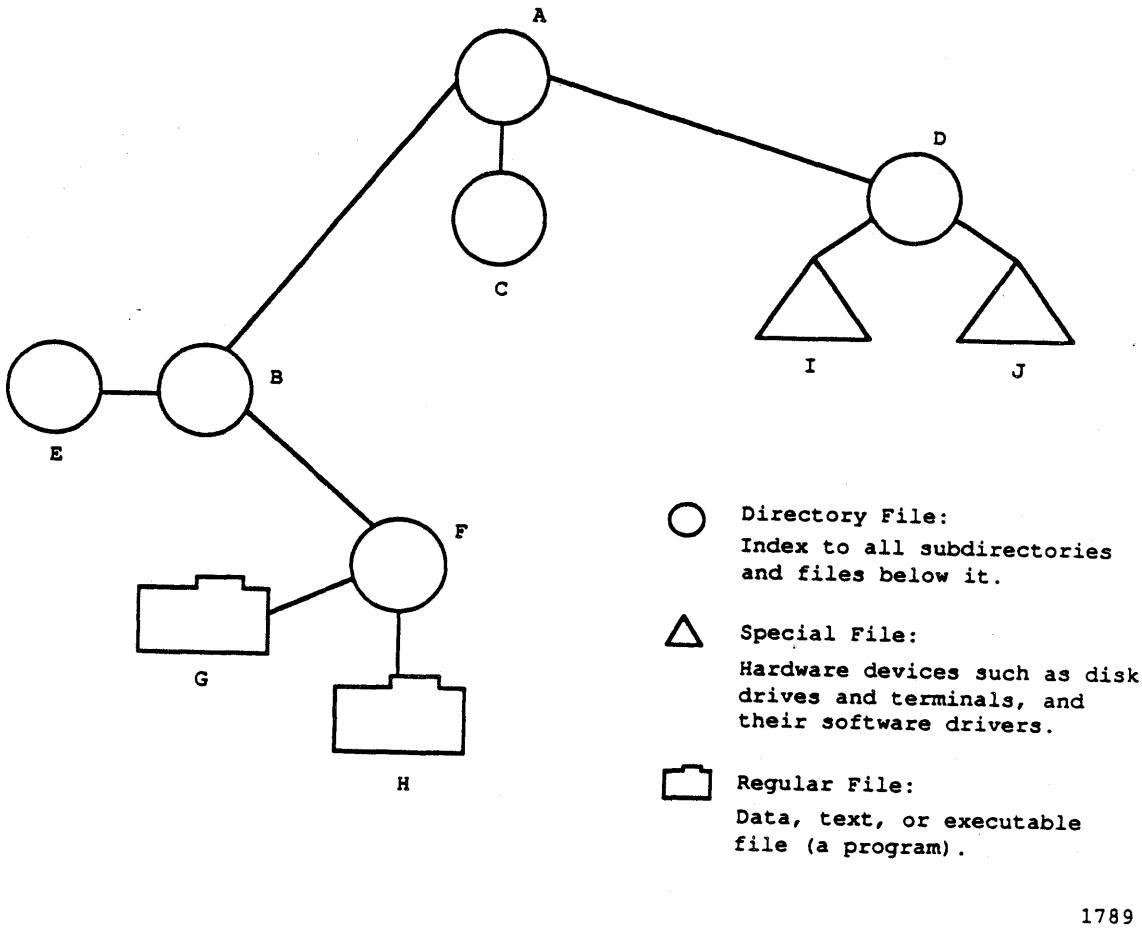

Figure 2-1. Basic Hierarchical Tree Structure

At the top of the *hierarchical tree structure* is the root directory from which all other files and directories branch. It forks into multiple branches, which, in turn, fork into more branches, and so on down the tree until the files are reached.

A directory file is an index to all of the files and subdirectories one level below it. In figure 2-1, the directory shown as A has entries in it for its three subdirectories B, C, and *D*. Directory B has two entries in it: one for each of its subdirectories E and F. Directory F has entries for the two files G and H.

On most UNICOS systems, the directory structure is similar to that shown in figure 2-2. The directory marked / is the *root directory* from which all other files and directories branch. It is common to all UNICOS systems.

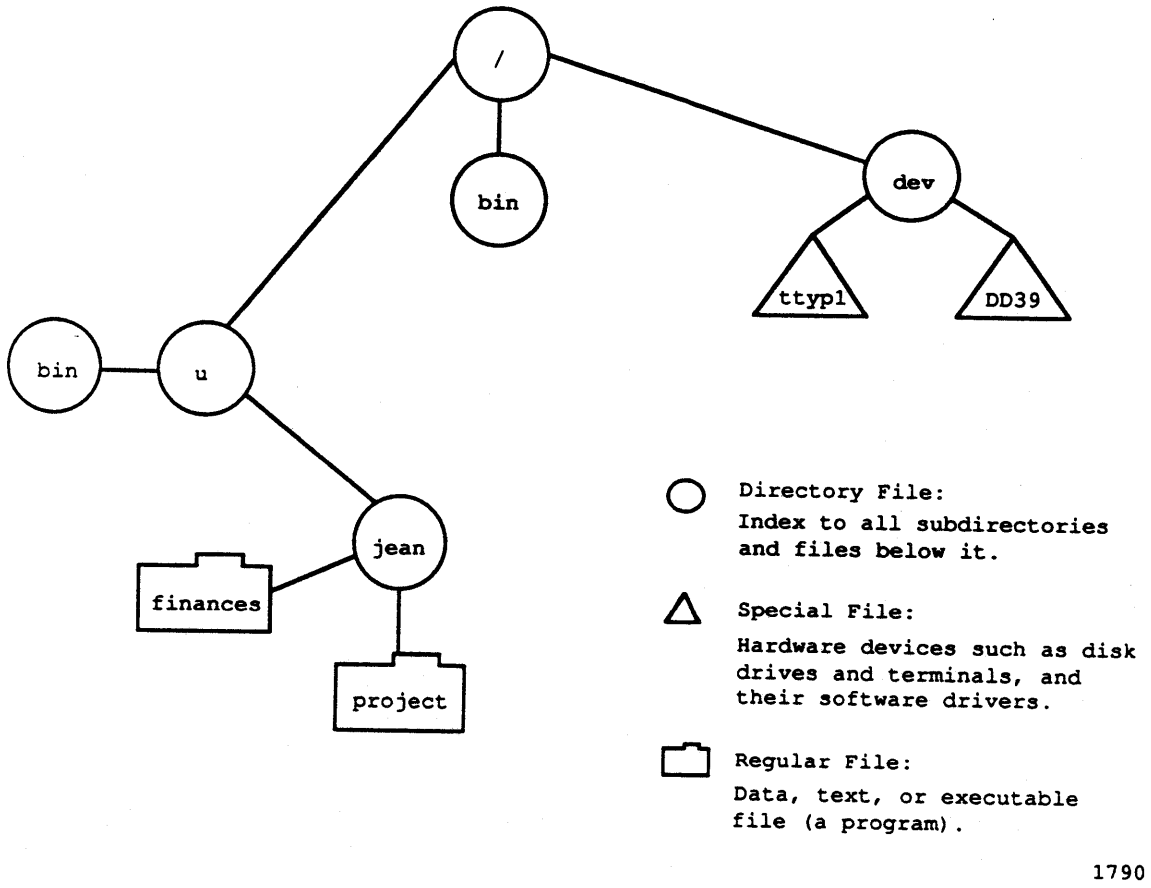

Figure 2-2. UNICOS File System Names

The directory **bin** that branches off the root directory in figure 2-2 is a file of system commands used by both users and the system devices (such as terminals and disk drives). The devices are treated as special files in the directory **dev**. The users are generally subdirectories under the directory **u** (for *user*), such as **jean** in figure 2-2. The file **bin** that branches from the directory **u** is a file of system commands used primarily by users (rather than the system and its devices). As long as they are in different directories, more than one file or subdirectory can have the same name, just as the two **bin** directories do in figure 2-2.

The specific names of these directories and files may vary from one Cray system installation to the next, but the basic structure is the same.

Type the *pwd* command (print working directory) at your terminal now. You will get a response similar to the following:

```
/u/jean
```

This is the response that the user, jean, on the system shown in figure 2-2 would get from typing *pwd*. This indicates that the user is currently located in the directory **jean**, which is located in the directory **u**, which is in turn located in the root directory. The root directory is indicated by the leading slash in the *pwd* response; the other slashes are only to separate directory and file names and have no special meaning.

In the response that you get from *pwd*, the last name, after the last slash, is the name of your current directory (**jean** in figure 2-2). This is known as your *home directory*; it is the place your are always at when you first log on to the system. The other names listed in the response from *pwd* are the names of the directories between your home directory and the root directory.

The *pwd* command tells you what directory you are currently in and all of the directories that are above it on the tree on a direct path back to the root. This list of directories, from any file upwards directly to the root, is known as a *path name*. To see the path name for the the special file representing the terminal at which you are logged on, type the *tty* command (formerly an abbreviation for printing terminals). The response you get is the path name to the file representing the terminal you are on; the final name in that path name being the the system's name for your terminal. On the system shown in figure 2-2, the response to *tty* would be the following:

```
/dev/ttyp1
```

This indicates that the system's name for the terminal is **ttyp1**, the only terminal file shown, because **DD39** is a disk drive file.

Figure 2-3 shows a more complete directory structure, which is probably similar to the one on your system.

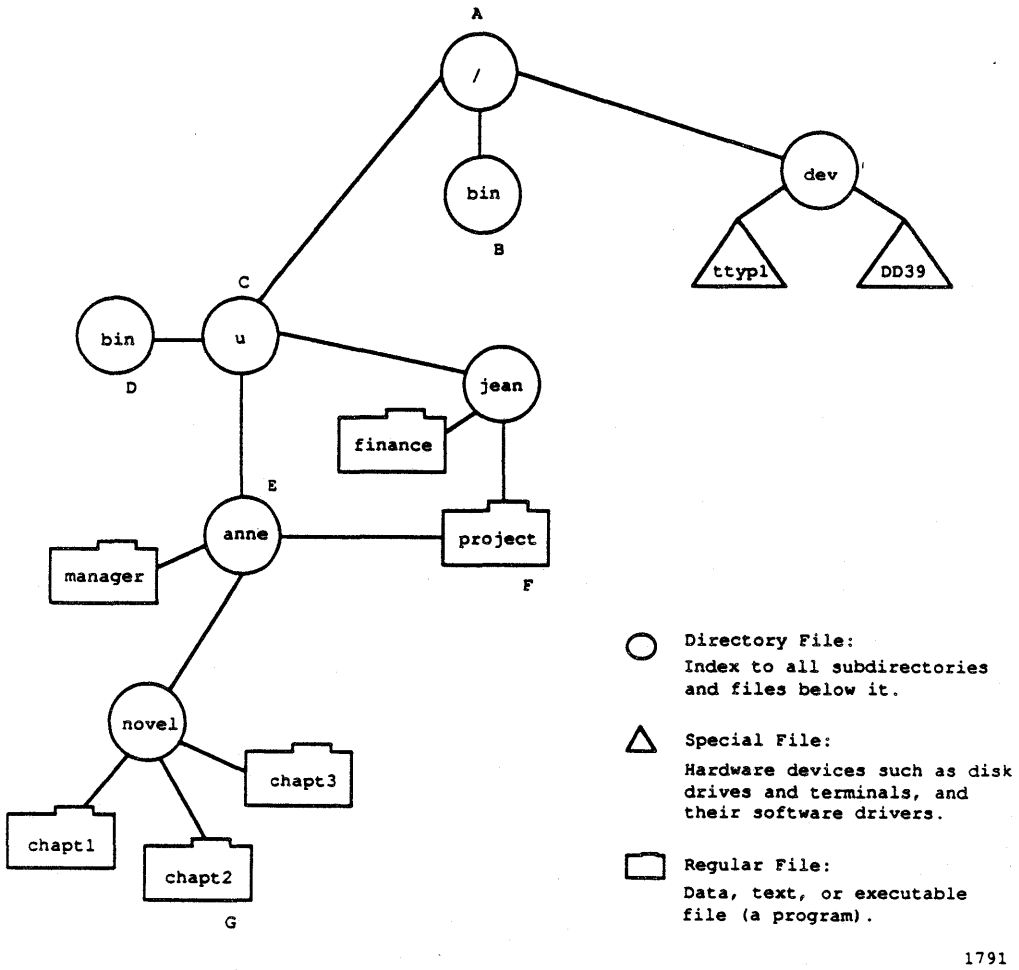

Figure 2-3. Sample UNICOS Directories and Files

To access a file that is not in your current directory (what you get from the *pwd* command), list the names of the directories from the root directly down the tree to the file. Separate their names with slashes and begin with a slash for the root directory. Examples of such path names (figure 2-3) are as follows:

Object	Path Name
A	**/**
B	**/bin**
C	**/u**
D	**/u/bin**
E	**/u/anne**
F	**/u/anne/project** or **/u/jean/project**
G	**/u/anne/novel/chapt2**

As the examples show, path names must proceed downward. The path name **/u/jean/project/anne**, for example, is invalid because it tries to go upward and because it tries to use a regular file, **project**, as a directory file. Path names consist of a list of directory file names, separated by slashes, and optionally followed by one regular file name, or a pattern with metacharacters, at the very end. To access the directory **anne**, you must use the path name **/u/anne**.

NOTE

In UNICOS, anywhere that you can use an ordinary file name, you can instead use a path name that ends in a file.

In all of the preceding examples, the path names start at the root directory and work downward to the desired file. These path names are called *absolute* path names. Another way of accessing a file is to use a *relative* path name, which is the path from your current directory (rather than the root) to the desired file.

Enter the command line *ls -a* at your terminal. The first two entries that you see in the output are a dot and a set of two dots. The single dot is UNICOS shorthand for your current directory. The two-dot set is shorthand for the directory one level above your current directory (known as the *parent directory*). If, for example, you are currently in **/u/dept13/egbert**, the . represents **egbert**, and it is shorthand for the full path name to it. The .. represents **dept13** and is a shorthand way to access files or directories relative to your current directory. In figure 2-3, a person at the directory **/u/anne/novel** could access the file **manager** by using the absolute path name, **/u/anne/manager**, or by using the relative path name, **../manager**. This user could access the file **finance** by typing the absolute path name, **/u/jean/finance**, or by typing the relative path name, **../../jean/finance**.

2.4 USING THE UNICOS FILE SYSTEM

All of the files you have created and used to this point have been created in your home directory. Now that you see how the UNICOS file system is set up, you will learn how to move around the structure and how to locate files among the directories in the tree structure.

The first step to using your system's file structure is to explore it, so you can see how it is arranged. The *ls* command, like any UNICOS command, can take a path name as an argument. To this point, you have only been using the *ls* command to see what files are in your home directory. Now you can see what files and subdirectories are elsewhere on your system.

Type the *cd* (change directory) command now with no arguments to be certain that you are in your home directory. Next, type the *pwd* command (print working directory) again, to see the full path name to your home directory. In figure 2-3, at the directory **jean**, the response to *pwd* would be as follows:

```
/u/jean
```

In the response, **u** is the parent directory of jean and is the directory that contains a subdirectory for each user on the system. Your system should be set up similarly, though the complete set of users may be divided up according to the manager for whom they work and grouped under a manager directory with a path name something like: **/u/manager/individual** or **/u/group/individual.** Whatever the case, type the following at your terminal:

```
ls -C ..
```

This lists (in columns, because of the -C option) all of the files and subdirectories of your parent directory (..). In figure 2-3, at the directory **jean**, the parent directory (..) is **u** and the response to the command line *ls -C ..* is as follows:

```
anne    bin    jean
```

If the arrangement of your system is analogous, the response will list all of the users on the UNICOS system, including yourself. If you are grouped under a manager, as previously mentioned, the response will list all of the files and users in your manager's group. To see what files any other user has, type that user's login name (from the response to *ls -C ..*) after the *ls* command, as follows:

```
ls /u/login_name
```

In figure 2-3, the analogous command line that the user jean would type is as follows:

```
ls /u/anne
```

The response to this command line would be as follows:

```
manager    novel
```

You will not be able to access all directories and files on the system. Some of them will have privacy permissions set so you cannot access them. Subsection 2.5.3, Permissions, discusses this in more detail.

Try the following command:

```
ls /
```

The response will be a list similar to the following:

```
bin
dev
etc
lib
lost+found
tmp
u
```

This is a collection of the basic directories recognized by the system; those that branch directly off the root directory (/). This same command, executed in figure 2-3, would respond with the following:

```
bin
dev
u
```

Use other path names with the *ls* command to develop a mental picture, analogous to that in figure 2-3, of your system's structure. You may occasionally get messages to the effect that you do not have permission to access a directory or file; just ignore them and try another path name.

2.4.1 MOVING AROUND IN THE FILE SYSTEM

Now that you know how your system is structured, you can move around in the tree of directories, locating yourself at any directory so that its files are easily accessible. To change your directory, use the *cd* (change directory) command, followed by the relative or absolute path name to the directory to which you want to switch. For example, if you were at the directory **/u/anne/novel** in figure 2-3 and wanted to move to the directory **/u/jean** by using an absolute path name, you would type the following:

```
cd /u/jean
```

To make the very same change using a relative path name, you would type the following:

```
cd ../../jean
```

Similarly, to see what files that directory contained, while remaining in your own directory (**/u/anne/novel**), you could type either of the following command lines:

```
ls /u/jean
ls ../../jean
```

If you use *cd* alone, with no arguments, it moves you to your home directory, which is the place you are always at when you first log in. This is very helpful if you move around enough to get lost and do not remember the exact path name to return you to a familiar area of the file system.

See the *cd* entry in the UNICOS User Commands Reference Manual, publication SR-2011, for more information on the *cd* command and its options.

2.4.2 LOCATING FILES

When you have used the file system for a while and have created a number of files on it, perhaps in different directories, you may at times forget the name of a file or the name of the directory it is in. If this happens, you can use the *find* command as follows to search for a file. The *find* command displays the path name of every file that has the name you specify. It begins searching at the directory you specify and works its way down the tree structure through all the subdirectories of that directory. The general format of the *find* command is as follows:

find *directory_pathname* -name *filename*1 -print

For example, if user anne, in figure 2-3 wanted to find the file **chapt2**, knowing only that it is somewhere in a subdirectory of her home directory, **/u/anne**, she would type the following command line when located at her home directory:

find . -name chapt2 -print

If user anne were located at a directory other than the one she wants to specify in the search, she would use an absolute path name for the search directory, as follows:

find /u/anne -name chapt2 -print

To find all files named **bin** in the root directory of figure 2-3, a user would type the following command line:

find / -name bin -print

The output looks like this:

/bin
/u/bin

If you are not certain what directory a file is in, specify the directory one level above the *level of directories* you know it is at. If, for example, you know that a file is in some user's directory, but you do not know which one, specify the directory of all users, **/u**, and *find* will first search the /u directory itself and then all of its subdirectories (each one a user's home directory, like **jean** and **anne** in figure 2-3). Naturally, in such a tree structure, the search time can increase exponentially with each higher level you specify in the tree.

See subsection 3.2.2.2, Searching for Strings in Directory Listings, for another, much faster, means of locating files.

2.5 CHANGING THE FILE SYSTEM STRUCTURE

Now that you know how the existing structure is arranged and how to move around in it, you can learn to modify that structure. There are five primary ways of modifying the structure: Creating new files, copying files, moving files and directories to new locations, creating new directories, and linking files and directories to one another. You have already learned the first of these, creating new files; this subsection covers the last four methods of altering the tree structure.

2.5.1 MOVING, COPYING, AND LINKING FILES BETWEEN DIRECTORIES

In figure 2-3, if you were located at the directory **jean** and you wanted to copy the file **chapt1** into your own directory, you would use the following command line:

```
cp /u/anne/novel/chapt1 .
```

The period (.) is shorthand for your current directory, **jean**, so this would copy the file **chapt1** into the directory **jean.**

To copy **chapt1** into your current directory under the name **book**, you would type the following:

```
cp /u/anne/novel/chapt1 book
```

The following are two general formats of the *cp* command that use path names. The words "file" and "directory" in parentheses are NOT part of the command lines, they merely indicate whether the path name preceding them is one file, one or more files, or a directory. the

cp *existing_path_name* (file) *new_path_name* (file)
cp *existing_path_names* (files) *new_path_name_* (directory)

As the second format indicates, you can simultaneously copy a number of files into one directory. In figure 2-3, to copy all of the chapters of anne's novel into jean's directory, you would type the following:

```
cp /u/anne/novel/* /u/jean
```

The *mv* command moves files analogously; its generic formats are as follows:

mv *existing_path_name* (file) *new_path_name* (file)
mv *existing_path_names* (files) *new_path_name* (directory)

With the first format of the *mv* command, you can move a file from one directory to another, retaining the same file name or giving it a new one. For example, in figure 2-3, the command line that follows would move the file **finance** from directory **jean** to directory **anne**, keeping the same name:

```
mv /u/jean/finance /u/anne
```

To perform the same move, but rename the file **budget**, you would use the following command line:

```
mv /u/jean/finance /u/anne/budget
```

With the second format of the *mv* command, you can move a number of files, or the entire contents of a directory, to another directory. For example, in figure 2-3, you could type the following command line:

```
mv /u/anne/novel/* /u/jean
```

This would move all of the files, **chapt**, **chapt2**, and **chpt3** in directory **novel** to directory **jean.**

You can also use multiple path names with this format of the *mv* command. For example:

```
mv /u/anne/novel/* /u/anne/manager /u/jean
```

This command line moves all of the files in the directory **novel**, and the two files **passwd**, and **manager** to the directory **/u/jean.**

Another means of sharing access to files is by linking them between directories. Unlike move or copy, this does not change the location of the file or make more copies of it. The *ln* (link) command allows a file or directory to be accessed from more than one directory without the use of path names. In figure 2-3, the file **project** has been linked between the two user directories **jean** and **anne** so that they both directly access it. The general format of the *ln* command is as follows:

ln *file_path-name directory_path_name*

Suppose, in figure 2-3, that **/u/jean** had originally been the sole owner of the file **project**. To link it to directory **/u/anne** so that both users could work on it, Jean would type the following command line:

```
ln project /u/anne
```

This command accomplishes the link from **project** to **/u/anne** as shown in figure 2-3.

2.5.2 CREATING AND REMOVING DIRECTORIES

It is usually convenient to arrange your files so that files related to one topic are grouped together in one directory separate from other projects. For example, suppose user jean, in figure 2-3, intends to start writing a novel just as user anne is doing. As a first step, jean will want to create a subdirectory under her home directory **jean**; all files related to the book will be stored in this subdirectory. To create such a directory, jean should first be located in her home directory **jean**, and then type the following:

```
mkdir book
```

This creates a new subdirectory, **book**, under the current directory, **jean**. User jean may want to further subdivide the material related to the book into text, outline, and notes, with a subdirectory for each under the directory **book**. The following command, executed from the home directory, **jean**, accomplishes this:

```
mkdir book/text book/outline book/notes
```

Alternatively, these three subdirectories under **book** could be created from within directory **book** with the following two commands:

```
cd book
mkdir text outline notes
```

After creating these directories, user jean would make some files under them, such as **chap1** and **chap2** under directory **text**, and **plot** and **lifelines** under **outline**, and **jan27** under **notes**.

After these changes, directory **jean** would be structured as figure 2-4 shows it:

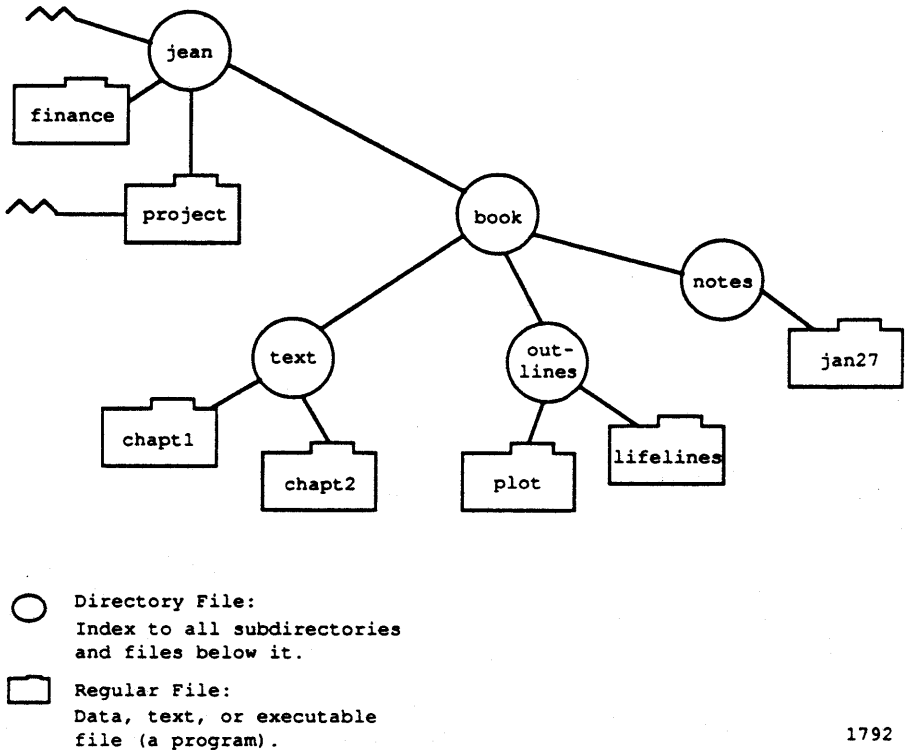

Figure 2-4. Example of an Altered Directory

To see how such alterations of the file structure work, complete the following exercise:

1. From your home directory, (*cd* moves you there) create a subdirectory named **book.**
2. Move to directory **book** and create three files containing any sort of text (copy from this manual, if you like).
3. Create three subdirectories, **text, outline,** and **notes** under directory **book.**
4. Copy the three text files, one into each of the three subdirectories, giving the files new names within the subdirectories.
5. Move to your home directory.
6. Type the command line: ls -RC (note the uppercase letters).

The **-R** option of the *ls* command means "Recursive." It makes the *ls* command list all of the entries in the current directory, including the names of any of its subdirectories. Then it moves down a level into those subdirectories, listing all of their contents, and so on down the tree structure.

Typing the ls -RC command line at directory **jean** in figure 2-4 evokes the following response:

```
book            finances        project

/book:
outline         notes           text

/book/outline:
lifelines       plot

/book/notes:
jan27

/book/text:
chap1           chap2
```

Now that you see how making directories works, you can remove the **book** directory structure just created, if you would like to. The *rmdir* command removes directories from the file structure. Before you can remove a directory, however, it must be empty; it cannot have any files in it. To begin removing the book structure that you have just set up, you must first remove all of the files in the lowest directories. The following commands, executed from your home directory, accomplish this:

```
rm  book/text/*
rm  book/outline/*
rm  book/notes/*
```

These commands remove the files **chap1**, **chap2**, **lifelines**, **plot**, and **jan2** from their directories. Now, to remove all three subdirectories under **book**, type the following:

```
rmdir  book/*
```

Finally, to remove the **book** directory itself, type the following:

```
rmdir  book
```

****

CAUTION

There is no way to undo the *rm* command. When you use the **-r** option, explained next, be certain that you will not need any of the files or directories that you remove, and be certain that you are in the directory where the files are located, or specify the path name to it.

****

All of the preceding steps to remove the book directories and files could have been done with the single following command line:

```
rm  -r  book
```

The **-r** (recursive) option of the *rm* command removes the specified directory, all of its subdirectories, and all files in those directories.

2.5.3 PERMISSIONS

When exploring your system's file structure, you may have received messages to the effect that you did not have permission to look at the files in certain directories or certain files within a directory. You can arrange similar degrees of privacy for your files and directories.

Each directory file and regular file has three sets of three permissions: read, write, and execute (rwx). You can set any of these permissions for any of the three groups: owner (you), the "group" you belong to (if any is defined on the system), and all other system users.

Before changing permissions, you should first determine how they are currently set. You can do this with the **-l** option of the *ls* command, which provides a *long listing* of the files in the specified directory (current directory if none is specified). In your home directory, type the following command line:

```
ls -l
```

You will get a response similar to the following (explanation follows):

```
total 2
-rw-rw-rw-   1   you   pubs     41   Jul   22   14:56   doc
-rwxrw-r--   2   you   pubs   1204   Jan   19   08:11   sample
-rw-rw-rw-   1   you   pubs    213   Jul   22   16:29   temp
```

The *total 2* is showing you how many blocks of system storage space this directory requires. A *block* is a unit of storage space defined in UNICOS as 4096 bytes. (Directories are a minimum of one block in size.) A discussion of column 1 follows, and the remaining columns of this output will be explained later in this subsection.

Look at column 1, which lists all of the permissions for a file. For file **sample** above, the permissions are -rwxrw-r--. The very first letter indicates the type of file, in this case, a - which indicates that **sample** is a regular file. The other possible first characters are a d, for *directory*, or other letters that indicate various types of special files.

The first set of three permissions (rwx) for file **sample** show that the owner has all permissions for the file. The owner may read the file (look at it), write to the file (alter or remove it), and execute it as a command (only if it is a command). The second set of three permissions (rw-) applies to other users belonging to the same group to which the owner belongs. For file **sample**, these users can read or write the file, but they cannot execute it as a command. The last set of three permissions (r--) is for all system users other than the owner or members of the owner's group, and it indicates that these users can only read the file. A permission set of all hyphens (---) would indicate that the users to whom the permission applied could do nothing with the file. (There is an exception to the preceding statement. If you have write permission to a directory, you can delete files from it, even if you do not have write permissions for the files themselves.)

The permissions are each assigned a number:

```
r = 4
w = 2
x = 1
- = 0
```

For each set of three permissions applying to one category of users (owner, group, others), the three (rwx) numbers are added up. Therefore, for file **sample**, the owner has a permission value of 7; rwx = 4 + 2 + 1 = 7. The owner's group members have a permission value of 6 (rw- = 4 + 2 + 0 = 6) and all others have a permission value of 4 (r-- = 4 + 0 + 0 = 4); therefore, the complete set of permissions for this file is represented by the

3-digit number 764. To change the permissions for a file or for a directory and all its associated files, use the *chmod* (change mode) command, followed by the 3-digit number representing the permission values you want to assign to the file, followed by the file or directory name. If, for example, you want to make your file doc more private so that no one but you could look at it or write to it, you could type the following:

```
chmod 711 doc
```

The 1's allow execute permission to your group members and all others on the system, but this does not matter in this case, because the file is not a command. To let anyone other than yourself look at the file and execute it, but be unable to change it, (no write permission), you can use the following command line:

```
chmod 755 doc
```

To have a completely open file for everyone to read, change, and execute, use the following command line:

```
chmod 777 doc
```

The general format of the *chmod* command is as follows, where *number* is the three-digit permission number discussed previously and *name* is one or more directory or file names (including path names):

```
chmod number name
```

Try the *chmod* command now on several of your files, using different permission numbers, then use the *ls -l* command line to see how the files' permissions have changed.

Changing directories does not change file permissions. If, for example, you do not have permission to read or write a file in another user's directory, changing to that directory (with *cd*) does not give you access to that file. The system recognizes your login id and associated permissions, regardless of where you are located in the filing structure.

Now that file permissions have been discussed, we will continue with an explanation of the sample output from the *ls -l* command, shown previously. It is repeated here for your convenience:

```
total 2
-rw-rw-rw-  1  you  pubs    41  Jul  22  14:56  doc
-rwxrw-r--  2  you  pubs  1204  Jan  19  08:11  sample
-rw-rw-rw-  1  you  pubs   213  Jul  22  16:29  temp
```

Column 2 shows the number of links to each file. A link is a path to a file or directory, like an implicit version of the path names you type to access files and directories. It is not a copy of the file, but a means of accessing it.

In this listing, **sample** is the only file with more than one link. Every file normally has one link to the directory it is under, allowing the file to be accessed from and listed in that directory. You can set up other links to access the file from directories other than the one in which it is initially created (see subsection 2.5.1, Moving, Copying, and Linking Files Between Directories).

In figure 2-3, file **project** is the only file with more than the usual single link. This file can be accessed from either of the two directories, **anne** or **jean** and it is listed in both.

Column 3 of the ls -l listing shows who is the owner of a file; that is, the user's login name (*you* in the example). Only the owner of a file and super users can change the file's permissions.

Column 4 shows the group to which the owner belongs (*pubs* in the example).

Column 5 lists the number of bytes in the file.

Columns 6, 7, and 8 show the month, date, and time that the file was last modified, if that was within the last six months. If the file has not been modified in the last six months, these three columns contain the year, month, and date of the last modification.

Column 9 of the long listing gives the file's name.

To get this detailed information for only one or a few files, list the file name(s) after the command, as follows:

ls -l *file1 file2* ...

Naturally, the file names can also be full path names.

See the *chmod* entry in the UNICOS User Commands Reference Manual, publication SR-2011, for more description of the *chmod* command and its options.

3. BEYOND THE BASICS

This section covers intermediate-level use of UNICOS, including the following topics:

- Redirecting command input and output
- Executing multiple commands and shell programs
- Communicating with other users
- Delaying execution of shell programs
- Fortran programs under UNICOS
- Pascal, C, and CAL program files under UNICOS
- The two shells: Bourne shell and C shell
- Changing shells

3.1 REDIRECTING COMMAND INPUT AND OUTPUT

Most of the commands explained so far produce output on the terminal. Other commands, such as the editor, take input from the terminal. The terminal can be replaced by a file for input, output, or both. This is called *input/output redirection*.

3.1.1 REDIRECTING OUTPUT WITH >

Type the following command to display a list of the files in your current directory on your terminal:

```
ls
```

Type this next command line to place that list in a file named **filelist**:

```
ls > filelist
```

File **filelist** is created if it does not already exist, or it is overwritten if it does already exist. Symbol > means "put the output of the preceding command into the following file, rather than on the terminal." No output (except error messages) is produced on the terminal. Try this now at your terminal, typing the following command:

```
ls -C > filelist
```

To look at the contents of the file, use the command line:

```
pg filelist
```

The *cat* command normally lists the contents of one or more files on the terminal. By using it with the > symbol to place its output into a file, you can combine several files (**f1**, **f2**, and **f3**) into one file (**three**):

```
cat f1 f2 f3 >> three
```

Use the preceding form of the *cat* command now, combining your two files, **doc** and **temp**, and placing the combined output into a new file **combo**. Then use the *pg* command on **combo** to verify that it is the concatenation of **doc** and **temp**.

You can use this redirection with most UNICOS commands. Suppose that you want to know all of the lines containing metacharacters in your file **temp**. To have a permanent record of just these lines in a file (called **metas**), you could use the following command line:

```
grep "[$>'*]" temp > metas
```

Type in the preceding command line now. File **metas** should contain the following two lines from **temp**:

```
This line contains the UNICOS metacharacters $ > ' and * as text
All character types are mixed here: $45.00 * 8/carton 'units'
```

You may, at times, not actually want to use the output of a command, but just send it somewhere so that it does not interrupt you. You might do this in a background process (see subsection 3.2.3, Executing Multiple Commands Simultaneously: Background Processing) or when you only want to test a command for successful completion. You can redirect output to the special system file, /dev/**null**, which takes output that is no longer needed. Once you send output here, you cannot retrieve it.

3.1.2 REDIRECTING OUTPUT WITH >>

Symbol >> operates very much like >, but it does not overwrite the target file if it already exists; instead, it places the new output at the end of that file. You can think of symbol >> as meaning "add to the end of" or "append." With it, you can add the output of a command onto the end of a file, without overwriting the existing contents of the file. If the file does not already exist, it is created. The following command directs the system to combine files **f1**, **f2**, and **f3** and place the result at the end of whatever is already in the file named **temp**, instead of overwriting the existing contents of **temp**:

```
cat f1 f2 f3 >> temp
```

Try this command yourself with your two files, **filelist** and **combo**:

```
cat filelist >> combo
```

Then use *pg* to verify the addition to **combo**.

3.1.3 REDIRECTING INPUT WITH <

Symbol < means to take the input for a command from the file following the < instead of from the terminal. Thus, suppose you want to create a standard memo heading to automatically place at the beginning of memo files as you begin writing them, instead of typing the same heading in from the terminal each time. You can create a file named **begin** that contains the heading lines and the editing commands for inserting those lines. For example, your **begin** file could contain the following:

```
a

MEMORANDUM

DATE:

TO:

FROM:

SUBJ:

.
```

The **a** is the *ed* command to begin adding text. The next nine lines (counting blank ones) are the text to be added. The final period is the *ed* command to stop adding text. Whenever you want to type a memo, you can automatically start it with these text lines by typing:

```
ed memofile < begin
```

This places the nine text lines from **begin** at the start of **memofile**. You can then continue your editing session with *ed* to fill in the heading information and complete the memo.

You can use this same idea to create templates for batch job files that have some of the same information in them (job name, account number, and so on). You would use the template file in the same way that **begin** is used above, then adding the specific commands you wanted (to compile, load, and execute a program, for example) after the template information.

Other examples of input redirection are shown later as you learn more commands that can make use of this facility.

3.2 MULTIPLE COMMANDS

UNICOS has a number of ways in which you can combine or simultaneously run more than one command. You can string commands together to run as they normally would, but in a series one after the other. You can combine commands so each successive one uses the previous one's output, or run more than one command at the same time, or store any of these actions to run repeatedly as a command of your own.

3.2.1 EXECUTING MULTIPLE COMMANDS IN A SERIES: THE SEMICOLON

You can run commands in series, using one command line, by separating the commands with a semicolon. The shell recognizes the semicolon and breaks the line into its individual commands. The following command line executes both commands, in the order specified, before returning a prompt:

```
date; who
```

You can do this with any number of commands that you want to run in a series. If the series is longer than one line, remember to type the \ character just before pressing RETURN at the end of the line. This lets you continue the input over more than one line. (See subsection 2.2.6.4, The ' " and \ metacharacters.)

3.2.2 COMBINING COMMANDS INTO ONE: PIPES

A pipe is a way to connect the output of one program to the input of another program so that the two run in sequence as if a single command. There is virtually no limit to the combinations of commands that you can string together in this way. The only restriction is that the output of the preceding command must be suitable input for the successive command. One indication of appropriate commands are those that take input from *standard input* or write output to *standard output* as indicated on the man page that discusses them (type: man *commandname*). The following subsections are only a few examples of useful pipes.

3.2.2.1 Combining and sorting multiple files

For example, you have already seen that the following command line displays files **f1**, **f2**, and **f3**, beginning each on a new page:

```
pg f1 f2 f3
```

To order the contents of these three files (ASCII order) with the *sort* command and display the output, you could type the following command lines:

```
sort f1 f2 f3 > tempry
pg tempry
rm tempry
```

To do this more quickly and simply you could use a pipe. The pipe symbol is the vertical bar, | , and it is placed between two commands that are to be combined. For example, to achieve the same effect as the three preceding command lines, you could use the following pipe:

```
sort  f1 f2 f3 | pg
```

The | character means to take the output from the *sort* command, which would normally have gone to the terminal, and use it as input to the *pr* command to be displayed on the terminal in pages.

3.2.2.2 Searching for strings in directory listings

A pipe that is particularly helpful is the following, which searches for a file name, owner name, or group name in all directories, from the current directory down through all its branching subdirectories:

```
ls -lR | grep string
```

This gives a long listing of the contents of the current directory and all subdirectories. The **-l** option of the *ls* command specifies long listing (see 2.5.3, Permissions) and the **-R** (recursive) option (mandatory uppercase) specifies that all directories from the current one down the tree structure will have their contents listed. This long listing is then piped as input to the *grep* command which searches for *string* in it. If the specified *string* were *foo*, the output could look like this:

```
-rw-------   1   las   proj   5533   Jul   22   14:56   foo
-rwxr-xr-x   1   las   proj     41   Jan   04    9:31   foo.2
-rw-rw-rw-   1   foo   dir    2426   Nov   13   18:12   mig
drw-r--r--   1   jan   foo    4096   May   29   11:42   novel
```

Note two important things: First, the string is matched anywhere in the long listing, so it may match the file name, owner name, or group name. Second, because the **-R** option generates listings for all directories, from the one specified down to the end of the tree, doing this at the root directory or the directory of all users will probably take some time, (though it is faster than using the *find* command).

If you are only searching for a file name (not group or owner name), and you know that it is somewhere below directory **/usr/group3**, you could use the following command:

ls -R /usr/group3 | grep *filename*

Try the following useful pipe, which displays a three-column listing of the files in your current directory:

ls | pr -3t

The **-3** option to the *pr* command specifies three columns, and the **-t** option truncates the full-page listing, preventing it from scrolling off the terminal display.

3.2.2.3 Using pipes to count

You can use a pipe to determine the number of users logged onto the system at any one time, beginning with the *who* command. The *who* command lists all of the users currently on the system, displaying information about them on your screen, one user per line.

Type *who* at your terminal now just to see what the output looks like.

Obviously, if you only wanted to know the number of users on the system, this would be a clumsy way of going about it, counting the lines on your terminal as they scrolled past. The *wc* command counts the number of lines (**-l** option), words (**-w** option), and characters (**-c** option) in its input, the default being to count all three. Try it now with the file **temp** to see what the output looks like:

wc temp

If your file **temp** is identical to the one shown in subsection 2.2.2.5, you should get the following response:

6 68 333 temp

This indicates that file **temp** contains six lines; those six lines containing a total of 68 words, which comprise 333 characters.

By counting the number of lines in the output of the *who* command, you can determine how many users are on the system. A pipe accomplishes this very effectively:

who | wc -l

Type this command line now. The result should just be one number; the number of users currently on the system.

To see a count of the files in your current directory, type the following:

ls | wc -l

If you want to know how many executable files you have in your current directory, use the following command:

```
ls -l | grep "^\-..x" | wc -l
```

The ^ symbol in the pattern matches the beginning of a line. The backslash protects the – metacharacter, preventing the shell from interpreting it as the – that precedes an option. The – metacharacter itself, at the beginning of a line, matches a regular file (not directory file) in the *ls* -l output. The two periods (..) match any single character for the read and write permissions, and the x matches the user's executable permission. You may want to review the explanation of long listings in subsection 2.5.3, Permissions.

Most commands that read from the terminal can read from a pipe instead, and most commands that write to the terminal can write to a pipe instead. As many commands as desired can be connected by pipe as long as each one's output is suitable input for the following command.

3.2.3 EXECUTING MULTIPLE COMMANDS SIMULTANEOUSLY: BACKGROUND PROCESSING

The shell can simultaneously run two or more programs or commands. This is beneficial when you have a time-consuming task, such as formatting a text file or compiling a program. Running commands simultaneously lets you work on something else while waiting for the results of one or more commands.

You can run programs simultaneously by putting one or more of them "in the background" where they run noninteractively while you continue with other UNICOS tasks. The ampersand (&) command accomplishes this. By putting the & at the end of a command line, you specify that the comamnd line's action is to be performed in the background. This is called *background processing*.

The following are examples of three separate legal command lines that initiate background processes (*nohup* is explained in the following paragraph):

```
nohup who > whofile &
nohup who | grep jill &
nohup date > listing; ls -C >> listing; pwd >> listing &
```

The *nohup* command automatically stops the background process (following it on the command line) and saves any output in a file, **nohup.out**, if you log out (or the system goes down) before the background process is complete. For more information, see the *nohup* entry in the UNICOS User Commands Reference Manual, publication SR-2011.

The first creates a file, **whofile**, listing all users currently logged on the system. The second searches for user *jill* in the output of who to see if she is logged on the system. The third puts the date into a file, **listing**, adds a list of the current directory's files, and then adds the path name of the directory at the end of the **listing** file.

The three preceding command lines are simple examples, but they are really too brief in execution time to take advantage of the background feature; more complex examples follow.

3.2.3.1 Example: Background processing an editing job

Another instance in which background processing can be useful is if you have a file (**text2**) of text and a file (**edits**) of editing commands to make changes to the text file; for example, many global replace actions that would take some time. To prevent the output from these actions from interfering with what you are doing on the terminal, name a file (**text2out**) to receive the editor's output. The command would look like this:

```
ed text2 < edits > text2out &
```

The system immediately responds with a number like the following:

```
877
```

This is the PID, *process id number*, of the backgrounded process. This is some number, different for each process, by which the system identifies the process. The 877 listed here is only an example; the numbers your system uses will be different.

Returning to the command line example, the results are as follows:

File **text2** will be changed according to the editing commands in **edits**
Editor messages or output will be in the file **text2out**
File **edits** will be unchanged

3.2.3.2 Example: Background processing a compiling job

As another example, you may have a long Fortran program to compile, requiring considerable time. Instead of waiting for it, you can start it compiling and then go on to do other work. You would type the following command:

cft77 *program*.f &

This command line begins the compilation process and puts it in the background, allowing you to do other work with UNICOS. In effect, the & tells the shell, "start this command running, then take further commands from the terminal immediately; do not wait for the first command to complete." When the compilation is complete, *cft77* automatically places the output in a file with the name *program*.o.

The .f suffix is a convention, indicating a Fortran source file under UNICOS. For more information on the *cft77* command and its options, see the *cft77* entry in the UNICOS User Commands Reference Manual, publication SR-2011.

3.2.3.3 Commands for background processing: ps and kill

There are two commands that relate to background processing, *ps* and *kill*. The *ps* command provides information about all processes that you have running. After typing the command line in the preceding example, the response from the *ps* command would look as follows:

```
PID TTY      TIME COMMAND
1779 tty47  0:12 sh
1882 tty47  0:01 sh
1883 tty47  0:01 cft77
1891 tty47  0:00 ps
```

The first *sh* is your login shell (assuming your system runs the Bourne shell, indicated by the $ prompt). The second *sh* is a subshell started for the background job initiated with the *cft77* command. This allows the background job to run with a shell, while still giving you access to a shell; you both need a shell (to interpret commands), so you both get a copy of one. This creation of a subshell happens for all commands and shell programs. It is called a *subshell* because it is created by a preceding shell and is subordinate to that preceding shell. The second (sub) shell can be terminated without terminating the first shell, but if the first shell is terminated, both shells are terminated.

The *cft77* is the background Fortran compile job, and *ps* is the *ps* command itself. The output of the *ps* command provides the following information:

Heading	Description
PID	The process id number for each process
TTY	The special file name of the controlling terminal for each process (the terminal on which you are logged)
TIME	The cumulative execution time for each process in hours and minutes.
COMMAND	The name of each command being processed

The -e option to the *ps* command (ps -e) provides this information about all processes on the system; for other users as well as yourself.

The *kill* command lets you terminate any process by specifying its PID after the *kill* command. The *kill* command terminates processes very neatly, closing any open files and taking care of other such "housekeeping." If you begin a background process and then want to stop it (for instance, you realize that it is in an infinite loop or you need to change it), type *kill*, followed by the process id number for the process/command that was returned when you started the background process (also listed in the output of *ps*). For example, to stop the cft77 compile job just listed, you would type either of the following:

```
kill  1883
kill  -9  1883
```

The PID, 1883, is from the preceding example of the *ps* command. The -9 in the second command line is an option to the kill command. It has the effect of making the *kill* command more effective in removing certain processes from execution that an ordinary *kill* will not effect.

3.2.3.4 Practice: Background processing an editing job

Create a file named **edits** containing the following editing commands. You may omit the parenthetical comments (indicated by the #), if you like; they are included here as explanatory information:

```
1s/only/55 99              #Line 1; replace only with 55 99.
1s/betic/numeric           #Line 1; substitute numeric for betic.
1,$s/ //1                  #All lines; remove first space (indicated by the /1).
2s/well/well as text       #Line 2; replace well with well as text.
4i                         #Line 4; begin insert.
This is a new line 4       #Insert this text.
.                          #End insert mode.
1,$s/2/222/g               #All lines; substitute 222 for 2.
1,$s/19/3,999/g            #All lines; substitute 3,999 for 19.
5s/ odd//                  #Line 5; remove the word odd.
$a                         #After last line; begin appending text.
This is the eighth line    #Text to append
And the ninth line (new)   #Text to append
and a new tenth line       #Text to append
.                          #End append mode
w                          #Write (save) changes
q                          #Quit the editor
```

You are now going to run this process in the background and enter other commands at the command line while it is running. Because the process is not a large one, you will want to work quickly to do the "foreground" commands before the background process completes running. Look at the following command lines, so you know ahead of time what you need to do. Once the actions are clear to you, proceed with the demonstration, omitting the parenthetical explanatory comments:

cp temp dem	(Make a copy of **temp**.)
ed demo < edits > demo2 &	(Edit demo in the background with edits commands, putting output in file demo2.)
ps	(See what processes you currently have.)
who	(See who is on the system.)
ls -l	(Get a long listing of the files in your current directory.)

When the background process is complete, you are not notified (Bourne shell only), though any output it produces is displayed on your terminal (as if executing in the foreground), unless you have redirected its output to a file. You must remember any background processes that you have and check them periodically with the *ps* command to see if they have completed. When a background process is done, it is no longer listed in the output from the *ps* command.

3.2.4 FILES OF COMMANDS: SHELL SCRIPTS

You can create files containing any of the shell commands and combinations of them, executing those files as your own commands. This is useful when you frequently use a particular command line, command sequence, or pipe. Having command lines in files lets you simply type a file name to execute the command line(s) in the file. These files are called *shell scripts*.

You can use shell scripts only in the directory in which they exist. If you move to another directory, you can access only the script files that it contains, unless you specify the absolute or relative path name to the directory in which the script is located. There are several ways around this limitation, discussed in the following subsections:

4.3.1.2 The PATH Variable (Bourne shell)
4.3.3 Shell Functions (Bourne shell)
5.3.1.2 The PATH Variable (C shell)
5.3.2 Renaming Shell Commands: The *alias* Command (C shell)

Before you write any shell scripts, know that it is not a good idea to name any of your shell scripts with the same names as UNICOS commands, and you cannot use the name **test** (it has a special meaning in the UNICOS Bourne shell). In certain cases, trying to use a shell script with the same name as a system command can cause infinite recursion, which will lock up your terminal and require intervention by your system administrator. If you are not sure if a name you intend to use belongs to a system command, you can use the *type* command to check. If your prompt is %, type *sh* before using the *type* command and type *exit* when you are done (for more information about this, see subsection 3.7, The Two Shells: Bourne Shell and C Shell). Use the *type* command as follows, where *name* is the name you want to check:

type *name*

If *name* is not used by the system for any commands, you will get the following response, and know that you can safely use *name* for your script:

name not found

If you want to use the name of a system command for a script of your own, you should specify the absolute path names to any of the system commands that you use within the script. To find the absolute path name to a system command, use the *type* command as follows, the response is indicated in italics (try this example):

```
type ls
ls is /bin/ls
```

If the response tells you that the command is a *shell builtin*, you do not need to specify an absolute path name for the command in your script.

As an example of a use for a shell script, suppose you frequently want to use the *ls* command with its -C and -F options, to get output in columns and to indicate which entries are files and which are directories. You might also want to add a blank line before and after the output, to make a neater display on the terminal. Create the following file, naming it **lf**, then read the paragraphs of explanation following it:

```
echo
ls -CF
echo
```

The *echo* command, on the first and third lines of the file, does just what its name suggests; it echoes (displays on the screen) the arguments given it. If it has no text to echo (as in this case), it displays a blank line. The second line of the file contains the *ls* command and the options that you want. The third line of the preceding script simply produces another blank line of output.

You cannot yet use this file as a command, however. Try to now to see what message you get; type the following:

```
lf
```

You should get the following response:

```
lf:  cannot execute
```

Before you can use this file as a command, you must give it execute permission. (Subsection 2.5.3, Permissions, discusses this.) If you want to give yourself all permissions and other users only read and execute permission, type the following command:

```
chmod  755  lf
```

Now type:

```
lf
```

You should get a listing of all the files in your current directory. If you want the command to indicate what the current directory is, add the *pwd* command to the file as follows:

```
echo
pwd
ls -CF
echo
```

Once you have made a file executable, it remains that way when you make changes to it; you do not need to change the permissions when you change the file. Thus, if you added the *pwd* command to your *lf* command file, you would not need to use the *chmod* command afterward to again make the file executable.

3.3 COMMUNICATING WITH OTHER USERS

You can use the *mail* and *write* commands to communicate with other system users. The *mail* command lets you send a message to one or more users, and the receiver can read the message at any time. The *write* command lets you have an interactive "conversation" with another user through your terminals. The following subsections describe these two commands.

3.3.1 THE mail COMMAND

UNICOS provides a postal system so that you can communicate with other users of the system. Consequently, you may sometimes get the following message, when you log in, or during a session on the system:

```
You have mail.
```

To read your mail, type the following command:

```
mail
```

The system displays your mail one message at a time, the most recent message first. After each message, *mail* waits for you to respond. The two basic responses are *d*, which deletes the message, and RETURN, which does not (so it will still be there the next time you read your mail). Other responses are described in the *mail* entry in the UNICOS User Commands Reference Manual, publication SR-2011.

The following example shows you how you can send mail to another user. Assume that *jones* is the login name of another user on the system. The easiest way to send mail to *jones* is as follows:

```
mail jones
Are you available this afternoon    (Type the text on as many lines as you like.)
at 3:00 to go over the schedule?
smith
.
```

To end the message, type a period on a line by itself, then press RETURN.

For practice, send mail to yourself. (This is not as strange as it might sound; mail to oneself is a handy reminder mechanism.)

There are other ways to send mail. For example, you can use *ed* to prepare a letter in a file named let. The contents of file let could then be sent to several people as follows:

```
mail adam mary joe < let
```

For more details, see the *mail* entry in the UNICOS User Commands Reference Manual, publication SR-2011. See also, the *mailx* entry in the UNICOS User Commands Reference Manual, publication SR-2011, for a more complex electronic mail command with more features.

3.3.2 THE write COMMAND

NOTE

Before using the *write* command, be certain that you know what the EOF character is for your terminal. Often it is CONTROL-d, but check with your system administrator. The EOF is your only way of quitting the *write* command.

At some point, you may get a message like the following on your terminal:

```
Message from jones tty07
```

A beep may accompany the message, depending on the type of terminal you are using. This message and beep mean that the user whose login name is *jones* wants to talk to you, but unless you take explicit action, you will not be able to talk back. The message may appear to clutter whatever you are working on, but it is only on your screen; if you are editing a file, the message does not get entered into that file.

To respond to the message, type the following command:

```
write jones
```

This establishes a two-way communication path. Now, whatever Jones types on her terminal appears on yours, and vice versa. The path is slow because it is limited by your typing speed, and because no characters are sent until you press RETURN. If you are in the middle of a command or editing session, you have to get to the shell prompt before you can type the command to respond. Normally, whatever program you are running has to terminate or be terminated. If you are editing, you can escape temporarily from the editor (see subsection 2.2.2.2, Editing Files with ed Commands).

The following example shows the typical protocol used to keep messages from each person separate:

1. Jones types "write smith" and waits.
2. Smith types "write jones" and waits.
3. Jones now types a message (as many lines as desired). When she is ready for a reply, she signals it by typing the letter o (which stands for "over," as in radio communications) on a separate line.
4. Now Smith types a reply, also terminated by typing the letter o.

This cycle repeats until the messages are complete and Jones signals an intent to quit with oo (for "over and out").

To terminate the conversation, both Smith and Jones type a CONTROL-d character at the beginning of a line. (The interrupt key also works.) When one user types CONTROL-d, the message EOF appears on the other user's terminal.

If you do not want to be interrupted by *write* messages, you can type the following command to suppress receiving any messages:

```
mesg n
```

To reinstate the ability to receive messages, use the following command line:

```
mesg y
```

If you write to someone who is not logged in or who does not want to be disturbed, you get the message "*loginname* is not logged on." If you write to someone who is logged in but who does not respond after a reasonable interval, type CONTROL-d. For additional information, see the *write* entry in the UNICOS User Commands Reference Manual, publication SR-2011.

3.4 DELAYING EXECUTION OF SHELL PROGRAMS

If your system administrator has enabled the *at* command for your login, you can use it to execute commands at a later time and date. The command format is as follows:

```
at time date + increment
```

Indicate the time as follows:

- Specify 1, 2, or 4 digits. The system interprets 1- and 2-digit numbers (such as 6 or 06) as hours, 4-digit numbers (such as 0630) as hours and minutes. Alternatively, you may express hours and minutes as 3 or 4 digits with a colon; for example, 6:30.
- Append the optional suffix am or pm. If you do not include am or pm, the system assumes a 24-hour clock.
- Use the following words in place of numbers if you like: noon, midnight, now, or next. Use a day, date, or month after next.

The date is optional.

- It may be any of the following expressions: A month and day, such as Jan 24
- A month, day, and year, such as Jan 24, 1986
- A day, such as Monday, or the three-letter abbreviation, Mon
- The word today or tomorrow

If you do not specify a date, UNICOS assumes today if the hour is greater than the current hour or tomorrow if the hour is less. If you do not include a year, UNICOS assumes the present year if the month is after the current month or next year if the month is before.

An increment is optional. It consists of a +, a number, and one of the following units: minutes, hours, days, weeks, months, or years (for example, + 2 weeks).

The following commands are legitimate:

- at 0815 Jan 24
- at now + 1 day
- at 5 pm Fri

You can specify more than one command with the *at* command. On one or more lines following the *at* command line specify the commands to be executed, as follows, ending with CONTROL-d:

```
at 7:30 Mon
command
.
.
.
CONTROL-d
```

As with any UNICOS command, the *at* command and the actions it is to perform can be placed in a file and executed repeatedly as a *shell script*.

3.5 FORTRAN PROGRAMS UNDER UNICOS

This subsection very briefly covers information necessary to use Fortran programs under UNICOS, including file-naming conventions, loading, compiling, linking, flowtrace, and libraries.

The Fortran compiler used in these examples is the Cray Fortran compiler CFT77 , based on the American National Standards Institute (ANSI) standard X3.9-1978, often called Fortran 77. CFT77 supports extensions to this standard, to offer broader capabilities and to take advantage of the features of Cray supercomputers.

3.5.1 FORTRAN FILE-NAMING CONVENTIONS

Fortran files should be named according to the UNICOS file-naming conventions. Names must begin with a letter and after that, may contain any sequence of letters or numerals up to a maximum of 14 characters. By convention, Fortran files should end with the suffix .f.

3.5.2 COMPILING, LOADING, AND EXECUTING FORTRAN PROGRAMS

To compile a Fortran program, use command line that follows, where *filename* is the name of your file, with the suffix .f. (If you have a working Fortran program, use it with these steps.)

```
cft77 sourcefile.f
```

If your program is very large and complex, you may want to put this process in the background and continue with other UNICOS tasks (see subsection 3.2.3 Executing Multiple Commands Simultaneously: Background Processing).

When the shell prompt reappears (if you did not put the process in the background), the compilation is done. The compiled file has the same prefix name as your input file, with a .o suffix instead of the .f suffix. In the preceding example, the compiled output file would be as follows:

sourcefile.o

The next step is to load this file with the segment loader:

segldr -o *execfile* *sourcefile*.o

The executable output of this command is placed in the file you name in **execfile** with the **-o** option. To execute your program, type that name:

execfile

This is for the case in which you explicitly specify all input and output data files by name within the Fortran code.

If you use the default output unit (used by the Fortran statements PRINT and WRITE(*)), use output redirection with the name of the output data file on the command line:

execfile > *outdata*

If you use the default input unit in your Fortran code, use input redirection with the name of the input data file on the command line:

execfile < *datafile*

If you use both default input and default output units, use the following command line:

execfile < *indata* > *outdata*

These directions are for some of the simpler cases of compiling and loading Fortran programs. There are many options available with the *cft77* and *segldr* commands that let you use special features and optimize those processes and your own program. For a complete discussion of these commands and their options, see the following publications:

- *cft77* and *segldr* entries in the UNICOS User Commands Reference Manual, publication SR-2011
- CFT77 Reference Manual, publication SR-0018
- Segment Loader (SEGLDR) Reference Manual, publication SR-0066

A complete shell program for performing all of these functions is in subsection 4.1.5, A Sample Shell Script to Compile, Load, and Execute Program Files.

3.5.3 LINKING UNICOS FILES TO FORTRAN LOGICAL UNITS

UNICOS automatically makes any file premanent when it is opened within a Fortran program.

I/O unit numbers can be in the range 0 through 101. Unit 0 is preconnected to file **stderr**, where error and infomative messages are written by executing processes. An asterisk (*) used as a unit identifier number specifies unit 100 for reading and unit 101 for writing; these units are always connected to the files **stdin** and **stdout**, respectively, and cannot be reassigned. Units 5 and 6 are also preconnected to files **stdin** and **stdout**, but you can change these assignments with OPEN statements, so they are not equivalent to an asterisk identifier.

You can redirect **stdin** and **stdout** to and from other files, so that I/O statements using the * unit identifier can indirectly access different files. For example, the command line, *pgm < infile > outfile*, makes READ(*) read from **infile** and WRITE(*) write to **outfile**.

In addition to unit connections established by the OPEN statement, you can establish an I/O unit *nn*, using the UNICOS *ln* (link) command as follows: *ln filename fort.nn*. Fortran I/O statements can then access unit *nn* without a previous OPEN statement. The *ln* command works only within one file system. The link is permanent, so to remove the file, you must remove both the original name and the alias name. An example for file *infile* and I/O unit 8 follows:

```
ln infile fort.8          (This allows WRITE(8) and READ(8) with no OPEN statement.)
```

3.6 PASCAL, C, AND CAL PROGRAM FILES UNDER UNICOS

The following subsections show simple compile, load, and execute instructions for Pascal, C, and CAL programs on Cray computer systems running UNICOS. A complete shell script for performing all of the functions shown here is in subsection 4.1.5, A Sample Shell Script to Compile, Load, and Execute Program Files.

3.6.1 PASCAL PROGRAM FILES

To compile a pascal source file, use the command line that follows. (If you have a working Pascal program, use it with these steps).

```
pascal -i sourcefile.p
```

The **-i** option is required for you to be able to specify the source file name, *sourcefile*.**p**. There are many other options to the *pascal* command that provide other capabilities and optimizations. See the *pascal* entry in the UNICOS User Commands Reference Manual, publication SR-2011 for a more complete description of the *pascal* command and its many options.

The *pascal* command places the compiled code in a file named **a.o**. To load this file, use the following command line:

```
segldr -o execfile a.o
```

The **-o** option lets you specify the name of the file to which the executable output is to be sent. To then execute this file, type the following:

```
execfile
```

If your program requires a data file as input, use input redirection with the name of that data file on the command line:

execfile < datafile

These directions are for some of the simpler cases of compiling and loading Pascal programs. There are many options available with the *pascal* and *segldr* commands that let you use special features and optimize those processes and your own program. For a complete discussion of these commands and their options, see the following publications:

- *pascal* and *segldr* entries in the UNICOS User Commands Reference Manual, publication SR-2011
- Pascal Reference Manual, publication SR-0060
- Segment Loader (SEGLDR) Reference Manual, publication SR-0066

3.6.2 C PROGRAM FILES

Compiling C programs is accomplished with the command line that follows. (If you have a working C program, use it with these steps.)

cc *sourcefile*.c

The name of the C program source file is *sourcefile*.c. The C compiler automatically produces an executable file, **a.out**. To execute your program, type that name at the shell prompt:

a.out

To specify another name for the executable file, use the **-o** option of *cc* and a file name:

cc -o *execfile sourcefile*.c

The *execfile* argument is the name of the file receiving the executable output. To execute your program, type that name at the shell prompt as follows:

execfile

If your program requires a data file as input, use input redirection with the name of that data file on the command line:

execfile < datafile

These directions are for some of the simpler cases of compiling and loading C programs. There are many options available with the *cc* and *segldr* commands that let you use special features and optimize those processes and your own program. For a complete discussion of these commands and their options, see the following publications:

- *cc* entry in the UNICOS User Commands Reference Manual, publication SR-2011
- Cray C Reference Manual, publication SR-2024

3.6.3 CAL PROGRAM FILES

To assemble a CAL source file, use the command line that follows. (If you have a working CAL program, use it with these steps.)

as *sourcefile.s*

By default, this command line places the assembled output into a file named *sourcefile.o*. You may also explicitly specify the name of the output file with the **-o** option of *as*:

as -o *assemfile sourcefile.s*

Next, input the assembled file to the segment loader, using the **-o** option to name an output file:

segldr -o *execfile assemfile*

Finally, to execute your program, type the name of the executable file produced by *segldr*:

execfile

If your program requires a data file as input, use input redirection with the name of that data file on the command line:

execfile < datafile

These directions are for some of the simpler cases of assembling and loading CAL programs. There are many options available with the *as* and *segldr* commands that let you use special features and optimize those processes and your own program. For a complete discussion of these commands and their options, see the following publications:

- *as* and *segldr* entries in the UNICOS User Commands Reference Manual, publication SR-2011
- CAL Assembler Version 2 Reference Manual, publication SR-2003
- Segment Loader (SEGLDR) Reference Manual, publication SR-0066

3.7 THE TWO SHELLS: BOURNE SHELL AND C SHELL

The shell is a UNICOS command interpreter that translates metacharacters (discussed in subsection 2.2.6, Using Metacharacters in File Names) into lists of file names and translates <, >, and >> into changes of input and output streams. It also interprets commands and their options to initiate the appropriate actions from the operating system kernel. The shell is like an interface to the UNICOS operating system itself. Because a shell is not the operating system, but only your link to it, there can be more than one type of shell.

To this point, you have been using one shell or command interpreter, probably the Bourne shell. The Bourne shell is indicated by a default system prompt of $ and is also known as sh, which is the UNICOS command that invokes it. The other UNICOS shell is the C shell, indicated by a default system prompt of %, and known as *csh*, which is the UNICOS command that invokes it.

These two shells differ somewhat in how they interpret commands, the set of commands they offer, and in their intrinsic options. All of the commands (except background job notification) in the previous sections of this manual operate identically under either the Bourne or C shell. Section 4, The Bourne Shell, and Section 5, The C Shell, describe more detailed and advanced operations that can differ between the two shells.

Each shell can have advantages and disadvantages, depending on how you use it and for what you use it. If you skim the table of contents, you will see that sections 4 and 5 have very similar structures, covering analogous material in their subsections. To decide which shell you prefer to use, review these sections, comparing analogous subsections, to see which shell has features that best meet your needs in an operating system.

3.8 CHANGING SHELLS

Nearly all systems default to the Bourne shell on startup, giving you the $ prompt (unless your system administrator has changed the prompt, in which case ask what your default login shell is). If your system defaults to the C shell, you will see a system prompt of %. To start a C shell from within a Bourne shell, just type the *csh* command, as follows:

```
$ csh
%
```

The preceding command invokes a C shell under your current Bourne shell, while the Bourne shell continues to exist. To start a Bourne shell from within a C shell, use the *sh* command, as follows:

```
% sh
$
```

These two commands, *csh* and *sh*, in effect create another layer to the operating system. The initial shell you were in upon logging in remains; you just start another shell executing as a process under it, just as a compiling job or editing job runs as a process under that shell.

If you have the $ prompt, indicating the Bourne shell, type the following command to create a C shell:

```
csh
```

You should now see a % prompt, indicating that you are in a C shell environment.

If you have the % prompt from login, type the following command to create a Bourne shell:

```
sh
```

You should now see a $ system prompt, indicating that you are now in a Bourne shell environment.

After typing either of these commands, type the process status command to see how many processes you have in operation:

```
ps
```

If your default shell is Bourne ($), and you typed the *csh* command, you should see something like the following output:

```
  PID    TTY    TIME    COMMAND
12046   tty06   0:41    sh
14803   tty06   0:01    csh
14809   tty06   0:00    ps
```

The *sh* is your login shell; you can tell because it has the lowest PID (it's the first process started during this login session). This will not, however, always be true. Just like an automobile odometer, the PID's on your system turn over, in which case processes that you start afterwards will have lower PID's than your login shell process.

The *csh* is the C shell that you just started with the *csh* command.

Because you have two shells, when you press CONTROL-d or type *exit* to logout, you will first exit the C shell you have started, but still be in the initial login Bourne shell. Pressing CONTROL-d again (or typing another *exit*) discontinues *that* shell, logging you out.

You can, if you wish, invoke many shells, each one creating another shell layer and requiring another CONTROL-d or *exit* to get out. Each of these can be a different environment, or interface to the operating system. You can specify different characteristics for these shells, as discussed in later sections of this manual. (For the Bourne shell, see subsection 4.3.4, Shell Invocation Options. For the C shell, see subsection 5.3.4, Shell Invocation Options)

Your default login shell can be either the Bourne shell or the C shell. If, after looking over the following two sections, you prefer a different shell than your current default login shell, you can change your default login shell with the *chsh* command. Depending on the shell that you want (**/bin/sh** is Bourne shell and **/bin/new/csh** is C shell), use one of the following two command lines, where *loginname* is your login (what you type after the *login* prompt):

```
chsh loginname /bin/sh

chsh loginname /bin/new/csh
```

After typing one of these command lines, when you login you will automatically be put into the shell that you have chosen.

4. THE BOURNE SHELL

This section covers the UNICOS Bourne shell, including the following topics:

- Creating shell scripts
- Using shell parameters and variables
- Changing the shell environment
- Debugging shell scripts

4.1 SHELL SCRIPTS

In section 3, you began to write shell scripts in the Bourne shell. To go beyond such scripts to more complex ones requires using features specific to either the Bourne shell or the C shell. This section covers features of the Bourne shell that let you create more advanced and useful shell scripts, using shell commands as you would a programming language. These features include variables, control flow (if-then-else, while, for, case), and input, among others.

Now that you will be writing more complex shell scripts, it is a good idea to add comment lines to them, documenting what the scripts do. The comment character is the pound sign (#). Place it anywhere on a line to begin a comment. Once a # is encountered, the rest of the line is ignored by the shell as nonexecutable.

Examples:

```
# This a comment-only line.
ls -CF | pg  # This line contains both code and comment.
```

4.1.1 BASIC SHELL SCRIPT DEBUGGING: TRACING MECHANISMS

Before you begin to write more complex shell scripts, it is a good idea to know the tools available to help in debugging those scripts. This subsection will explain two of the tracing mechanisms that the Bourne shell provides for debugging. As you proceed to the next subsections, refer back to this discussion as you need the debugging tools. This subsection covers only the simplest of several debugging tools. Later in section 4, as your shell scripts become more complex, the other tools will be discussed (subsection 4.4, Debugging Shell Scripts). The shell has two tracing mechanisms, the -v and -x options, that you will find useful in debugging shell scripts. Both are invoked within a shell script with the following lines:

```
set -v
set -x
```

These are *options* to the *sh* (shell) command and they act like flags; they can be set on or off, and when on, they tell the shell to perform certain actions. Other shell options will be covered in subsection 4.3.4, Shell Invocation Options.

The line *set -v* causes the script to display each line before it is interpreted or executed, so you can see if there are any syntax or typographical errors.

The line *set -x* causes the script to display the interpreted version of each line before it is executed. Each such line is preceded with the plus (+) character. Write and execute the following shell script, **tracer**, to see how both of these options work:

```
set -vx
echo
ls -l *mp
echo
```

Your output should look similar to the following:

```
/bin/ls -l *mp
+ /bin/ls -l temp
-rwxr-xr-x   1 name    group          335 Mar 21 10:53 temp
```

Your login name will be in place of *name* and your group will be in place of *group*. The date and time will also be different in your output, as will the permissions, if your system sets them differently.

The following command line unsets these options:

```
set -
```

To use these options without having to edit and change the shell script itself, use the following command line where *scriptname* is the name of the shell script:

```
sh -vx scriptname
```

The preceding command line has the effect of creating a subshell with the appropriate option(s) set as an environment in which the shell script, **scriptname**, can run. The discussion of the *ps* command in subsection 3.2.3.3, Commands for Background Processing: ps and kill, contains more information about subshells.

The current setting of these shell options is stored in the variable $–.

****

CAUTION

There is an option **-n** that you can also set within shell scripts to prevent execution of any commands following it. DO NOT type **set -n** at a terminal because this causes that terminal to ignore all commands, including any commands to log out. Only a system administrator can remedy this situation.

****

As you learn about variables and learn more about metacharacters in this section, try these options (particularly option **-x**) with more complex shell scripts, to see more exactly what their output is in different situations.

4.1.2 VARIABLES IN SHELL SCRIPTS

As with any programming language, the Bourne shell lets you use variables to contain information. These variables come in two basic types: predefined variables and user-defined variables. This subsection covers user-defined variables, which can be either named variables or command-line positional variables.

4.1.2.1 Named variables

Named variables in the shell do not need to be declared before use as they do in some programming languages (such as Pascal). Whenever you first use the name of a new variable, that variable exists, having a default null value. Variables may be any legal UNICOS name which consists of 1 to 14 alphanumeric characters. UNICOS distinguishes between uppercase and lowercase, so the variables *name* and *Name* are different. To set a variable, use the following syntax:

variable=value

No spaces are allowed on either side of the equal sign (=).

To access the value of a variable, use a dollar sign ($) in front of the variable's name as follows: *$variable*.

NOTE

Variables are correctly interpreted inside of double quotes, the value being correctly substituted for the variable name. Inside of single quotes, however, the string is interpreted literally as a $ with other characters after it.

Make a simple shell script now to try using variables. Create the following file, **carfile**:

```
car=Ferrari
driver=Mathilda
tires=Pirelli
echo $car
echo $driver
echo $tires
echo
echo "$driver drives a $car with $tires tires"
```

Save this file, then give it execute permission (see subsection 2.5.3, Permissions) so you can execute it as a program. Now, at the shell prompt, type the following:

```
carfile
```

You will get the following response:

```
Ferrari
Mathilda
Pirelli

Mathilda drives a Ferrari with Pirelli tires
```

You can also assign values to variables as input to a shell script with the *read* command. Write the following shell script, **car2**, and then run it:

```
echo
echo "Please enter the name of the driver:"
read driver
echo "Please enter the make of car:"
read car
echo "What sort of tires:"
read tires
echo
echo "$driver drives a $car with $tires tires."
```

4.1.2.2 Availability of variables: Scoping rules and commands

The *export* command lets you use the value of a variable in a shell other than the one in which it is defined. Recall from the discussion of the *ps* command (subsection 3.2.3.3) that subshells are created for most commands and all shell scripts that are executed. Therefore, such commands are not executing in your current shell and cannot use the values of variables declared in your current shell. To see this, write the following shell script, **scope:**

```
var1=one
var3=three
echo
echo "var1 equals $var1"
echo "var2 equals $var2"
echo "var3 equals $var3"
echo
```

Now type the following commands at the command line:

```
$ var1=111
$ var2=222
$ var3=333
$ export var2 var3
```

Now execute **scope.** You should get the following results:

```
var1 equals one
var2 equals 222
var3 equals three
```

Because var1 was not exported from the login shell, its value of 111 is not available to the script **scope,** executing in a subshell. var2 was exported from the login shell so **scope** echoes its login shell value. var3 was exported from the login shell, but it was reassigned a new value in the subshell by **scope,** so **scope** echoes its new subshell value. This does not change the value of var3 in the login shell, as you can verify by typing the following command line after running **scope** (output indicated in italics):

```
$ echo $var1 $var2 $var3
111 222 333
```

This shows that exporting variables is a one-way process; variables can be exported down to subshells which can receive the values and change them locally, but subshells cannot export the changed values up to the shell in which the variable was defined.

If you want variables that can only be referenced and not changed, you can define variables as read-only, with the *readonly* command, where *name* is the variable name:

readonly *name*

Once a variable has been set to a value and declared read-only, its value cannot be changed. When you log out and log back in, however, the variable will be gone just like all others.

You can prevent the creation of subshells to execute commands, by using the period (.) command. The . forces the command or shell script following it to execute in the current shell. Execute **scope** with the . command:

. scope

The results will not appear to be any different than before, until you echo the values of the login-shell variables:

$ echo $var1 $var2 $var3
one 222 three

Because **scope** executed in the current shell--the shell in which the variables were defined--it changed their defined values.

Another way to give variables specific values in commands or shell scripts, without changing the variable's value in the current shell, is with the following syntax:

name=value command

name is the variable name and *value* is its new value for use within the shell script, *command.* Variables assigned in this way are called *keyword parameters.*

Use the *echo* command at the command line to see what the current values are of variables, $var1, $var2, and $var3. Then call **scope** with the following command line:

var1=AAA var2=BBB var3=CCC scope

You will get the following response:

var1 equals one
var2 equals BBB
var3 equals three

Assignment of the variables within the script still temporarily overrides the values they are given at the invoking command line. If they are not reassigned in the script, however, (as var2 is not) they have the values given them on the invoking command line. Now again echo variables var1, var2 and, var3 at the command line to verify that their values in the shell in which they were defined are the same as they were before and have not been changed to AAA, BBB, and CCC.

4.1.2.3 Command-line positional variables

Command-line positional variables are variables that automatically exist in the shell. These variables are the numerals 0 through 9, and the shell sets them to the values of the words on a command line, according to the order of the words. Look at the following command line and then at the breakdown of it in the next paragraph:

ls -CF bookdirectory

In the preceding command line, the first word, the *ls* command, goes into the variable 0. The second word, *-CF*, goes into the variable 1. The third word, *bookdirectory*, goes into the variable 2. Because there is this association by order or position on the command line, these variables are know as *positional parameters*. The positional parameter 0 is unlike the others (1 through 9), in that it always is set (automatically by the shell) to the value of the command on a command line. You cannot explicitly set it to anything else.

To access the values in positional parameters, place a $ before the numeral, just as you would access the value of a named variable.

To set the positional parameters 1 through 9, substitute the appropriate numeral into the command line in a shell script for the argument that you want it to take on as a value. For example, in the preceding command line, to put the value of the directory to be listed into a positional parameter, you would use the following line in a shell script:

```
ls -CF $1
```

Write the following shell script, **parms**, to demonstrate these concepts: (Please copy it exactly, you will be using it again later.)

```
echo "The first parameter is:  $1"
echo "The second parameter is:  $2"
echo "The third parameter is:  $3"
```

Change the file's permission so you can execute it (see subsection 2.5.3, Permissions), then try it with the following input (the $ characters should be your shell prompt):

Input	**Response**
$ parms one	The first parameter is: one The second parameter is: The third parameter is: $
$ parms one two	The first parameter is: one The second parameter is: two The third parameter is: $
$ parms one two three	The first parameter is: one The second parameter is: two The third parameter is: three $
$ parms "one two" three	The first parameter is: one two The second parameter is: three The third parameter is: $

As another example, create the following shell script, **parms2**:

```
echo "The command is:  $1"
echo "The first argument is:  $2"
echo "The second argument is:  $3"
$1 $2 $3
```

Make the file executable, then use it as follows, examining the output carefully:

```
parms2 ls -CF .
```

Try it again with the following input:

```
parms2 grep d carfile
```

The two preceding examples demonstrate that you can use the contents of these positional parameters (or of named variables) as either data or as executable commands. You can assign to variables the full path names of executable files. You are encouraged to explore this flexibility further with your own exercises and experimental shell scripts.

Create the following shell script for your own use, naming it **x**:

```
chmod 755 $1
```

This shell script, **x**, changes the permissions of the file argument you give it, to make that file executable for everyone, readable by everyone, and writable only by you. Use it as follows:

```
x filename
```

4.1.2.4 More than nine positional parameters: The shift command

The shell keeps track of all arguments on the command line, even when there are more than the nine that can be held in positional parameters. There are no names for the arguments greater than nine, but you can access them by moving them into the nine positional parameter variables. The *shift* command accomplishes this by moving the values of all the positional parameters down one number and discarding the value of $1. Therefore, you can operate on more than nine parameters by using them sequentially, discarding each as you use it. This lets you design a shell script to iterate over an unknown number of arguments. Write the following shell script, **tenparms**, to see how this works:

```
echo $1; shift
echo $1; shift
echo $1; shift
echo $1; shift
echo $1; shift
echo $1; shift
echo $1; shift
echo $1; shift
echo $1; shift
echo $1
```

Now execute the script, calling it with the following arguments:

```
$ tenparms one two three four five six seven eight nine ten
```

You can specify a numeric argument with the *shift* command to shift more than one positional parameter. Try the following shell script, **moreparms**:

```
echo $1
shift 3
echo $1
shift 3
echo $1
shift 3
echo $1
```

Invoke it with the following command line:

```
$ moreparms one two three four five six seven eight nine ten
```

You should get the following response:

```
one
four
seven
ten
```

4.1.2.5 Special command-line variables

This subsection discusses five special variables, automatically set by the shell, that are related to those in the previous subsection. The first of these special parameters, **$0**, is a variable that always contains the name of the command currently executing. In your login shell, it contains the name of the shell you are running (sh or csh), because your login shell is a continuous process for as long as you are logged on. To see this, just type the following command at your terminal:

```
echo $0
```

Another special variable automatically set by the shell is **$#**. This variable contains the number of positional parameters typed on a command line and can be used in shell scripts to count arguments to a script.

Add the following line at the end of your shell script **parms**:

```
echo "The number of parameters is:  $#"
```

Now run the script with the same data as you did in subsection 4.1.1.2, looking carefully at the output.

The special shell variable **$*** takes on the values of all the positional parameters, except **$0**. Add the following line to the end of your shell script **parms**:

```
echo "All arguments are:  $*"
```

Run **parms** just as you did before, examining the results. The **$*** variable lets you apply a command to more than one argument. Use it again to alter your shell script **x**. Substitute **$*** for **$1**, so the file looks like the following one:

```
chmod  755  $*
```

Shell script x will now change the permissions to 755 for any number of files you give it, making all of them executable. Use it as follows:

x *filename1 filename2 ... filenamen*

The next special shell variable is $!. This variable contains the process ID number of the last process that you put in the background during the current login session. It is an easy way to get the number of the process if you want to see its status or kill it:

```
ps -p $!
kill $!
```

See subsection 3.2.3 for more information on background processing.

The last special shell variable that this subsection will discuss is $$. This variable contains the process id number (PID) of the current shell. Because every command and shell script gets its own subshell when it executes (subsection 4.1.2.2, Availability of Variables: Scoping Rules and Commands), this number is unique to each invocation of a command or shell script. Therefore, this number is often used to name temporary output files. That way, successive uses of a command/script do not overwrite previous output files. Use it as follows:

```
who | grep mygroup > groupfile.$$
```

This gives you a different file name, groupfile.*number* than another user gets from using the same command line, the difference being *number*. You also get a unique number for $$ each time that you run a shell script with a $$ in it.

4.1.3 CONTROL FLOW

The Bourne shell has the following five control flow constructs basic to programming languages: test condition, for, while, if-then-else, and case. The following five subsections discuss the Bourne shell's treatment of these constructs.

4.1.3.1 Evaluating conditions: The test command

Before you can do any conditional programming, you must be able to test conditions. The *test* command lets you check many different conditions. The general conditions that you can test for are as follows:

- File information such as existence, size, permissions, and type
- Variable and string information such as length, equality, and relational operators
- Numeric comparison with the relational operators such as = > < and so on
- Test combinations with the logical operators AND, OR, and NOT

The *test* command can take either of the following two formats:

test *expression*
[*expression*] (Spaces around the brackets are required)

Many people prefer the latter syntax for use within conditional constructs (such as *if* and *case*) because of its brevity.

The logical operators of the *test* command are as follows:

test *expression* -a *expression*	Logical AND
test *expression* -o *expression*	Logical OR
test !*expression*	Logical NOT

The conditions for files are as follows:

-b *file*	True if *file* exists and is a block special file
-c *file*	True if *file* exists and is a character special file
-d *file*	True if *file* exists and is a directory file
-f *file*	True if *file* exists and is a regular file
-g *file*	True if *file* exists and the set-group-ID bit is set
-p *file*	True if *file* exists and is a fifo special file (named pipe)
-r *file*	True if *file* exists and can be read by the user making the test
-s *file*	True if *file* exists and has a size greater than zero
-u *file*	True if *file* exists and the set-user-ID bit is set
-w *file*	True if *file* exists and can be written to by the user making the test
-x *file*	True if *file* exists and can be executed by the user making the test

You can combine commands with tests, using the && metacharacter. Putting the && metacharacter after a *test* command and before another command will execute the second command only if the test returns a true value. To see how this works, type the following command lines at your terminal, observing their output:

```
$ test -f carfile && echo "carfile exists"
$ test -x carfile && echo "carfile is executable"
$ test -d carfile && echo "carfile is a directory"
$ test -w parms2 && echo "parms2 is writable"
$ test -w / && echo "I can write to the root directory"
```

The || metacharacter has the opposite effect, executing the second command only if the test returns a false value. Try the following command lines, comparing them with the preceding five:

```
$ test -f carfile || echo "carfile does not exist"
$ test -x carfile || echo "carfile is not executable"
$ test -d carfile || echo "carfile is not a directory"
$ test -w parms2 || echo "parms2 is not writable"
$ test -w / || echo "I cannot write to the root directory"
```

Now try using the *test* command's logical operators -a, -o, and !:

```
$ test -f carfile -a -w carfile && echo "carfile regular and writable"
$ test -x parms2 -a -d parms2 || echo "parms2 is not an executable directory"
$ test -r / -o -x / && echo "root is readable or executable"
$ test !-w / && echo "root is not writable"
$ test !-f carfile && echo "carfile not regular"
```

The test commands for strings/variables are as follows:

-z string	True if the length of the string is zero
-n string	True if the length of the string is nonzero
string	True if string is not null
string1 = string2	True if *string1* is equal to *string2*
string1 != string2	True if *string1* is not equal to *string2*

To try out these tests, type the following series of commands at your terminal:

```
$ name=kelly ; animal=bat ; person=kelly
$ echo $nothing

$ test $name = "kelly" && echo "true"
$ test $name = $person && echo "true"
$ test -n $name && echo "true"
$ test -z $nothing && echo "true"
$ test $person != $animal && echo "true"
```

4.1.3.2 Numeric tests and expressions

The tests available for numeric values are as follows:

num1 -eq *num2*	True if *num1* is equal to *num2*
num1 -ne *num2*	True if *num1* is not equal to *num2*
num1 -gt *num2*	True if *num1* is greater than to *num2*
num1 -ge *num2*	True if *num1* is greater than or equal to *num2*
num1 -lt *num2*	True if *num1* is less than to *num2*
num1 -le *num2*	True if *num1* is less than or equal to *num2*

The numeric expression evaluator is the *expr* command and it recognizes the basic arithmetic operators, + – * / as well as a remainder operator (modulus), %. The syntax of *expr* is as follows:

expr *expression*

You can use command substitution (subsection 4.2.1, Substituting a Command's Output for Other Shell Values) to assign the value of an expression to a variable. This is useful for operations such as incrementing or counting. Try the following example at the shell prompt:

```
$ counter=4
$ echo $counter
4
$ counter=`expr $counter + 1`
$ echo $counter
5
```

4.1.3.3 Branching on one condition: The if command

The *if* command lets you test one condition and perform a command or series of commands if the condition evaluates to true. All of the conditions of the *test* command may be used in the *if* construct.

The simplest format of the *if* construct is as follows (note the bracket format of the *test* command):

Format-1	**Example**
if *condition*	if [-f carfile]
then	then
command list	echo "# Exists and is regular" >> carfile
fi	fi

Try out the example on your system, then substitute other file and directory names for **carfile**, to see the results.

The *if* and *condition* must be together on one line by themselves. If *condition* is longer than one line, use a protected new line (backslash before the RETURN) to start the second line. The *then* must be on a line by itself. Any number of commands may follow the *then*, either on separate lines or separated by semicolons. Pipes of commands, I/O redirection, and any other valid shell constructs or functions are also allowed in this command list. The *fi* ends the *if* construct. It must be the last line of the *if* construct and must be on a line by itself.

Two other formats of the *if* construct follow. (Note the bracket form of the *test* command.)

Format-2	**Example**
if *condition*	if [-w /]
then	then
command list$_1$	echo "I can write to root directory" > fileR
else	else
command list$_2$	echo "I cannot write to root directory" > fileR
fi	fi

Format-3	**Example**
if *condition*$_1$	if [-b unknown]
then	then
command list	echo "This is an existing block special file"
elif *condition*$_2$	elif [-c unknown]
then	then
command list	echo "This is an existing block special file"
...	*(test the other file conditions, d through w, with elif sequences)*
elif condition$_n$	elif [-x unknown]
then	then
command listn	echo "This is an existing file for which I have execute permission"
else	else
command list$_m$	echo 'File "unknown" does not exist.'
fi	fi

Try the preceding examples on your system, then substitute other file and directory names for **carfile**, examining the results to gain a clear understanding of how the various file tests work.

For further practice, try using variable and numeric tests with these *if* command formats.

4.1.3.4 Branching on many conditions: The case command

The *case* command provides a way to quickly and simply test a condition for many possible values.

Format:

```
case $variable in
pattern1)   command list1;;
pattern2)   command list2;;
patternn)   command listn;;
esac
```

This is a pattern-matching command, checking the pattern in *variable* against any patterns that you specify in the list following the case line, including patterns containing the metacharacters ?, *, and [ ] to match groups of similar patterns.

Create the following shell script, **month**, to print out month names, given a numeric argument:

```
case $1 in
    1)  echo "January";;
    2)  echo "February";;
    3)  echo "March";;
    4)  echo "April";;
    5)  echo "May";;
    6)  echo "June";;
    7)  echo "July";;
    8)  echo "August";;
    9)  echo "September";;
   10)  echo "October";;
   11)  echo "November";;
   12)  echo "December";;
    *)  echo "You can only use numbers between 1 and 12.";;
esac
```

The last pattern, *, matches any pattern, just as it does with the shell's filename substitution capability (subsection 2.2.6, Using Metacharacters in File Names), therefore; * acts as the default case when either nothing, or something not in the list of case choices is specified. Give this script executable permission, then test it with command lines like the following:

```
$ month 11
$ month 13
```

The following shell script, **language**, uses metacharacters to determine if the first argument it is given (**$1**, a file name) is a Fortran, C, or Pascal source file. The naming conventions given in subsection 2.2.1, File-Naming Conventions, must be followed for the file names.

```
case $1 in
    *.f)    echo "Fortran source file";;
    *.c)    echo "C source file";;
    *.p)    echo "Pascal source file";;
     *)     echo "unknown";;
esac
```

You can use the *case* command to set up options for your shell scripts. For example, you can write a shell script that prints out the current list of system users in various formats. Write the following script, **menu**:

```
# -c displays calendar for current year
# -d displays the date and time
# -l provides a listing of the current directory
# -q quits the shell script
# -w provides a list of who is currently on the system
# Anything other than the above options terminates the program
case $1 in
    -c)   echo "What is the current year?"
          read year
          cal $year | pg;;
    -d)   date;;
    -l)   echo "Directory is:"
          pwd
          ls -CF | pg;;
    -q)   exit;;
    -w)   who | pg;;
     *)   echo "The permitted options are: c, d, l, q, and w";;
esac
```

Use the script with each of its options and some incorrect ones, as follows:

```
menu -option
```

For more information on processing command-line options within shell scripts, see the *getopt* entry in the UNICOS User Commands Reference Manual, publication SR-2011.

4.1.3.5 Looping with a condition: The while and until commands

The *while* construct lets you repeat a series of commands in a shell script until a condition becomes false.

Format:

```
while condition
do
    command_list
done
```

There can be a single condition, or a compound condition using the logical AND, OR, and NOT options (-a, -o, -n) of the *test* command. The commands in the command list may be on separate lines or on one line, separated by semicolons. Pipes of commands and redirection of I/O are also permitted. The *while* and *condition* are on a line by themselves, and the *do* and *done* must by on lines by themselves, the *done* ending the *while* loop.

Write the following shell script, which executes the functions of the **menu** shell script until the user exits with the -q choice:

```
choice=null
while [ "$choice" -ne "-q" ]
do
echo "–c displays calendar for current year"
echo "–d displays the date and time"
echo "–l provides a listing of the current directory"
echo "–q quits this menu of options.  You remain logged on."
echo "–w provides a list of who is currently on the system"
echo "Anything other than the above options repeats this menu."
echo
echo "What is your choice?"
read choice
case $choice in
    -c)   echo "What is the current year?"
          read year
          cal $year | pg;;
    -d)   date;;
    -l)   echo "Directory is:"
          pwd
          ls -CF | pg;;
    -q)   exit;;
    -w)   who | pg;;
     *)   echo;;
esac
echo
done
```

Practice with the script, entering correct and incorrect options as it asks you for them. Alter the menu if you want to try other actions. You can create a menu just like this to simplify, to one keystroke, actions that you commonly perform, such as submitting jobs for compiling, loading, and executing. You can have the options preset in the shell script, or you can have it ask you for them, just as it will need to ask you for the name of the program file to compile.

The *while* command is often used with the *shift* command to operate on an unknown number of arguments until all have been processed. The following shell script is a simple example of this:

```
echo
while [ $# -ne 0 ]
do
    echo $1
    shift
done
echo
```

Try it if you like, calling it with any number of arguments. Compare the effects of this script with the one in subsection 4.1.2.4, More Than Nine Positional Parameters: The shift Command. The two are identical in what they do.

The *until* command lets you repeat a series of commands in a shell script until a condition becomes true; the converse of the *while* command. The general format of the *until* command is analogous to that of the *while* command:

```
until condition
do
    command_list
done
```

There can be a single condition or a compound condition using the logical AND, OR, and NOT options (**-a**, **-o**, **-n**) of the *test* command. The commands of the command list may be on separate lines or on one line, separated by semicolons. Pipes of commands and redirection of I/O is also permitted. The *until* and *condition* are on a line by themselves, and the *do* and *done* must by on lines by themselves, the *done* ending the *until* loop.

Contrast the following two equivalent loop conditions, either of which would work in the preceding shell script using the *shift* command:

```
while [ $# -ne 0 ]
until [ $# -eq 0 ]
```

4.1.3.6 Looping with a specified index: The for command

With the *for* command, you can repeat a sequence of commands in a shell script a specified number of times. One format of the *for* command is as follows:

```
for variable
do
    command_list
done
```

This command will perform the commands in the *command_list* as many times as there are arguments passed to the shell script, the variable taking on the value of each successive argument on each iteration through the loop. To see this, create the following shell script, **repeater:**

```
for index
do
    echo $index
done
```

Call the shell script with the following command lines:

```
repeater arg1 arg2 arg3 arg4
repeater this that "the other"
repeater you me them it
```

The script repeats the arguments with which you call it. Try the following shell script, **commander**:

```
for command
do
    $command
done
```

Call it with UNICOS commands passed as arguments, such as:

```
commander ls
commander who
commander ls who date
```

Another format of the *for* command explicitly specifies the list of indices over which to iterate:

```
for variable in word_list
do
    command_list
done
```

This will iterate *command_list* as many times as there are words in *word_list*, with *variable* successively taking on the values of those words. The following shell script performs similarly to the preceding example, **commander**:

```
for command in  ls who date
do
    $command
done
```

The *word_list* can contain variables, in which case the values of those varialbes are substituted into the word list. The following shell script shows this:

```
commandlist="ls who date"
for command in $commandlist
do
    echo $command
    $command
done
```

If the actions of the preceding shell script are not obvious on inspection, write and execute the shell script to correlate its output with the commands in it.

You can combine the *for* and *case* commands to make shell scripts that accept more than one option. Recall from subsection 4.1.3.4 the example shell script, **menu**, that was called with one option letter, each option letter having one associated action. By adding a *for* command to that shell script, you can make the script accept any number of option letters at one time, performing the appropriate action.

Modify that script, **menu**, to look like this:

```
# -c displays calendar for current year
# -d displays the date and time
# -q quits the shell script
# -w provides a list of who is currently on the system
# Anything other than the above options terminates the program
for option
do
      case $option in
          -c)   echo "What is the current year?"
                read year
                cal $year | pg;;
          -d)   date;;
          -l)   echo "Directory is:"
                pwd
                ls -CF | pg;;
          -q)   exit;;
          -w)   who | pg;;
           *)   echo "The permitted options are: c, d, l, q, and w";;
      esac
done
```

You can now call the script with any number of options (separated by spaces), and it will perform the associated actions in the order in which you specify the options:

```
menu  -l -d -w -q
```

4.1.4 SHELL SCRIPTS CONTAINING THEIR OWN INPUT: here documents

You can create shell scripts that provide some or all of the input that they require; these are called *here documents*. This is useful for operations that you perform repeatedly, such as creating files that have the same content or format. Examples of such uses are batch files that have the user ID, account number, and such, the same for each job, or memos that have standard headings.

Format:

```
command << string
    command_ input
string
```

The *command* is a UNICOS command, *command_input* is the input that the command requires, and *string* is a delimiter, not found in the input, indicating where the input begins and ends.

Example:

```
ed textfile << EOF
a
Line one of the input text for the new file, textfile.
Line two of text to make up the new text file.
Last line of input for the new file, textfile.
.
w
q
EOF
```

This inclusion of input within a command can be done either in a shell script or from the command line. An example of doing it from the command line is as follows:

```
$ mail Jeanne  << STOP

Hi, Jeanne.

There is a new shell script, phoney, on the system that creates and maintains a phone directory. I
have already found it very handy.  It is in the directory /usr/lbin.

Howard
STOP
$
```

4.1.5 A SAMPLE SHELL SCRIPT TO COMPILE, LOAD, AND EXECUTE PROGRAM FILES

You can use the concepts presented in subsection 4.1 to greatly simplify the repetitive tasks of compiling, loading, and executing program files. Writing a shell script to perform these tasks will make it easier and faster for you to do them. Such a shell script can even make it possible for users who do not know UNICOS to perform these tasks, with simple menus such as the **menu** script in subsection 4.1.3.5.

You can write the following shell script to compile, load, and execute programs, having it specify a few simple options for the procedures and asking you for names of input and output files. This example shell script is for Fortran programs, but it can easily be adapted to call other language processors, or give you options to select among language processors. Read through the shell script carefully until you understand how it works, then read the suggestions that follow it to see how you can further tailor the script to your particular needs.

```
echo
echo "What is the name of your source code file?"
read sourcefile
cft77  -a stack  $sourcefile
echo "What do you want the name of your executable file to be?"
read execfile
segldr  -o $execfile  $sourcefile.o
echo "Does your program require an input data file (Y/N)?"
read answer1
if [ $answer1 = "Y" ]
then
  echo "The name of the input data file?"
  read indata
fi
echo "Does your program require an output data file (Y/N)?"
read answer2
if [ $answer2 = "Y" ]
then
  echo "The name of the output data file?"
  read outdata
fi
if [ $answer1 = "Y" ]
then
  if [ $answer2 = "Y" ]
  then
    $execfile < $indata > $outdata
  else
    $execfile < $indata
  fi
elif [ $answer2 = "Y" ]
then
  $execfile > $outdata
else
  $execfile
fi
```

You can enhance this shell script so that more options are included in the compiling and loading commands, so that it asks the user for different options to those commands, and/or it asks the user for the language of the source file to invoke the appropriate compiler/assembler.

4.2 SHELL PARAMETERS AND VARIABLES

This subsection discusses more complex details of the ways that the Bourne shell uses and interprets variables and parameters. Previous discussions have focused more on the use of variables; this subsection explains some of the concepts behind such usage.

4.2.1 SUBSTITUTING A COMMAND'S OUTPUT FOR OTHER SHELL VALUES

The accent grave metacharacter (`) lets you use the output of a command in a number of ways. You can store that output in a variable or use it as as input to another command or shell script. To store the output in a variable, use the following syntax:

variable=`command`

Storing the output of a command in a variable is handy when you will want to use that output repeatedly. Having the output in a variable requires less typing on your part and reduces the execution time of shell scripts over having the information in a file or repeatedly executing the command.

If, in a shell script, you want to use the date and time in several places, you can use the following line to place the output of the date command into the variable *d* and then use $d for the information:

d=`date`

To use the output of a command as input to another command or shell script, use the following syntax:

command1 `command2`

The command (*command1*) receiving the output can be a simple command, a pipe, or any other legal UNICOS command line.

As an example, suppose you have a file, **maillist**, containing the login names of several people to whom you regularly send mail on the system. If you discover some information you want to send them, you can use the following command line:

mail `cat maillist`
your message
CONTROL-d

4.2.2 SUBSTITUTING VALUES FOR VARIABLES

User-defined variables (subsection 4.1.2.1, Named Variables) can be assigned alternate or default values and can be used to produce new values. The special characters used to do this are braces, { }.

If a variable has not yet been assigned a value, or it has been assigned the null string (''), you can test for this, and then give the variable a default value in the same command line. This command has two general formats, producing identical results:

${*variable:-value*}
${*variable:=value*}

Try the following examples at the command line to see how this works (responses are given in italics):

```
$ car=''
$ echo "The car is a ${car:=Mercedes}"
The car is a Mercedes
$ echo $car
Mercedes
$
```

```
$ car="Ford"
$ echo "The car is a ${car:=Mercedes}"
The car is a Ford
$ echo $car
Ford
$
```

Another version of variable substitution lets you test a variable to see if it has a value and then reset it to a new value, leaving it null if it is currently null or has not been set at all. The general format for this is as follows:

```
${variable:+value}
```

Try the following examples at the command line to see how this works (responses are given in italics):

```
$ car=''
$ echo "The car is a ${car:+Mercedes}"
The car is a
$ echo $car

$


$ car="Ford"
$ echo "The car is a ${car:+Mercedes}"
The car is a Mercedes
$ echo $car
Ford
$
```

4.2.3 HOW VARIABLES, COMMAND ARGUMENTS, AND QUOTING METACHARACTERS ARE PROCESSED

The shell is a command interpreter that performs positional-parameter substitution, command substitution, and file-name generation for the arguments to commands. This subsection discusses the order in which these evaluations occur and the effects of the quoting mechanisms. You may find this helpful when trying to debug or write shell scripts.

The following substitutions occur before a command is executed:

1. Variable substitution. The actual values of user-named variables such as *$file* are substituted into the command line for the variable name. Positional parameters are also evaluated in this step.

2. Command substitution. The output of command lines enclosed in accent graves (`) is substituted into the command line. Only one evaluation occurs so that if, for example, the value of variable X is the string "$y", `echo $X` results in "$y", not the value contained in variable y.

3. Blank interpretation. Following the preceding substitutions, the resulting characters are broken into nonblank words (*blank interpretation*). For this purpose, blanks are the characters of the predefined variable "$IFS". By default, this string consists of blank, tab, and new- line characters. The null string is not regarded as a word unless it is quoted as in the following example, where the null string is passed as the first argument to echo:

```
echo ''
```

The next example calls *echo* with no arguments if variable *nada* has not been assigned a value or has been assigned the null string (''):

```
echo $nada
```

4. File-name generation. After blank interpretation has divided the command line into words, each word is then scanned for the file metacharacters *, ?, and [ ]. These metacharacters are used to match names of files in the directory in which the process is running (as in subsection 2.2.6, Using Metacharacters in File Names). Any files matching the specified patterns are assembled into an alphabetical list, with each file name being a separate argument to the command.

5. Variable assignment. Actual values are assigned to variable names for storage; for example, *name=value*. This is the converse of step 1.

The evaluations just described also occur in the list of words associated with a *for* loop. Only parameter and command substitution occur in the *word* used for a case branch; blank interpretation and file-name generation do not occur.

These five steps in command-line evaluation occur in the order listed; therefore, if one step produces output that is only evaluated in a previous step, that output is not evaluated. If, for example, the evaluation of a file name metacharacter results in the name of a command file, such as *ls*, that command is not executed and its output is not substituted, because command substitution occurs before file-name generation.

Another aspect of command evaluation to be aware of is that only one level of evaluation occurs; therefore, the following three command lines will produce the result shown in italics:

```
$ two='one'
$ three='$two'
$ echo $three
$two
```

The variable *$three* is evaluated to its value, *$two*, but the evaluation does not go any further. The result of one evaluation, *$two*, is not evaluated to see if it is a variable with a value. You can get the shell to interpret another level of variable or command substitution with the *eval* command. Try the following command lines, which demonstrate the *eval* command, at your terminal:

```
$ two='`ls -CF`'     # single quotes around accent graves around ls -CF
$ three='$two'
$ eval echo $three
```

The result of the three previous command lines is as follows:

```
`ls -CF`
```

Retype the third command line, *eval echo $three*, changing it as follows:

```
eval eval echo $three
```

The result is the execution of command line *ls -CF*; a listing of the files in your current directory.

In general, the *eval* command evaluates its arguments and treats the results as input to the shell. The shell then reads the input and executes any commands. Try the following example command lines:

```
$ wg="eval who | grep"
$ $wg fred
```

They are equivalent to the command line:

```
$ who | grep fred
```

In the preceding example, *eval* is required because there is no interpretation of metacharacters, such as |, following substitution; this is another example of the specific order of command-line parsing.

The following example sets up variables containing command substitutions, then executes a command calling those variables. The parsing process is shown at each step, with intermediate results:

```
$ user='/usr'          # Variable user gets the name of system directory /usr
$ dirs='$user /*bin'   # Variable dirs gets a string value, "$user /*bin"
$ ls `eval $dirs`      # Note accent graves indicating command substitution
```

Step 1. Positional parameter and variable evaluation.

$0 gets the string "ls" $1 gets the string "eval $dirs"

Step 2. Command substitution. The string value of $dirs is substituted.

Command line becomes: ls $user /*bin

Step 3. Blank interpretation. The substituted string value of *$dir* becomes two arguments.

Command line becomes: ls $user /*bin

Step 4. File-name generation. The shell searches for all file names that match string */*bin*, coming with two standard system files.

Command line becomes: ls $user /bin /lbin

Step 5. Variable assignment. The assigned value of variable *$user* is substituted for the variable.

Command line becomes: ls /usr /bin /lbin

In addition to the backslash and single quote metacharacters, there is a third quoting mechanism using double quotes. Within double quotes, parameter and command substitutions occur, though file name generation and the interpretation of blanks do not, just as with single quotes. Try the following two examples, comparing their output (in italics):

```
$ ship=Titanic
$ echo '$ship'
$ship
$ echo "$ship"
Titanic

$ echo 'This directory is: `pwd`'
This directory is: `pwd`
$ echo "This directory is: `pwd`"
This directory is: /u/john
```

To prevent variable and command substitution within double quotes, use the backslash metacharacter.

Example:

```
$ echo "You can\'t have \$99.00."
You can't have $99.00.
```

The following are characters that have a special meaning within double quotes and can be quoted using \:

Character	Meaning
$	Parameter substitution
*	Command substitution
"	Ends the quoted string
\	Quotes special characters $, *, ", and \

Figure 4-1 shows, for each quoting mechanism, which shell metacharacters are evaluated.

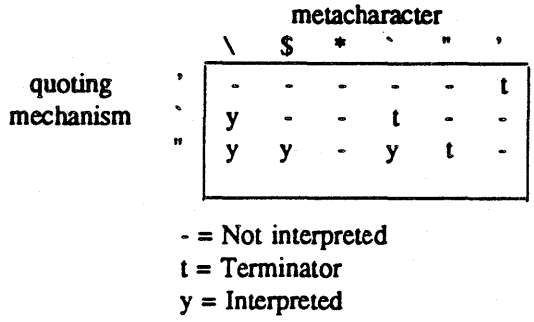

Figure 4-1. Quoting Mechanisms and Metacharacter Interpretation

4.2.4 A SAMPLE SHELL SCRIPT TO SEARCH FOR PATTERNS IN FILES

You can use the methods presented in subsection 4.2 to simplify and speed up repetitive searches. Writing a shell script to perform multiple searches will make them easier and faster for you to do them. Such a shell script can even make it possible for users who don't know UNICOS at all to perform these tasks, with simple menus such as the **menu** script in subsection 4.1.2.4, Looping With a Condition: The while and until Commands

The sample shell script presented here will search for all of the patterns listed in an input pattern file. The patterns must all be on one line, separated by spaces, although you can use the backslash to protect carriage returns, allowing you more than one line of patterns in the pattern file. The shell script also asks you for the name of the target file to search.

Example:

```
echo
echo "What is the name of the file you want to search?"
read searchfile
echo "What is the name of the file you want to create to contain the results of the search?"
read resultfile
echo "What is the name of the file containing the patterns to search for?"
read patternfile
patterns=`cat $patternfile`
for apattern in $patterns
do
     grep $apattern $searchfile >> $resultfile
done
```

Note the append redirection in the next-to-last line of the shell script. Append redirection must be used when inside a loop so that each successive iteration does not overwrite the output file with the output from only the latest iteration (as it would if > were used instead of >>).

You can enhance this script to ask for options to the *grep* command, and/or to search more than one file for the patterns. To search multiple files, create one file containing the names of the files to be searched, and use it in the same way this script uses a file containing multiple search patterns. You could also use this concept to perform multiple substitutions in a file (or files). Use two input files, one for the old strings to search for and one for the new replacement strings.

4.3 CHANGING THE SHELL ENVIRONMENT: PREDEFINED SHELL VARIABLES

The *shell environment* is the set of characteristics that determine how you interact with the shell and how it appears to you. Examples of such characteristics are what sort of shell prompt is displayed, where you can access the value of variables, and how your terminal is defined for the system. UNICOS lets you modify these characteristics and many others, with special predefined shell variables called *environment variables.*

4.3.1 ENVIRONMENT VARIABLES

Environment variables are predefined shell variables, usually with uppercase names, that affect your shell environment. They have the following properties:

- You can define, change, and access their values just as you do any of the variables that you define, as discussed in subsections 4.1.2.1, Named Variables, and 4.1.2.2, Availability of Variables: Scoping Rules and Commands.
- If you change their values and then log off the system, when you log on again the variables will have returned to their original values, because these values are system defaults.
- You can set up your login shell so that some environment variables automatically take on the values you want when you log in (subsection 4.3.2, The .profile File).

There are many of these variables that control the shell environment, but this subsection discusses only six of the most commonly used ones.

4.3.1.1 The HOME variable

The HOME variable contains the full path name of your home directory; the directory that you are always located at when you first log in. This is the directory that you automatically go to when you use the *cd* command with no arguments.

Type the following command lines at your terminal. The responses you see to the last two command lines will be the same:

```
$ cd
$ pwd
$ echo $HOME
```

You should not change the HOME variable without storing its value in another variable of your own and restoring the correct value of HOME when you are done. One occasion where you might want to do this is if you are doing work in a directory other than your home directory and the work requires a great deal of switching to other directories. In such a case, it would be convenient to be able to just type *cd* to return to your primary working directory. The following command lines let you do this:

```
$ realhome=$HOME
$ HOME=new_directory
```

new_directory is the full path name of the directory that will be your temporary "home base" for the work you are doing. When you are done working in this new directory, be sure to use the following command line, if you are going to do any more work on the system:

```
$ HOME=$realhome
```

The HOME variable is also useful in making your shell scripts more portable. If you write a shell script that references files in your home directory, you cannot use that script in any directory other than your home directory, because it will not be able to locate those files. Suppose that you have a script that must be able to use *cat* to display the contents of the file stuff in your home directory. To make this script portable, so you can execute it from other directories on the system, use the following line in the script:

```
cat $HOME/stuff
```

If **stuff** were in a subdirectory, **project**, of your home directory, you would use the following:

```
cat $HOME/project/stuff
```

4.3.1.2 The PATH variable

The **PATH** variable contains a series of path names that end in directories and are separated by colons. The shell uses these path names to search for the files containing the commands that you type. The shell searches these directories in the order in which they are specified in the **PATH** variable. To see what paths are in your **PATH** variable, type the following:

```
$ echo $PATH
```

Most system commands are in the directories **/bin, /usr/bin, /usr/ucb**, and **/usr/lbin** so these directories are usually specified in the **PATH** variable. Additionally, if you write many of your own shell scripts and use them more often than most system commands, you may want to create a subdirectory, **bin**, in your home directory, placing all of your shell script files in it. Then, if you want that directory searched first for commands (like shell script names), you can change your PATH variable as follows:

```
PATH=$HOME/bin:$PATH
```

****

CAUTION

If, as shown here, you add $HOME/bin *before* the system directories (in $PATH), you must be careful about naming your shell scripts. If you name a shell script with any system command name (such as *ls*, *rm*, *pwd*, and so on), the shell cannot access that system command, because it first checks your $HOME/bin directory, and finds a command there by that name. Further, if your script (for example, ls) calls a system command that has the same name (*ls*), the result is infinite recursion. See subsection 3.2.4, Files of Commands: Shell Scripts, for more information about this.

****

4.3.1.3 The MAILCHECK variable

The **MAILCHECK** variable lets you specify how often the system is to notify you of incoming mail from other users. The variable is specified in seconds, the default being 600 seconds, which is every 10 minutes. If you specify MAILCHECK=0, you will be immediately notified of any incoming mail. For information about the *mail* command, see subsection 3.3.1, The mail Command, or the *mail* entry in the UNICOS User Commands Reference Manual, publication SR-2011.

4.3.1.4 The PS1 and PS2 variables

The **PS1** variable contains the primary shell prompt string. By default, this is simply $ for the Bourne shell. You can set this to anything you want; to specify what machine you are on, what day it is (substitute the output of the *date* command), or other values:

```
PS1="CRAY2$ "
PS1=`date`
```

Setting the PS1 variable is a way to demonstrate that subshells invoked either explicitly with the *sh* command or automatically to run commands, are separate environments from your login shell (subsection 3.2.3.3, Commands for Background Processing: ps and kill). Write and execute the following shell script:

```
echo
echo $PS1
echo "No PS1 value in this shell yet"
PS1="prompt"
echo $PS1
echo
```

Your output should be as follows:

```
No PS1 value in this shell yet
prompt
```

The *first echo* $PS1 produces a blank line, because the subshell that is automatically created to run the shell script does not have a default value for PS1. This also shows that the value of PS1 in your login shell is not available in subshells. You could get around this using the *export* command, specifying the PS1 variable (subsection 4.1.2.2, Availability of Variables: Scoping Rules and Commands). The *second echo $PS1* produces the value, *prompt*, that the subshell's PS1 variable was set to within the subshell.

The PS2 variable contains the secondary system prompt string. The only time that this is displayed is if, at the $ shell prompt, you do not complete a quoted string or definition before entering a command line, or if you enter a multiline command such as *if*, *while*, or *case*. The default value for the PS2 variable is the character >, as you can see by typing the following command line:

```
$ echo "This is incomplete
```

The system responds with >, which is not very helpful. Type " to complete the command. You may want to use the following command line to assign PS2 a more explanatory string:

```
$ PS2="Close quotes! "
```

Do this, then repeat the preceding incomplete command line.

4.3.1.5 The TERM variable

The TERM variable contains a string value that tells the shell what kind of terminal you have. You would only want to reset this if you logged in from a different terminal. The strings to which to set TERM are frequently site-specific, so if you have occasion to reset this variable, you will need to ask your system administrator for your system's name for the terminal you want to define.

4.3.2 THE .profile FILE

The **.profile** file is a file of commands and environment variables that is automatically executed each time that you log on to the system. You can check to see if you already have a **.profile** file (you may not), with the **-a** option of the *ls* command. This option lists all files, including those that have names beginning with periods. Try it now:

```
ls -aCF
```

If you already have a **.profile** file, display its contents with the *cat* command:

```
cat .profile
```

You will probably see several of the environment variables being set to various values, along with other site-specific commands.

****

CAUTION

If you are uncertain of the meaning or purpose of some of the lines in **.profile**, DO NOT modify them. Because this file is automatically executed when you log in, incorrectly altering it can lock up your terminal or cause other problems until your system administrator redefines your **.profile** file.

****

Changing the values of environment variables in **.profile** is the way that you can automatically redefine them, so that every time you log in they take on the values you have given them, rather than their original values (the values they had when you first received your system account). You can also define your own system commands within **.profile**, as the next subsection discusses.

4.3.3 SHELL FUNCTIONS

Shell functions are commands that you can define either at the command line or in your shell environment file, **.profile**. Shell functions work just like other system commands, but are memory-resident, rather than fetched off of disk, so they usually execute more quickly. Unlike shell scripts, shell functions execute within the current shell; no subshell is created for them.

Shell functions that you define at the command line are temporary, being lost when you log out. Shell functions that you define in **.profile** are automatically set each time you log in, remaining the same until you change them in **.profile**.

When you define shell functions, avoid naming them with the same names as system commands. For the full caution about this, see subsection 3.2.4, Files of Commands: Shell Scripts. The generic format for a function definition is as follows:

```
name ()
{
  command_list
}
```

Define the following useful example function, *list*, by typing it at the command line (note the secondary prompts):

```
$ list ()
> {
> echo
> echo "Directory:  `pwd`"
> echo
> ls -alF | pg
> echo
> }
$
```

Now type *list*.

4.3.4 SHELL INVOCATION OPTIONS

When a shell is invoked it looks for options that specify what sort of environment it is to set up. Three of these options, **-x**, **-v**, and **-n**, were discussed in subsection 4.1.1.

There are three ways that Bourne shells are invoked. First, there is the login shell which is automatically invoked when you log in. Next, there are subshells invoked for the commands and shell scripts that you run. Finally, you can explicitly invoke Bourne shells with the *sh* command, as discussed briefly in subsection 3.8, Changing Shells.

When you explicitly invoke (create) subshells with *sh*, you can specify options to the command. You can then add the name of a command or shell script as a final argument, to execute that command/script in a subshell with particular characteristics that you specify with the options. The general format of such a command line is as follows, where *options* is one or more shell invocation option letters and *name* is the name of the command or shell script that you want to run in the specified shell environment.

sh *–options name*

To set options in your login shell, use the *set* command in your **.profile** file:

set *–options*

The following is a list of the more common options to the *sh* command:

Option	Description
-c *string*	The shell, immediately upon invocation, reads commands from a file named *string*.
-e	The shell terminates upon detecting any command execution errors (even minor ones).
-i	The shell is interactive; this is the normal default mode.

-n The shell ignores all commands upon having this option set.

****

CAUTION

Setting the **-n** option at the command line locks up your terminal, causing it to ignore all input, including commands to log out. Use this option only in shell scripts and subshells that terminate themselves.

****

-r The shell is restricted. This is a special security feature. Such shells prohibit the user from changing directories (cd), changing the value of the PATH variable, specifying path or command names containing /, and redirecting output (> >).

-s The shell reads commands from standard input (terminal); this is the normal default mode.

-v The shell is in verbose mode, echoing all commands, uninterpreted, to the screen as they are executed.

-x The shell echoes all commands, interpreted, to the screen as they are executed.

The special shell variable $- contains the names of all shell options that are set in the current shell (in whatever shell $- is accessed). To see the values for your login shell, type the following command line:

```
echo $-
```

If you are doing a great deal of debugging and do not want to keep typing the command line, *sh -options scriptname*, to debug scripts, you can set shell options for the duration of your login session by replacing your login shell with a different one that has different options set. You do this by combining two commands; the *sh* command which creates a shell with the specified options set, and the *exec* command, which replaces the current shell with the command that follows it (in this case a different shell). The command line is as follows:

```
exec sh -options
```

It is not necessary to replace your login shell; instead, use the *sh* command without the *exec* command, creating a subshell with the tracing mechanisms set, and leaving your login shell intact. Use the following command line:

```
sh -vx
```

When you want to get out of this "debugging mode" and execute shell scripts normally, just press CONTROL-d to exit this subshell and return to your login shell.

4.4 DEBUGGING SHELL SCRIPTS

Subsection 4.1.1, Basic Shell Script Debugging: Tracing Mechanisms, discussed the Bourne shell's basic debugging tool, the tracing mechanisms. This subsection covers two more complex debugging features of the Bourne shell, error handling and signal handling.

4.4.1 ERROR HANDLING AND COMMAND EXIT STATUSES

The shell handles errors in different ways depending on the type of error and on whether the shell is being used interactively. The UNICOS definition of an interactive shell is a shell that has its input and output connected to a terminal as determined by the system call *ioctl* (see the *ioctl* entry in the UNICOS System Calls Reference Manual, publication SR-2012). A shell invoked with the -i option is interactive.

Execution of a command can fail for any of the following reasons:

- Input or output redirection can fail if, for example, an input file does not exist or an output file cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally; for example, with a Bus error or Memory Fault signal.
- The command terminates normally but returns a nonzero exit status.

In all of these cases, the shell goes on to execute the next command, be it in a shell script or from the command line of a terminal. All other errors cause the shell to exit from a command procedure.

When any command executes, it returns an *exit status*, zero or nonzero, to the shell. You do not see this number, but you can test for it with the logical command combining metacharacters, && and || (discussed briefly in subsection 4.1.3.1, Evaluating Conditions: The test Command). The exit status of the last command executed is stored in the special shell variable **$?**.

The metacharacters && and || can be used to combine two or more commands. Successive commands execute conditionally, depending on the exit status of the immediately preceding command. The && metacharacter executes a command following it only if the exit status of the preceding command is zero (completely successful). The || metacharacter is the converse, executing a command that follows it only if the exit status of the preceding command is nonzero. Examples of each of these cases follow.

A zero exit status indicates that the command completed normally with no problems. Other exit statuses have other meanings. A command can have a nonzero exit status without producing an error message if it terminates normally, but without accomplishing its task. An example of this is the *grep* command:

grep *pattern file*

If *pattern* is not found in *file*, the result is neither an error nor a successful completion; therefore, the command terminates with no error message and a nonzero exit status. You can use this as an implicit *test* command. For example, suppose you are searching for a particular line in a file and you know that it contains one of two unique patterns, but not which one. The following command line will locate that line:

$ grep *pattern1 file* || grep *pattern2 file*

If the first search does not find a line containing *pattern1*, it will have a nonzero exit status (though no error message is returned), causing the || metacharacter to execute the second search.

Another use for these metacharacters is to test system conditions for simple yes/no answers, as the following command line does:

```
$ who | grep victoria > /dev/null  &&  echo "Victoria is logged on."
```

Because we are not interested in the actual output of the search, that output is redirected to the special system file **/dev/null**. This is a sort of system wastebasket for unwanted output.

Write the following shell script, **loggedon**:

```
$ who | grep $1 > /dev/null
$ echo $?
```

Call it repeatedly with different names of users on your system:

```
$ loggedon name
```

You cannot see the output of the search this way, but you will know whether or not the person is logged on, because a zero result indicates the search succeeded (the person is logged on) and nonzero indicates that the search failed (the person is not logged on).

4.4.2 UNICOS SIGNALS

A signal is the mechanism that UNICOS uses to notify a process (an executing command or shell script) that something has happened to influence the execution of that command/script. Signal types are indicated by numbers. UNICOS has 27 defined signals and another 32 available for users. These signals are defined in the *system header file*, **/usr/include/signal.h**, and are listed in appendix D, UNICOS Signals.

4.4.3 USING SIGNALS: THE trap COMMAND

The *trap* command lets you control what the system does when an error signal occurs. Place *trap* commands at the beginning of a shell script, before any other commands. The general format of the *trap* command is as follows:

```
trap  'command list; exit'  signal_number
```

The command list must be enclosed in single or double quotes, individual commands must be separated by semicolons, and the last command is usually the *exit* command, which terminates the subshell invoked to execute the script (subsection 4.1.2.2, Availability of Variables: Scoping Rules and Commands). Generally, scripts should terminate when they receive signals, because signals usually indicate some change affecting the script. If *exit* is omitted, the shell resumes executing the process from the point at which it received the signal. The *trap* command with no arguments displays all of the current signals that have traps set, showing those traps (the commands to be executed).

To display the message, "Program terminated by user," whenever a user enters an interrupt (usually CONTROL-c) to prematurely end a shell script, you would add the following line at the beginning of a shell script:

```
trap 'echo "Program terminated by user." ; exit' 2
```

The 2 is the interrupt signal as specified in the list in appendix D, UNICOS Signals.

5. THE C SHELL

This section discusses the UNICOS C shell, including the following features:

- Shell scripts
- Parameters and variables
- Changing the shell environment
- Debugging shell scripts
- Repeating previous commands

5.1 SHELL SCRIPTS

In section 3, you began to write shell scripts in the Bourne shell. These simple programs also run under the C shell. To go beyond such scripts to more complex ones requires using features specific to either the Bourne or the C shell. This section covers features of the C shell that let you create more advanced and useful shell scripts, using shell commands as you would a programming language. These features include variables, control flow (if-then-else, while, for, case), and input, among others.

Now that you will be writing more complex shell scripts, it is a good idea to add comment lines to them, documenting what the scripts do. The comment character is the pound sign (#). Place it anywhere on a line to begin a comment. Once a # character is encountered, the rest of the line is ignored by the shell as nonexecutable. Examples of comments are as follows:

```
# This is a comment-only line.
ls -CF | pg  # This line contains both code and comment.
```

The # character has another use in the C shell. As the first line of every shell script, you must have the following line, beginning at the first character position of the line and typed exactly as shown:

```
#!/bin/csh
```

This line tells the C shell that the shell script is a C shell script, rather than a Bourne shell script. This is necessary because the UNICOS C shell assumes that shell scripts are Bourne shell scripts, unless explicitly told otherwise.

Because you must put this line at the beginning of every C shell script file, write one of the following two C shell scripts (depending on which editor you prefer) to automatically begin editing sessions for script files with this first line.

Shell script, **edscript**, for the *ed* editor:

```
#!/bin/csh
ed $1
a
#!/bin/csh
```

Whenever you are going to write a shell script, use **edscript** as follows (the % is your system prompt):

```
% edscript scriptname
```

Shell script, **viscript**, for the *vi* editor:

```
#!/bin/csh
vi $1
a
#!/bin/csh
```

Whenever you are going to write a shell script, use **viscript** like this:

```
% viscript scriptname
```

Before you try to use either of these scripts, remember to make them executable as follows:

```
chmod 755 edscript
```

5.1.1 BASIC SHELL SCRIPT DEBUGGING: TRACING MECHANISMS

Before you begin to write more complex shell scripts, it is a good idea to know the tools available to help in debugging those scripts. This subsection explains two of the tracing mechanisms that the C shell provides for debugging. As you proceed to the next subsections, refer back to this discussion as you need the debugging tools. This subsection covers only the simplest of several debugging tools. Subsection 5.4, Debugging Shell Scripts, discusses other debugging tools, useful for debugging more complex shell scripts.

The shell has two tracing mechanisms, the **-v** and **-x** options, that you will find useful in debugging shell scripts.

To set these options in a C shell, you must invoke the shell with the following options, where *options* is one ro both of the option letters **-v** or **-x**:

```
csh -options
```

This command line creates a subshell with the specified options set, in essence creating a particular environment (one useful for debugging) in which you can run scripts. For more information on subshells, see subsection 3.2.3.3, Commands for Background Processing: ps and kill. A complete list of shell options is presented in subsection 5.3.4, Shell Invocation Options.

The **-v** option causes the script to display commands after the shell has done history substitution (see subsection 5.5, Repeating Previous Commands: The History Mechanism), allowing you to see exactly what commands and options are being executed.

The **-x** option causes the shell to display commands after command substitution, file name generation, and variable substitution have taken place (subsections 5.1.1.1, 5.2.1, and 5.2.3). You can see exactly what input the shell is getting for each command and its options.

Write the following shell script, **tracer**, to see the **-v** and **-x** options in operation:

```
#!/bin/csh
ls -l *mp
```

Now execute the file with the following command line (the % is your system prompt):

```
% csh -vx tracer
```

Your output should look as follows:

```
/bin/ls -l *mp
/bin/ls -l temp
-rw-r--r--  1 name    group        333 Mar 21 10:53 temp
```

Your login name will be in place of *name* and your group will be in place of *group*. The date and time will be different in your output, as will the permissions, if your system sets them differently.

Each command line (there is only one here) is repeated twice. The -v option echoes it, just as it appears in the script file, checking the line for syntax errors. The -x option again echoes the line after interpretations and substitutions, which, in the preceding example, is the substitution of all file names that match the pattern, *mp.

As you learn about variables and learn more about metacharacters in this section, try the -v and -x options with more complex shell scripts, to see more exactly what their output is in different situations.

****

CAUTION

There is another option you can set, the -n option, that causes the shell to read commands and display them. It does not ,however, execute them, but checks them for syntax errors.

Setting the -n option at the command line (the $ or % prompt) locks up your terminal, causing it to ignore all input, including commands to log out. Use this option only in shell scripts and subshells that terminate themselves.

****

5.1.2 VARIABLES IN SHELL SCRIPTS

As with any programming language, the C shell lets you use variables to contain information. These variables come in two basic types: pre-defined shell variables and user-defined variables. This subsection covers userdefined variables, which come in two varieties; named variables and command-line positional variables.

5.1.2.1 Named variables

Variables may be any legal UNICOS name which can consist of one to 14 alphanumeric characters. UNICOS does distinguish between uppercase and lowercase, so the variables *name* and *Name* are different.

To set a variable to a value, use either of the following syntaxes:

set *variable* = *value*
set *variable=value*

As a general rule, there must be an equal number of spaces on each side of the equal sign (=). The *value* can be a word list, which is one or more words (contiguous nonblank characters) enclosed in parentheses. To set a variable to the default null value, use either of the following command lines:

set *variable*
set *variable* = ""

NOTE

Named variables in the C shell do not have a default null value; they must be set to some value before you try to access them with the $. Trying to access the value of an undefined variable is an error in the C shell. Any shell script attempting this is immediately terminated.

The *set* command with no arguments displays a list of all set variables and the values to which they are set. Type the following command line, and you should see a list of variables that you have set:

set

These variables are system variables that are set by the shell to default values when you log in. Some of them will be discussed in later subsections.

The C shell has several different ways to access the value of a variable. The simplest is to use a dollar sign ($) in front of the variable's name: *$variable*. With another syntax, *$variable[integer]*, you can access the individual words of a variable's word list, as if the words were in an array and *integer* were the index to that array. Try the following example (the % is your system prompt, and output is indicated in italics):

% set list = (one two three four "five six")
% echo $list
one two three four five six
% echo $list[3]
three
% echo $list[5]
five six
% echo $list[6]
Subscript out of range

You can set a variable equal to itself plus an increment for counting operations, but you cannot use arithmetic with the *set* command. You must use the @ operator (explained in subsection 5.1.3.1, Evaluating Conditions:

Shell Expressions). Try the following example at the shell prompt (italics indicate output):

```
% set counter = 4
% echo $counter
4
% @ counter = $counter + 3
% echo $counter
7
```

Another syntax for incrementing variables by 1 (and only 1) is as follows (try it):

```
% set count = 0
% echo $count
0
% @ count ++
% echo $count
1
```

NOTE

Variables are correctly interpreted inside of double quotes, the value being correctly substituted for the variable name. Inside of single quotes, however, the string will be interpreted literally as a $ character with other characters after it.

Make a simple shell script now to try using variables. Create the following file, **carfile**:

```
#!/bin/csh
set car = (Alpha Romeo)
set driver = Emily
set tires = Pirelli
echo $car
echo $car[1]
echo $car[2]
echo $driver
echo $tires
echo "$driver drives an $car with $tires tires"
```

Make the file executable, then execute it at the command line as follows:

```
% carfile
```

You will get the following response:

```
Alpha Romeo
Alpha
Romeo
Emily
Pirelli
Emily drives an Alpha Romeo with Pirelli tires.
```

You can also have variables accept values typed in from the command line as interactive input, using the special shell variable $<. Write the following shell script, car2, and then execute it:

```
#!/bin/csh
echo "Please enter the name of the driver:"
set driver = $<
echo "Please enter the make of car:"
set car = $<
echo "What sort of tires:"
set tires = $<
echo "$driver drives a $car with $tires tires."
```

You can unset one or more variables with the *unset* command as follows:

unset *variable1 variable2 ...*

5.1.2.2 Availability of variables: Scoping rules and commands

The *setenv* command lets you use the value of a variable in a shell other than the one in which it is defined. Its syntax for setting a variable is as follows:

setenv *variable value*

To display a list of all *setenv* variables and their values, use *setenv* with no arguments. Recall from the discussion of the *ps* command (subsection 3.2.3.3) that subshells are created for all commands and shell scripts that are executed. Therefore, such commands are not executing in your login shell and cannot use the values of variables declared in your login shell. To see this, write the following shell script, scope:

```
#!/bin/csh
set var1 = one
set var3 = three
echo "var1 equals $var1"
echo "var2 equals $var2"
echo "var3 equals $var3"
```

Next, type the following commands at the command line:

```
% set var1 = 111
% setenv var2 222
% setenv var3 333
```

Now verify the results of the preceding three command lines with the following commands (output in italics):

```
% set
```

```
...
var1    111

% setenv
...
var2=222
var3=333
```

Now execute **scope**:

```
% scope
```

You should get the following results:

```
var1 equals one
var2 equals 222
var3 equals three
```

Because var1 was not set in the login shell with *setenv*, its value of 111 is not available to the script **scope**, executing in a subshell. var2 was set with *setenv* in the login shell so **scope** echoes its login shell value. The variable var3 was also set with *setenv* in the login shell, but it was reassigned a new value in the subshell in which **scope** is running, so **scope** echoes the new subshell value of var3. This change in the value of var3 is local to the subshell and does not change the value of var3 in the login shell, as you can verify by typing the following command lines after running **scope** (output indicated with italics):

```
% set
...
var    111

% setenv
...
var2=222
var3=333
```

This shows that setting variables with *setenv* is a one-way process; variables defined in shells can be accessed in subshells and locally redefined subshells. Those local redefinitions, however, are not transferred upward to the shell in which the variable was originally defined. Variables, even when set with *setenv*, can only be altered in the shell in which they were originally defined.

You can prevent subshells being created to execute commands with the *source* command. This forces the command or shell script to execute in the current shell. Execute **scope** with the *source* command:

```
% source scope
```

The results will appear to be the same as they were before, until you check the values of the variables in the login-shell:

```
% set
...
var1    one
var3    three

% setenv
...
```

var2=222
var3=333

Because scope executed in the current shell where all the variables were originally defined, it changed the values for the two variables assigned within the script (var1 and var3). The value of the *set* variable, var1, was changed. The value of the *setenv* variable, var3, remains the same, but a new *set* variable, var3, is created, taking on the value assigned to it within the shell script.

The *unsetenv* command is analogous to the *unset* command. It removes the values of *setenv* variables, removing them from the list of defined variables:

unsetenv *variable*

5.1.2.3 Command-line positional variables

Command-line positional variables are variables that automatically exist in the shell. These variables are the numerals 0 through 9, and the shell sets them to the values of the words on a command line, according to the order of the words. Look at the following command line and then at the breakdown of it in the next paragraph:

ls -CF bookdirectory

The first word, command *ls*, goes into variable 0. The second word, *-CF*, goes into variable 1. The third word, *bookdirectory*, goes into variable 2. Because there is this association by order or position on the command line, these variables are known as positional parameters. Positional parameter 0 is unlike the others (1 and greater), in that it always is set (automatically by the shell) to the value of the command on a command line. You cannot explicitly set it to anything else.

To access the values in positional parameters, place a $ before the numeral, just as you would access the value of a named variable.

To set positional parameters 1 through n, substitute the appropriate numeral into the command line in a shell script for the argument that you want it to take on as a value. For example, in the preceding command line, to put the value of the directory to be listed into a positional parameter, you would use the following line in a shell script:

ls -CF $1

Write the following shell script, **parms**, to demonstrate these concepts: (Please copy it exactly, you will be using it again later.)

```
#!/bin/csh
echo "The first parameter is: $1"
echo "The second parameter is: $2"
echo "The third parameter is: $3"
```

Change the file's permission so you can execute it (subsection 2.5.3, Permissions), then try it with the following input:

Input	**Response**
% parms one	The first parameter is: one The second parameter is: The third parameter is:

```
                          %
% parms one two           The first parameter is:  one
                          The second parameter is:  two
                          The third parameter is:
                          %
% parms one two three     The first parameter is:  one
                          The second parameter is:  two
                          The third parameter is:  three
                          %
% parms "one two" three   The first parameter is:  one two
                          The second parameter is:  three
                          The third parameter is:
                          %
```

As another example, create the following shell script, **parms2:**

```
#!/bin/csh
echo "The command is:  $1"
echo "The first argument is:  $2"
echo "The second argument is:  $3"
$1 $2 $3
```

Make the file executable, then use it as follows, examining the output carefully:

```
% parms2  ls  -CF  .
```

Try it again with the following command line:

```
% parms2  grep  d  carfile
```

The two preceding examples demonstrate that you can use the contents of positional parameters (or of named variables) as either data or executable commands. You can assign to variables the full path names of executable files. You are encouraged to explore this flexibility further with your own exercises and experimental shell scripts.

Create the following shell script for your own use, naming it **x:**

```
#!/bin/csh
chmod  755  $1
```

This shell script, **x**, changes the permissions of the file argument you give it, to make that file executable. Use it as follows:

```
% x filename
```

5.1.2.4 Moving positional parameters: The shift command

The shell keeps track of all arguments on the command line, held in the positional parameters, 1 through n. You may want to have a shell script iterate over any number of arguments with which you call it, operating on each one of them. The *shift* command makes this possible by moving the values of all the positional parameters down one number and discarding the value of $1. Therefore, you can operate on any number parameters by using them sequentially, discarding each as you use it. Write the following shell script, **tenparms1**, to see how this works:

```
#!/bin/csh
echo $1 ; shift
echo $1 ; shift
echo $1 ; shift
echo $1 ; shift
echo $1 ; shift
echo $1 ; shift
echo $1 ; shift
echo $1 ; shift
echo $1
```

Now execute the script, calling it with the following arguments:

```
% tenparms one two three four five six seven eight nine ten
```

You can specify a numeric argument with the *shift* command to shift the arguments by more than one positional parameter. Try the following shell script, **moreparms**:

```
#!/bin/csh
echo $1 ; shift 3
echo $1 ; shift 3
echo $1 ; shift 3
echo $1
```

Invoke it with the following command line:

```
% moreparms one two three four five six seven eight nine ten
```

You should get the following response:

```
one
four
seven
ten
```

5.1.2.5 Special command-line variables

This subsection discusses six special variables, automatically set by the shell, that are related to those in the previous subsection.

The first special parameter, **$0**, is a variable that always contains the name of the command currently executing.

Another special variable automatically set by the shell, is **$#argv**. This variable contains the number of positional parameters typed on a command line. It is used in shell scripts to count arguments to a script.

Add the following line at the end of your shell script **parms**:

```
echo "The number of parameters is:  $#argv"
```

Now run the script with the same data as you did in subsection 5.1.1.2, looking carefully at the output.

The special shell variable **$*** (also **$argv**) takes on the values of all the positional parameters, except the special one, **$0**. Add the following line to the end of your shell script *parms*:

```
echo "All arguments are:  $*"
```

Run **parms** just as you did before, examining the results. The **$*** variable lets you apply a command to more than one argument. Alter your shell script **x**, substituting **$*** for **$1**, so the file looks like this:

```
#!/bin/csh
chmod  755  $*
```

Shell script **x** will now change the permissions to 755 for any number of files you give it, making all of them executable. Use it as follows:

```
% x filename1 filename2 ... filenamen
```

The next special shell variable is **$#name**. This variable contains the number of words (contiguous nonblank characters) in the variable *name*. If *name* contains a word list of 5 words, **$#name** has the value 5.

Another special shell variable is **$?name**. This variable is useful in setting up conditions for conditional branching in shell scripts (discussed in subsection 5.1.3.1). It returns a 1 if the variable *name* is defined or it returns a 0 if the variable *name* is undefined or null.

The last special shell variable that this subsection will discuss is **$$**. This variable contains the process id number (PID) of the current shell. Because every command and shell script gets its own subshell when it executes (subsection 5.1.2.2, Availability of Variables: Scoping Rules and Commands) this number is unique to each invocation of a command or shell script. Therefore, this number is often used to name temporary output files. That way, successive uses of a command/script will not overwrite previous output files. Use it as follows:

```
who | grep mygroup > groupfile.$$
```

This gives you a different file name, groupfile.*number* than any other user gets from using the same command line, the difference being *number*. You also get a unique number for **$$** each time that you run a shell script with a **$$** in it.

5.1.3 CONTROL FLOW

The C shell has five of the control flow constructs basic to programming languages: conditional branching, for, while, if-then-else, and case. The following five subsections discuss the C shell's treatment of these constructs.

5.1.3.1 Evaluating conditions: Shell expressions

Before you can do any conditional programming, you must be able to set up conditions. The C shell has a number of facilities for doing this, most of them arithmetic. The C shell has a built-in capability to perform arithmetic operations, which allows you to test any condition for which you can devise a mathematical test. The following operators are available:

Standard arithmetic operators:

```
+  -  *  /  >  <  ≥  ≤
```

Additional arithmetic operators:

%	Modulus
==	Equal (can also compare strings)
!=	Not equal (can also compare strings)
( )	Arithmetic grouping

Logical operators:

||	Logical OR
&&	Logical AND
!	Logical NOT

Bit operators:

~	one's complement
>>	right shift
<<	left shift
>>	right shift
&	bitwise AND
^	bitwise exclusive OR
|	bitwise inclusive OR

The C shell's arithmetic function, @, can be executed any of the three following ways:

@ Prints the values of all variables (like the *set* command does)

@ name = *expr*
Assigns the value of the arithmetic expression, *expr* to the variable *name*

name[index] = *expr*
Assigns the value of the arithmetic expression, *expr*, to the *index*-th word of the predefined word list in variable, *name*. (You cannot index a variable that is not a word list.)

Try the following example (system's response in italics):

```
% @ eight = 6 + 2
% echo $eight
8
```

The conditions for files return a 1 if they are true; otherwise, they return a 0. These conditions are as follows:

-d *file*	True if *file* exists and is a directory file
-e *file*	True if *file* exists
-f *file*	True if *file* exists and is a regular file
-o *file*	True if the user making the inquiry is the owner of *file*
-r *file*	True if *file* exists and can be read by the user
-w *file*	True if *file* exists and can be written to by the user
-x *file*	True if *file* exists and can be executed by the user
-z *file*	True is *file* has zero length (is empty)

Try the following conditions at the command line, being certain to unset the variable, *result*, after each use so successive tests do not accidentally use the results of a previous test. Conditions evaluating to true result in a 1, and conditions evaluating to false result in a 0. Italics indicate the shell's responses:

Example 1:

```
% @ result = 99 == 9 * 11
% echo $result
1
% unset result
```

Example 2:

```
% @ result = -f carfile
% echo $result
1
% unset result
```

Example 3:

```
% @ result = -d temp
% echo $result
0
% unset result
```

Example 4:

```
set animal = rabbit
% @ result = $animal == bunny
% echo $result
0
% unset result
```

Example 5:

```
% set name = kelly ; set animal = bat ; set person = kelly
% @ result = $animal == bat  ||  $person == "kelly"
% @ echo $result
1
```

5.1.3.2 Branching on one condition: The if command

The *if* command lets you test one condition and perform a command if the condition evaluates to true (1). All of the conditions in the preceding subsection may be used in the *if* construct.

The simplest format of the *if* construct is as follows, where *expr* is a condition and *command* is a simple command (not a command list, pipeline, or sequence of commands separated by semicolons).

```
if (expr) command
```

If *command* involves redirection of I/O, that redirection occurs even if the expression evaluates to false.

Another format of the *if* construct is as follows:

```
if (expr1) then
        command_list1
else if (expr2) then
        command list2
else
        command_list3
endif
```

The *if*, *expr*, and *then* must all be on one line with nothing else. If *expr* is longer than one line, use a protected new line (backslash before the return) to start the second line. Any number of *else if* statements are allowed and any number of commands may be in the command_list, either on separate lines or separated by semicolons. Pipes of commands and I/O redirection are also allowed in this command list. The *endif* ends the *if* construct, it must be the last line of the *if* construct, and it must be on a line by itself. As an example of this format of the *if* command, write and execute the following shell script:

```
#!/bin/csh
set var0 = something
if (-w \ /) then
    echo "I can write to root directory" > cond1
else if ($var0 == something) then
    echo "var0 equals $var0"
else if (-w doc && $var0 == something) then
    echo "I can write to file doc and var0 equals something"
else
    echo "The date is: `date`"
endif
```

Execute this script. The output should be as follows:

```
I can write to file doc and var0 equals something
```

Try changing the conditions so that file **cond1** is created with a message in it, and/or the command substitution of the last condition is performed. For further practice, try other types of conditions and numeric tests to gain familiarity with them.

5.1.3.3 Branching on many conditions: The switch command

The *switch* command provides a way to quickly and simply test a string for many possible matches. The format of the *switch* command is as follows:

```
switch (string)
case pattern1:
     command_list1
     breaksw
case pattern2:
     command_list2
     breaksw
default:
     command_listn
     breaksw
endsw
```

This is a pattern-matching command, checking the *string* against the patterns in each of the case lines. You can use the file metacharacters *, ?, and [ ] in any of the patterns. If no matches are found in any of the case patterns, the commands after the keyword *default* are executed. If no *default* exists, execution continues after the *endsw*. Any number of commands may be in the command_list, either on separate lines or separated by semicolons. Pipes of commands and I/O redirection are also allowed in this command list.

Create the following shell script, **weekday**, to print out weekday names, given a numeric argument:

```
#!/bin/csh
switch ($1)
case 1:
     echo "Sunday"
     breaksw
case 2 :
     echo "Monday"
     breaksw
case 3 :
     echo "Tuesday"
     breaksw
case 4 :
     echo "Wednesday"
     breaksw
case 5 :
     echo "Thursday"
     breaksw
case 6 :
     echo "Friday"
     breaksw
case 7 :
     echo "Saturday"
     breaksw
default:
     echo "You must enter a number between 1 and 7"
     breaksw
endsw
```

Make the file executable, then test it with command lines like the following:

```
% weekday 2
% weekday 8
```

The following shell script, **language**, uses metacharacters to determine if the file argument it is given is a Fortran, C, or Pascal source file. The naming conventions given in subsection 2.2.1, File-Naming Conventions, must be followed.

```
#!/bin/csh
switch ($1)
case *.f :
    echo "Fortran source file";;
    breaksw
case *.c :
    echo "C source file";;
    breaksw
case *.p :
    echo "Pascal source file";;
    breaksw
default :
    echo "unknown";;
    breaksw
endsw
```

You can use the *switch* command to set up options for your shell scripts. For example, you can write a shell script that prints out the current list of system users in various formats. Write the following script, **menu**:

```
#!/bin/csh
# -c displays calendar for current year, -d displays the date and time,
# -l provides a listing of the current directory, -q quits the system, logging you out,
# and -w provides a list of who is currently on the system.
# Anything other than the above options terminates the script
switch ($1)
case -c :
     echo "What is the current year?"
     set year = $<
     cal $year | pg
     breaksw
case -d :
     date
     breaksw
case -l :
     echo "Directory is:"
     pwd
     ls -CF | pg
     breaksw
case -q :
     echo
     breaksw
case -w :
     who | pg
     breaksw
default:
     echo "The permitted option letters are:  c, d, l, q, and w"
     breaksw
endsw
```

Use the script with each of its correct options and some incorrect ones; menu *-option*

For more information on processing command-line options within shell scripts, see the *getopt* entry in the UNICOS User Commands Reference Manual, publication SR-2011.

5.1.3.4 Looping with a condition: The while command

The *while* construct lets you repeat a series of commands in a shell script until a condition becomes false. The following is the general format of the *while* command:

```
while (expr)
     command_list
end
```

The *while* and *expr* must be on one line with nothing else, and the *end* must by on a line by itself, ending the while loop. For *expr*, you can use any of the conditions mentioned in subsection 5.1.3.1, Evaluating conditions: Shell expressions. Any number of commands may be in the *command_list*, either on separate lines or separated by semicolons. Pipes of commands and I/O redirection are also allowed in this command list.

Write the following shell script, which uses the *while* command to execute the functions of the **menu** shell script until the user exits by typing -q:

```
#!/bin/csh
set choice   # give choice the default null value
while ($choice != -q)
     echo "–c displays calendar for current year,  –d displays the date and time,"
     echo "–l provides a listing of the current directory,  –q quits this menu of options,"
     echo "and –w provides a list of who is currently on the system."
     echo "Anything other than these options repeats this menu."
     echo "What is your choice?"
     set choice = $<
     switch ($choice)
     case -c :
          echo "What is the current year?"
          set year = $<
          cal $year | pg
          breaksw
     case -d :
          date
          breaksw
     case -l :
          echo "Directory is:"
          pwd
          ls -CF | pg
          breaksw
     case -q :      # Dummy action for the -q option, until it exits at top of loop
          echo
          breaksw
          case -w :
          who | pg
          breaksw
     default:
          echo "The permitted option letters are:  c, d, l, q, and w"
          breaksw
     endsw
end     #end the while loop
```

Practice with the script, entering correct and incorrect options as it asks you for them. Alter the menu, if you want to try other actions. You can create a menu just like this to simplify, to one keystroke, actions that you commonly perform, such as submitting jobs for compiling, loading, and executing, or searching files for strings.

You can have the compiler, loader, or search options present in the shell script, or you can have it ask for them, just as it must ask you for the name of the program file to compile, load, or search.

The *while* command is often used with the *shift* command to operate on an unknown number of arguments until all have been processed. The following shell script is a more elegant version of the first script in subsection 5.1.2.4, Moving Positional Parameters: The shift Command:

```
#!/bin/csh
while ($#argv != 0)
     echo $1
     shift
end
```

Try it if you like, calling it with any number of arguments.

5.1.3.5 Looping with a specified index: The foreach and repeat commands

With the *foreach* command, you can repeat a sequence of commands in a shell script a prespecified number of times. One format of the *foreach* command is as follows:

```
foreach variable (wordlist)
        command_list
end
```

The *foreach*, *variable*, and *wordlist* must be on the same line, with nothing else. The parentheses around the wordlist are required. Any number of commands may be in the *command_list*, either on separate lines or separated by semicolons. Pipes of commands and I/O redirection are also allowed in this command list. The *end* must be on a line by itself, and it ends the *foreach* loop.

This construct will perform the commands in the *command_list* as many times as there are words in the *wordlist*, the variable taking on the value of each successive word on each iteration through the loop. To see this, create the following shell script, **forscript**:

```
#!/bin/csh
foreach index (word1 word2 word3)
    echo $index
end
```

Execute the shell script as follows (output in italics):

```
% forscript
word1
word2
word3
```

This simple example script merely repeats the words in the wordlist. The real use for the *foreach* command is when you need to perform a command or series of commands on many arguments, data, or inputs.

The words in the wordlist can be executable commands, which is useful if there is a series of commands that you repeatedly perform.

The following example might be something a person would do at the start of each session:

```
#!/bin/csh
foreach index ( "cal 1987"  who  "ls -CF" )
    $index | pg
end
```

This script will perform each of the commands in the order specified in the wordlist.

You can combine the *foreach* and *switch* commands to make shell scripts that accept more than one option. Recall the example shell script, **menu**, from subsection 5.1.3.3 that was called with one option letter, each option letter having one associated action. By adding a *foreach* command to that shell script, you can make the script accept any number of option letters at one time, performing the appropriate action. Modify that script to look like the following one:

```
#!/bin/csh
# -c displays calendar for current year,  -d displays the date and time,
# -l provides a listing of the current directory,  -q quits script,
# and  -w provides a list of who is currently on the system
echo "What options do you want?  (specify each option with a dash,
echo "putting spaces between the options:  -c -d -q)
set options = $<
foreach opt ($options)
    switch ($opt)
    case -c :
        echo "What is the current year?"
        set year = $<
        cal $year | pg
        breaksw
    case -d :
        date
        breaksw
    case -l :
        echo "Directory is:"
        pwd
        ls -CF | pg
        breaksw
    case -q :
        exit
        breaksw
    case -w :
        who | pg
        breaksw
    default:
        echo "The permitted option letters are:  c, d, l, q, and w"
        breaksw
    endsw
end
```

You can now call the script with any number of options (separated by spaces), and it will perform the associated actions in the order you specify the options. Call it with a number of options as follows:

```
menu  -l -d -k -w -q
```

Another command that iterates a specified number of times is the *repeat* command. This will repeat a simple UNICOS command the number of times specified with a counter. The syntax is as follows, where *count* is an integer and *command* is a simple command (not a command list, pipeline, or sequence of commands separated by semicolons).

repeat *count command*

If *command* involves redirection of I/O, that redirection occurs exactly once, even if count=0.

5.1.4 SHELL PROGRAMS CONTAINING THEIR OWN INPUT: here documents

You can create shell scripts that provide some or all of the input that they require; these are called *here documents*. This is useful for operations that you perform repeatedly, such as writing memos that have a standard heading. The following is the general format for a here document:

```
command << string
    command_input
string
```

The *command* is a UNICOS command, *command_input* is the input that the command requires, and *string* is a delimiter, not found in the input, indicating where the input begins and ends.

Example:

```
ed textfile << EOF
a
Line one of the input text for the new file, textfile.
Line two of text to make up the new text file.
Last line of input for the new file, textfile.
w
q
EOF
```

This inclusion of input within a command can be done either in a shell script or at the command line. An example of doing it from the command line is follows:

```
$mail Jeanne  << STOP

Hi, Jeanne.

There is a new shell script, phoney, on the system that creates and maintains a phone
directory. I've already found it very handy.  it is in the directory /usr/lbin.

Howard
STOP
$
```

5.1.5 A SAMPLE SHELL SCRIPT TO COMPILE, LOAD, AND EXECUTE PROGRAM FILES

You can use the methods presented in subsection 5.1 to greatly simplify the repetitive tasks of compiling, loading, and executing program files. Writing a shell script to perform these tasks will make it easier and faster for you to do them. Such a shell script can also make it possible for users who do not know UNICOS to perform these tasks, with simple menus such as the **menu** script in subsection 5.1.2.4.

You can write the following shell script to compile, load, and execute programs, having it specify a few simple options for the procedures and ask you for names of input and output files. This example shell script is for Fortran programs, but it can easily be adapted to call other language processors, or give you options to select among language processors. Read through the shell script carefully until you understand how it works, then read the suggestions that follow it to see how you can further tailor the script to your particular needs.

```
#!/bin/csh
echo "What is the name of your source code file?"
set sourcefile $<
cft77  -a stack  $sourcefile
echo "What do you want the name of your executable file to be?"
set execfile $<
segldr  -o $execfile  $sourcefile.o
echo "Does your program require an input data file (Y/N)?"
set answer1 $<
if ($answer1 == Y || $answer1 == y) then
     echo "The name of the input data file?"
     set indata = $<
endif
echo "Does your program require an output data file (Y/N)?"
set answer2 = $<
if ($answer2 == Y || $answer2 == y) then
     echo "The name of the output data file?"
     set outdata = $<
endif
if ($answer1 == Y || $answer1 == y) then
     if ($answer2 == Y || $answer2 == y) then
          $execfile < $indata > $outdata
     else
          $execfile < $indata
     endif
else if ($answer2 == Y || $answer2 == y) then
     $execfile > $outdata
else
     $execfile
endif
```

You can enhance this shell script with any combination of the following features:

- Include more options to the compiling and loading commands
- Have the script ask users for options to the compiling and loading commands
- Have the script ask users for the language of the source file and invoke the appropriate compiler/assembler

There are many other capabilities you can build into this script to tailor it to you specific needs for flexibility, usability, and convenience.

5.2 SHELL PARAMETERS AND VARIABLES

This subsection discusses more complex details of the ways that the C shell uses and interprets variables and parameters. Previous discussions have focused more on the use of variables; this subsection will explain some of the concepts behind such usage.

5.2.1 SUBSTITUTING A COMMAND'S OUTPUT FOR OTHER SHELL VALUES

The accent grave metacharacter (`) lets you use the output of a command in a number of ways. You can store that output in a variable or use it as as input to another command or shell script. To store the output in a variable, use the following syntax, where *command* can be a simple command, a pipe, or any other legal UNICOS command line:

set *variable* = `*command*`

Storing the output of a command in a variable is handy when you will want to use that output repeatedly. Having the output in a variable requires less typing on your part and reduces the execution time of shell scripts over having the information in a file or repeatedly executing the command.

If, in a shell script, you want to use the date and time in several places, you can use the following line to place the output of the date command into the variable *d* and then use *$d* for the information:

set d = `date`

To use the output of a command as input to another command or shell script, use the following syntax:

command1 `*command2*`

The command (*command1*) receiving the output can be a simple command, a pipe, or any other legal UNICOS command line.

As an example, suppose you have a file, **maillist**, containing the login names (all on one line) of several people to whom you regularly send mail on the system. If you discover some information you want to send them, you can use the following command line:

```
mail `cat maillist`
your message
...
CONTROL-d
```

5.2.2 HOW VARIABLES, COMMAND ARGUMENTS, AND QUOTING METACHARACTERS ARE PROCESSED

The C shell is a command interpreter that performs positional parameter substitution, command substitution, and file-name generation for the arguments to commands. This subsection discusses the order in which these evaluations occur and the effects of the quoting mechanisms. The following substitutions occur before a command is executed:

1. Variable substitution. The actual values of user-named variables such as *$file* are substituted into the command line for the variable name. Positional parameters are also evaluated in this step.

2. Command substitution. The output of command lines enclosed in accent graves (`) is substituted into the command line. Only one evaluation occurs so that if, for example, the value of variable *X* is string "$y", then `echo $X` results in "$y", not the value contained in variable y.

3. Blank interpretation. Following the preceding substitutions, the resulting characters are broken into nonblank words (*blank interpretation*). `blanks` are the characters of the predefined shell variable fB$IFS. By default, this string consists of blank, tab, and new line characters. The null

string is not regarded as a word unless it is quoted as in the following example where the null string is passed as the first argument to *echo*:

```
echo ''
```

The next example calls *echo* with no arguments if variable *nada* has not been assigned a value or has been assigned the null string (''):

```
echo $nada
```

4. File-name generation. After blank interpretation has divided the command line into words, each word is then scanned for the file metacharacters *, ?, and [ ]. These metacharacters are used to match names of files in the directory in which the process is running (as in subsection 2.2.6, Using Metacharacters in File Names). Any files matching the specified patterns are assembled into an alphabetical list, with each file name being a separate argument to the command.

5. Variable assignment. Actual values are assigned to variable names for storage; for example, *set name = value*. This is the converse of step 1.

The evaluations just described also occur in the wordlist of a *foreach* loop. Only parameter and command substitution occur in the pattern used for a *switch* branch (in parentheses *after switch*); blank interpretation and file name generation do not occur.

These five steps in command-line evaluation occur in the order listed, so if one step produces output that is only evaluated in a previous step, that output does not get evaluated. If, for example, the evaluation of a file name metacharacter results in the name of a command file, such as *ls*, that command is not executed and its output not substituted, because command substitution occurs before file-name generation.

Another aspect of command parsing and evaluation to be aware of is that only one level of evaluation occurs; therefore, the following three command lines will produce the indicated result:

```
% set two = one
% set three = '$two'
% echo $three
$two
```

$three is evaluated to its value, *$two*, but the evaluation does not go any further. The result of one evaluation, *$two*, is not evaluated to see if it is a variable with a value. You can get the shell to interpret another level of variable or command substitution with the *eval* command. Try the following command lines, which demonstrate the *eval* command, at your terminal:

```
% set two = `ls -CF`
% set three = '$two'
% eval echo $three
```

The result of the three previous command lines is as follows:

If the third command line, *eval echo $three*, is changed as follows, the result is the execution of command line *ls -CF*; a listing of the files in your current directory.

```
% eval eval echo $three
```

In general, the *eval* command evaluates its arguments and treats the results as input to the shell. The shell then reads the input and executes any commands. Try the following example command lines:

```
% set wg = "eval who | grep"
% $wg fred
```

These two command lines are equivalent to the following command line:

```
% who | grep fred
```

In the preceding example, *eval* is required because there is no interpretation of metacharacters, such as |, following variable evaluation. This is another example of the specific order of command-line parsing.

The following example sets up variables containing command substitutions, then executes a command calling those variables. The parsing process is shown at each step, with intermediate results:

```
% set user = '/usr'          # Variable user gets name of the system directory /usr
% set dirs = '$user /*bin'   # Variable dirs gets a string of two words
% ls `eval $dirs`            # Note the accent graves indicating command substitution
```

Step 1. Positional parameter and variable evaluation.

$0 gets the string "ls"
$1 gets the string "eval $dirs"

Step 2. Command substitution. The string value of $dirs is substituted.

Command line becomes: ls $user /*bin

Step 3. Blank interpretation. The substituted string value of *$dir* becomes two arguments.

Command line becomes: ls $user /*bin

Step 4. File-name generation. The shell searches for all file names that match string */*bin*, coming up with two standard system files.

Command line becomes: ls $user /bin /lbin

Step 5. Variable assignment. The assigned value of variable *$user* is substituted in for the variable.

Command line becomes: ls /usr /bin /lbin /usr/bin

In addition to the backslash and single-quote quoting mechanisms, there is a third quoting mechanism using double quotes. Within double quotes, parameter and command substitutions occur, although file-name generation and the interpretation of blanks do not, just as with single-quotes. Try the following two examples, comparing their output (in italics):

```
% set ship = Titanic
% echo '$ship'
$ship
% echo "$ship"
Titanic

% echo 'This directory is: `pwd`'
This directory is: `pwd`
% echo "This directory is: `pwd`"
This directory is: /usr/john
```

To prevent variable and command substitution within double quotes, use the backslash metacharacter.

Example:

```
% echo "You can\'t have \$99.00."
You can't have $99.00.
```

The following are characters that have a special meaning within double quotes and can be quoted using \:

Character	Meaning
$	Parameter substitution
*	Command substitution
"	Ends the quoted string
\	Quotes special characters $, *, ", and \

Figure 5-1 shows, for each quoting mechanism, which shell metacharacters are evaluated.

		metacharacter					
		\	$	*	`	"	'
quoting mechanism	'	-	-	-	-	-	t
	`	y	-	-	t	-	-
	"	y	y	-	y	t	-

- = Not interpreted
t = Terminator
y = Interpreted

Figure 5-1. Quoting Mechanisms and Metacharacter Interpretation

5.2.3 A SAMPLE SHELL SCRIPT TO SEARCH FOR PATTERNS IN FILES

You can use the methods presented in subsection 5.2 to speed up and simplify repetitive searches. Writing a shell script to perform multiple searches will make them easier and faster for you to do. Such shell scripts can even make it possible for people who do not know UNICOS to do these tasks, with simple menus such as the **menu** script in subsection 5.1.2.4.

The sample shell script presented here will search the file you specify for matches to all of the patterns listed in an input pattern file. The patterns in the pattern file must all be on one line, separated by spaces, although you may use the backslash to protect carriage returns, allowing you more than one line of patterns in the pattern file. The shell script also asks you for the name of the target file to search.

```
#!/bin/csh
echo "What is the name of the file you want to search?"
set searchfile = $<
echo "What is the name of the file you want to create to contain the results of the search?"
set resultfile = $<
echo "What is the name of the file containing the patterns to search for?"
set patternfile = $<
set patterns = `cat $patternfile`
foreach apattern ($patterns)
    grep $apattern $searchfile >> $resultfile
end
```

Note the append redirection in the next-to-last line of the shell script. Append redirection must be used inside a loop so that each successive iteration does not overwrite the output file with the output from only the latest iteration (as it would if > were used instead of >>).

You can enhance this script to ask for options to the *grep* command (such as -i, ignore case). You can also make it search several files for patterns, by using an input file containing the names of the files to be searched; this is analogous to way this script uses a file containing several search patterns. You could also use this general idea to perform multiple substitutions in a file (or files), using two input files; one for the old strings to search for and one for the new replacement strings.

5.3 CHANGING THE SHELL ENVIRONMENT: PREDEFINED SHELL VARIABLES

The shell environment is the set of characteristics determining how you interact with the shell; how it appears to you. Examples of such characteristics are what sort of system prompt is displayed, where you can access the value of variables, and how your terminal is defined for the system.

UNICOS lets you modify these characteristics and many others, with special predefined shell variables called *environment variables*. You use these variables as you would any variables that you define, but these are interpreted by the shell, have effects on it, and most have default values if you do not specify them.

5.3.1 ENVIRONMENT VARIABLES

Environment variables are pre-defined shell variables, often with uppercase names, that effect your shell environment. They have the following properties:

- These variables (except for prompt) are maintained with the *setenv* command, rather than the set command, so their values are accessible in subshells. (They are in your environment.)
- You can define, change, and access their values just as you do any of the variables that you define, as discussed in subsection 5.1.2.1, Named Variables and in subsection 5.1.2.2, Availability of Variables: Scoping Rules and Commands.

- If you change their values and then log off the system, when you log on again the variables will have returned to their original values, because they are system defaults.

- You can set up your login shell so that some environment variables automatically take on the values you want when you log in or create a new shell (discussed in subsection 5.3.3, The .login and .cshrc Files).

There are many of these variables that control the shell environment, but this subsection will only discuss five of the most commonly used ones.

The *csh* entry in the UNICOS User Commands Reference Manual, publication SR-2011, has a more detailed description of precisely how the C shell exports and defines predefined and environment variables.

5.3.1.1 The HOME variable

The *HOME* variable contains the full path name of your home directory; the directory that you are always located at when you first log in. This is the directory that you go to when you use the *cd* command with no arguments.

To see this type the following command lines at your terminal. The responses you see to the last two command lines will be the same:

```
% cd
% pwd
% echo $HOME
```

You should not change this variable without storing its value in another variable of your own and restoring the correct value of HOME when you are done. One occasion where you might want to do this is if you are doing work in a directory other than your home directory and the work requires a great deal of switching to other directories. In such a case, it would be convenient to be able to just type *cd* to return to your primary working directory. The following command lines let you do this:

```
% set realhome = $HOME
% setenv HOME new_directory
```

The *new_directory* is the full path name of the directory that will be your temporary home base for the work you are doing. When you are done working in this new directory, be sure to use the following command line, if you are going to do any more work on the system:

```
% setenv  HOME  $realhome
```

The *HOME* variable is also useful in making your shell scripts more portable. If you write a shell script that references files in your home directory, you cannot use that script in any directory other than your home directory, because it will not be able to locate those files. Suppose that you have a script that must be able to concatenate the contents of file **stuff** in your home directory. To make this script portable, so you can execute it from other directories on the system, use the following line in the script:

```
% cat  $HOME/stuff
```

If **stuff** were in a subdirectory, **project**, of your home directory, you would use the following:

```
% cat  $HOME/project/stuff
```

5.3.1.2 The PATH variable

The *PATH* variable contains a series of path names that end in directories and are separated by colons. The shell uses these path names to search for the files containing the commands that you type. The shell searches these directories in the order in which they are specified in the *PATH* variable. To see what paths are in your *PATH* variable, type the following:

```
% echo $PATH
```

Most system commands are in directories **/bin, /usr/bin, /usr/ucb**, and */usr/lbin*, so these directories are usually specified in the *PATH* variable. Additionally, if you write many of your own shell scripts and use them more often than most system commands, you may want to create a subdirectory, **bin**, in your home directory, placing all of your shell script files in it. Then, if you want that directory searched first for commands (like shell script names), you can that directory name to the beginning of your *PATH* variable with the following command line (parentheses enclose a word list):

****

CAUTION

If, as shown following, you add **$HOME/bin** *before* the system directories (in $PATH), you must be careful about naming your shell scripts. If you name a shell script with any system command name (such as *ls, rm, pwd.* and so on), the shell cannot access that system command, because it first checks your **$HOME/bin** directory, and finds a command there by that name. Further, if your script (say it is named **ls**) calls a system command that has the same name (*ls*), the result is something like an infinite loop. See subsection 3.2.4, Files of Commands: Shell Scripts, for more information about this.

****

```
% set PATH = ($HOME/bin $PATH)
```

5.3.1.3 The SHELL variable

The *SHELL* variable contains the full path name from the root directory to the executable file that runs your shell. For the C shell, this path name is as follows:

```
/bin/csh
```

This is the path name that you have been typing at the beginning of all your C shell scripts. You are simply telling the system where to locate the appropriate executable shell to run your shell script.

5.3.1.4 The prompt variable

The *prompt* variable contains the system prompt *string*. By default, this is simply % for the C shell. You can set this to anything you want, to specify what machine you are on, what day it is (substitute the output of the date command), or other values.

```
set prompt = "CRAY2$ "
set prompt = `date`
```

Setting the *prompt* variable is a way to see how subshells that are invoked, either explicitly with the *csh* command or automatically to run commands, are separate environments from your login shell (subsection 3.2.3.3, Commands for Background Processing: ps and kill). Write, but do not execute, the following shell script, **prmt**:

```
#!/bin/csh
echo $prompt
```

At the command line type the following line:

```
% echo $prompt
```

The response will be the percent symbol (%), which is the default prompt value. Now run the shell script **prmt**. Your output should be as follows:

```
prompt: Undefined variable.
```

Because the prompt variable is defined with the *setenv* command, rather than the *set* command, it is not defined in any subshells, such as the one automatically created to run the script, **prmt**. This can be useful in keeping subshells distinct if you create a number of different subshells (perhaps with different environments for different uses). You can set the prompt in each subshell to indicate the level of that subshell.

Example:

```
% csh                        # Create subshell from login shell
% set prompt = 'sub1% '      # Set prompt to indicate shell level
% csh                        # Create subshell from within subshell
% set prompt = 'sub2% '      # Set prompt to indicate shell level
. . .
```

5.3.1.5 The TERM variable

The *TERM* variable contains a string value that tells the shell what kind of terminal you have. You would only want to reset this if you logged in from a different terminal. The strings which to set *TERM* are frequently site-specific, so if you have occasion to reset this variable, you will need to ask your system administrator for the system's name for the terminal you want to define.

5.3.2 RENAMING SHELL COMMANDS: THE alias COMMAND

The alias mechanism of the C shell allows you to define a command or series of commands and associate a name with it. You can then execute the command(s) by typing the name. In this way, you can tailor the C shell commands to mimic other operating systems, or customize it to perform commands uniquely suited to your needs.

Aliases can do all of the operations that shell scripts can, including using positional variables ($1, $2, ...), command substitution, and so on. The general format of the *alias* command is as follows:

```
alias name 'command_list'
```

The *name* is the name you assign to the *command_list*. The *command_list* must be in single quotes and may be a simple command or a series of commands separated by semicolons, and it may involve pipes of commands, I/O redirection, and may reference other aliases. Try the following examples at the shell prompt:

```
% alias dir  'ls -al'
% dir

% alias lsfile 'echo "directory:  `pwd`" > dirfile ; dir >> dirfile'
% lsfile
% cat dirfile | pg

% alias sayhi  'echo "hello there, $1"'
% sayhi Freida
```

A useful alias is your own *rm* command that is an alias of the *rm -i* command line. This provides a safeguard against accidentally removing files; the **-i** (interactive) option asks your for a yes or no answer before removing a file. Similarly, you may want to alias the *cp* command to always use the **-i** option, to prevent accidentally overwriting files. Another useful alias, if you use the *vi* editor, is to alias *vi* to its **-r** (recover) option. This will always recover files, before editing them, so that if there is a system crash while you are editing, when you log back on, typing *vi* will automatically recover the file for you. The command lines that set up these aliases are as follows:

```
alias rm 'rm -i $*'
alias cp 'cp -i $*'
alias vi 'vi -r $1'
```

Unlike shell scripts, aliases are not files and are not stored on disk; they are memory-resident parts of the shell. Aliases therefore are faster to execute, but they are also impermanent; when you log off, they are lost. If there are aliases that you use particularly often and want as permanent features of your shell environment, you can have them automatically defined each time that you log on by putting their definitions in your **.cshrc** file.

5.3.3 THE .login AND .cshrc FILES

The **.login** file is a file of commands and environment variables that is automatically executed each time that you log on to the system. Use the cat command to see what is in your **.login** file. You can add shell scripts to be executed or aliases or variable definitions to this file to automatically perform commands or set variables or options each time that you log on to the system. Except for environment variables, however, the definitions in this file are NOT available in any subshells; whether they are created explicitly with *csh* or created automatically to run shell scripts.

The **.cshrc** file is similar to **.login**, but it is executed every time that a C shell is created (including login). It can contain environment variables such as u*prompt*, and alias definitions, as well as shell scripts to execute. Because it is executed every time a C shell is created, the values and definitions in **.cshrc**, unlike those in **.login**, are available in subshells.

****

CAUTION

If you are uncertain of the meaning or purpose of some of the lines in .login or .cshrc, DO NOT modify them. Because these files are automatically executed when you log in, incorrectly altering them can lock up your terminal or cause other problems until your system administrator redefines your.login and .cshrc files.

****

5.3.4 SHELL INVOCATION OPTIONS

When a shell is invoked (created) it looks for options that specify what sort of environment it is to set up. Three of these options, **-x**, **-v**, and **-n**, were discussed in subsection 5.1.1, Basic Shell Script Debugging: Tracing Mechanisms.

There are three ways that C shells are invoked. First, there is the login shell which is automatically invoked when you log in. Next, there are subshells invoked for the commands and shell scripts that you run. Finally, you can explicitly invoke C shells with the *csh* command, as discussed briefly in subsection 3.8, Changing Shells.

When you explicitly invoke (create) subshells with *csh*, you can specify options to the command. You can then add the name of a command or shell script as a final argument, to execute that command/script in a subshell with particular characteristics that you specify with the options. The general format of such a command line is as follows:

csh *-options name*

In the preceding command line, *options* is one or more shell invocation option letters and *name* is the name of the command or shell script that you want to run in the specified shell environment. To set options in your login shell use the following line in your .login file:

source /bin/csh *-options*

The following is a list of the more common options:

Option	Description
-c *fname*	The shell, immediately upon invocation, reads commands from a file named *fname*.
-e	The shell terminates upon detecting any command execution errors (nonzero exit status).
-f	The shell starts faster because it does not execute the .cshrc file before startup.
-i	The shell is interactive. This is the normal default mode.

Option	Description
-n	The shell ignores all commands upon having this option set.

****

CAUTION

Setting the **-n** option at the command line locks up your terminal, causing it to ignore all input, including commands to log out. Use this option only in shell scripts and subshells that terminate themselves.

****

-s	The shell reads commands from standard input (terminal). This is the normal default mode.
-t	Causes the shell to read and execute one line of input.
-v	The shell is in verbose mode, echoing all commands, uninterpreted, to the screen as they are executed.
-x	The shell echoes all commands, interpreted, to the screen as they are executed.
-V	Sets the verbose mode before **.cshrc** is executed, letting you check the commands in **.cshrc** as they are executed.
-X	Sets the **-x** option before **.cshrc** is executed, letting you check the commands in **.cshrc** as they are executed.

If the -v or -V options are set, the *set* command with no options displays the word *verbose* in the list of set variables.

If you are doing a great deal of debugging and do not want to have to type the command line, *csh -options scriptname*, to debug scripts, you can set shell options for the duration of your login session. You do this by combining two commands; the *csh* command which creates a shell with the specified options set, and the *exec* command which executes the specified command in place of the shell in which you invoke it. You can therefore replace your login shell with a different shell that has different options set, with the following command line:

exec csh *-options*

It is not necessary to replace your login shell. You can instead use the *csh* command alone, creating a subshell with the tracing mechanisms set, and leaving your login shell intact. Use the following command line to accomplish this:

csh -vx

When you want to get out of debugging mode and execute shell scripts normally, just press CONTROL-d to exit this subshell and return to your login shell.

5.4 DEBUGGING SHELL SCRIPTS

Subsection 5.1.1, Basic Shell Script Debugging: Tracing Mechanisms, discussed the shell's basic debugging tool, the tracing mechanisms. This subsection covers two more complex debugging features of the C shell, error handling and signal handling.

5.4.1 ERROR HANDLING AND COMMAND EXIT STATUSES

The shell handles errors in different ways depending on the type of error and on whether the shell is being used interactively. UNICOS's definition of an interactive shell is one that has its input and output connected to a terminal as determined by the system call *ioctl* (see the *ioctl* entry in the UNICOS System Calls Reference Manual, SR-2012). Any shell invoked with the -i option is interactive.

Execution of a command can fail for any of the following reasons:

- Input or output redirection can fail if, for example, an input file does not exist or an output file cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally; for example, with a Bus Error or Memory Fault signal.
- The command terminates normally but returns a nonzero exit status.

In all of these cases, the shell goes on to execute the next command, be it in a shell script or from the command line of a terminal. All other errors cause the shell to exit from a command procedure.

When any command executes, it returns an *exit status*, zero or nonzero, to the shell. You do not see this number, but you can test for it with the logical command combining metacharacters, && and || (discussed briefly in subsection 5.1.3.1, Evaluating Conditions: Shell Expressions).

The && and || metacharacters can be used to combine two or more commands, executing successive commands depending on the exit statuses of preceding ones. The && metacharacter executes a following command only if the exit status of the preceding command is zero (completely successful). The || metacharacter is the converse, executing a following command only if the exit status of the preceding command is nonzero.

A zero exit status indicates that the command completed normally with no problems. Other exit statuses have other meanings. A command can have a nonzero exit status without producing an error message if it terminates normally, but without accomplishing its task. An example of this is the following *grep* command:

grep *pattern file*

If *pattern* is not found in *file*, this is neither an error, nor a successful completion; therefore, the command terminates with no error message and a nonzero exit status. You can use this as an implicit conditional test. For example, suppose you are searching for a particular line in a file. You know that it contains one of two unique patterns, but not which one. The following command line will locate that line:

% grep *pattern1 file* || grep *pattern2 file*

If the first search does not find a line containing *pattern1*, it will have a nonzero exit status (though no error message is returned), causing the || metacharacter to execute the second search.

Another use for these metacharacters is to test system conditions for simple yes/no answers, as the following command line does:

```
% who | grep victoria > /dev/null  &&  echo "Victoria is logged on."
```

Because we are not interested in the actual output of the search, that output can be redirected to the special system file **/dev/null**. This is a sort of system wastebasket for unwanted output. Write the following shell script, **loggedon**:

```
#!/bin/csh
who | grep $1 > /dev/null
echo $?
```

Use the following syntax to call the script, substituting the names of various users on your system for *name*:

```
% loggedon name
```

You cannot see the output of the search this way, but you will know whether or not the person is logged on, because a zero result indicates the search succeeded (the person is logged on) and a nonzero result indicates that the search failed (the person is not logged on).

5.4.2 UNICOS SIGNALS

A signal is the mechanism that UNICOS uses to notify a process (an executing command or shell script) that something has happened to influence the execution of that command/script. Signal types are indicated by numbers. UNICOS has 27 defined signals and another 32 available for users. These signals are defined in the *system header file*, **/usr/include/signal.h**, and they are listed in appendix D, UNICOS Signals.

5.4.3 USING THE INTERRUPT SIGNAL: THE onintr COMMAND

With the *onintr* command, you can specify the actions to be taken when a shell script receives the interrupt signal. The interrupt signal is system-specific. Often you enter it by pressing CONTROL-c, but check with your system administrator to know for certain how to enter the interrupt signal on the system you use.

The *onintr* command is usually specified at the beginning of a shell script, because it does not take effect in the script until the *onintr* command is reached in the flow of execution. The following is one format of the *onintr* command, where the ellipses indicate the body of a shell script:

```
onintr label
...
label:
command_list
exit
```

The *label* is a unique string that must have a colon after it and be on a line by itself. The *label* marks the beginning of a section of commands that you want executed when the script receives an interrupt command. The *command_list* can be any number of commands, either on separate lines or separated by semicolons. Pipes of commands and I/O redirection are also allowed in this command list. An *exit* must terminate the command list so that the shell script ends; this is not strictly necessary, if you do not want your script to end when you send it interrupt signals. If, however, you do not have it exit, you will not be able to interrupt execution of that

script except with the *kill* command. The *label* should be at the end of the shell script, because once execution begins at the first command following the *label*, it continues to the end of the script.

You can use *onintr* to remove temporary files that the shell script creates during its execution or to save intermediate results in a file. Saving intermediate results can be very useful, if you have a long shell script running and you get a system broadcast from the administrator that the system is going down soon.

Create the following shell script, **interr**, which demonstrates the action of this format of the *onintr* command:

```
#!/bin/csh
onintr savedata
set count = 0
while ($count < 10000)
    @ count ++              # Increment counter by 1
end
savedata:
echo $count > countfile
exit
```

When you type the interrupt signal, there may be a delay of several seconds before the system responds and stops the shell script. Run the shell script and interrupt it (usually CONTROL-c), then look at file **countfile**, to see at what count the shell script was interrupted.

Another format of the *onintr* command is as follows:

```
onintr -
```

This format causes the shell script to ignore all interrupt commands (not usually a good idea, particularly if there is any chance the script can get into infinite recursion or loops).

The last format of the *onintr* command is as follows:

```
onintr
```

This restores the shell to performing its default action in response to interrupts, which is to terminate shell scripts and return to the command line.

5.4.4 USING SIGNALS WITH THE kill COMMAND

In the C shell, the *kill* command can be used to send any of the 22 defined UNICOS signals to a process (and executing command or shell script). In this way, you can terminate or effect your executing shell scripts in many different ways. This format of the *kill* command is as follows:

```
kill -sig pid
```

The *sig* is the number, 1 through 23, of any of the UNICOS signals, or it can be the name of a UNICOS signal. The name of a UNICOS signal is the name given in the list in appendix D, stripped of the first three letters, SIG. For example, the name of the interrupt signal is INT.

The *pid* is the process id number of the executing process to which you want to send the signal. These numbers are output by the *ps* command.

Suppose that the following is the output from the *ps* command on some system:

```
PID   TTY    TIME   COMMAND
5419  tty06  0:08   sh
5967  tty06  0:01   sh
5968  tty06  0:01   longjob
5969  tty06  0:00   ps
```

To interrupt *longjob*, a user could type either of the following two command lines:

```
kill -INT 5968
kill -2 5968
```

5.5 REPEATING PREVIOUS COMMANDS: THE HISTORY MECHANISM

The C shell *history mechanism* gives you the capability to quickly reexecute a previous command line or recall parts of it to create a new command line. This is particularly helpful when you use long path names or complex command lines involving redirection, pipes, or other operations.

Every command line that you type at the command-line prompt (% is the default) is stored in a buffer called the *history list*. You must explicitly set this buffer each time that you log in, letting the system know how large you want the buffer to be (how many command lines you want saved). This is often done in a user's .login file. The format for this definition is as follows:

```
set history = count
```

The *count* is an integer specifying the number of lines that you want to save. Set your history buffer now to a value of 10 with the following command line:

```
set history = 10
```

The history buffer is maintained on a first-in, first-out basis; that is, only the most recent commands are saved. When you enter the *count+1* command on the command line, it becomes the last entry in the history buffer and the very first command line that you typed is removed from the history buffer.

Each command in the history buffer has a unique number associated with it, known as an *event number*. These numbers count from the very first command entered into the history buffer. Only the most recent *count* number of commands are saved, but their numbers reflect the total number of commands that you have entered during this logon session. To see your list of commands and their event numbers, type the following:

```
history
```

Enter several commands now (*date, ls, who*, and so on), periodically typing the *history* command to see the list of commands. As you do this, the numbers in the history buffer count upwards. When you type the eleventh command line, the first command, *set history = count*, is the first command in the buffer is number 2 and the last command is number 11.

You can use the event numbers to retrieve previous command lines. This retrieval uses the special history command character, !, followed by the event number of the command line that you want to repeat. Assume that you have a history buffer that looks like this:

```
1    set history = 5
2    ls
3    who
4    date
5    history
```

Typing the following command line will produce the indicated response:

```
% !4
Wed Mar 25 07:58:17 CST 1987
```

The history list will now appear as follows:

```
2    ls
3    who
4    date
5    history
6    date
```

You can also use the ! character with a subtraction to repeat a previous command. Given the immediately preceding history list, the following command line will produce the indicated response:

```
% !-2
3    who
4    date
5    history
6    date
7    history
```

Before executing the !-2, the current event number was seven, because six commands had already been executed. Consequently, subtracting two from the current number produced an event number of five, which reexecuted event number five, the *history* command.

If you use event numbers often, you may find it useful to set your prompt equal to the current event number. Given the most recent preceding history list, the following command would set the prompt and produce the indicated response, where the final "10%" is the next system prompt:

```
% set prompt = '% '
9%

9% history
5    history
6    date
7    history
8    set prompt = '% '
9    history
10%
```

Another way of referencing previous commands with the ! character is to specify a character string. This executes the most recent command that begins with the indicated character string. Assume that you have the following history list:

```
1   set history = 5
2   ls -l
3   pwd
4   ls -CF
```

The command *!ls* reexecutes the command line, *ls -CF*. The command *!cd* produces the error message, *cd: Event not found.*

You can also retrieve portions of previous commands, such as some or all of their arguments. The history mechanism has a number of special characters that retrieve arguments from previous command lines. Not only does the shell retrieve the arguments, but it immediately tries to execute them. Consequently, you must use the history commands within a new command line that will correctly interpret the arguments they return. The argument retrieval commands are as follows:

!*N*^	Retrieves the first argument of the *N*th command line
!*N*$	Retrieves the last argument of the *N*th command line
!*N**	Retrieves all arguments of the *N*th command line
!*N*:*m*	Retrieves the mth argument of the *N*th command line
!!	Reexecutes immediately preceding command line

Practice using the *history* command with these special characters. Type the *history* command now, to determine what your current event number is (remember to add 1 to the last event number in the history list). Enter the following series of commands, substituting the event number (corresponding to the command line "cat doc temp > bigfile") for *N* (you must have the two files **doc** and **temp** that you created in section 3, Beyond the Basics):

```
% cat doc temp > bigfile
% cat !N:1
cat doc
% cat !N:2
cat temp
% !N:0 bigfile
cat bigfile
% echo "!N*"
echo "doc temp > bigfile"
doc temp > bigfile
```

APPENDIX SECTION

A. UNICOS BATCH FACILITIES

This appendix provides a brief user-level introduction to the use of the Network Queuing System (NQS), which is the batch facility available with UNICOS. It also supplies a number of brief user examples, then refers the reader to the appropriate Cray publications which provide more specific information. This appendix is divided into the following subsections:

- Overview of NQS
- Getting started with NQS
- Using Cray station software to submit an NQS batch file

For NQS administrative information, see the UNICOS System Administrator Guide for CRAY-2 Computer Systems, publication SG-2019, or the UNICOS System Administrator Guide for CRAY X-MP and CRAY-1 Computer Systems, publication SG-2018.

A.1 OVERVIEW OF NQS

The nature of the batch environment provided by UNICOS is of great importance to many customers using Cray computer systems. Several standard UNIX batch facilities, including commands, utilities, and system calls, have been ported to UNICOS:

Facility	Function
&	Runs a process in the background
cron	Executes a command at the specified time and date
at	Schedules a command list at some time
kill and killall	Terminate a process from another process

NQS lets you submit, terminate, monitor, and, within limits, control batch requests submitted to the batch system. You can send batch requests to your own system (the local host) or to other appropriately configured computer systems in your network (remote hosts).

Specifically, NQS lets you perform the following activities:

- Submit requests to a batch queue with *qsub*.
  The *qsub* command lets you specify a number of qualifications for your batch request, including start time, memory and CPU resource limits, exporting of environment variables, and the queue to which the request is submitted. See the UNICOS User Commands Reference Manual, publication SR-2011, for more information about the *qsub* command.

- Display the status of NQS queues with *qstat*.
  The *qstat* command displays information about NQS queues and requests. Figure A-1 provides an example of the information you might receive from *qstat* about a request (cftjob) submitted with *qsub*.

The -l option, specifying "long format," has been included on the *qstat* command line in the example. By default, *qstat* displays the following information about a request: the *request-name*, the *request-id*, the owner, the relative request priority, and the current request state. For running requests (like the one in figure A-1), the UNICOS job identifier group is also shown, as soon as this information becomes available to the local NQS daemon.

When the -l option is specified, however, *qstat* output also shows the time at which the request was created, an indication of whether or not mail will be sent, and the user name on the originating machine. If *qstat* is examining a batch queue (as in figure A-1), it also shows resource limits, the planned disposition of standard error and standard output, any advice concerning the command interpreter, and the user file-creation mode mask (umask).

See the UNICOS User Commands Reference Manual, publication SR-2011, for more information about the *qsub* command.

```
-------------------------------
cray2: NQS BATCH REQUEST SUMMARY
-------------------------------
```

REQUEST NAME	IDENTIFIER	OWNER	QUEUE	JID	PRTY	REQMEM	REQTIM	ST
batjob1	9844.cray2	dge	A_little		31	20	10	q
batjob2	9257.cray2	dge	A_medium	1123	20	135	432	R
batjob3	9855.cray2	dge	A_monster		10	256	10000	q

Figure A-1. Example of *qstat* Output for an NQS Batch Queue Summary

- Delete or signal NQS requests with *qdel*.
  The *qdel* command lets you delete requests from NQS queues or signal all of the processes associated with a request by using the UNICOS signal mechanism. For more information, see the *signal* entry in the UNICOS System Calls Reference Manual, publication SR-2012, or the *qdel* entry in the UNICOS User Commands Reference Manual, publication SR-2011.

- Display the status of NQS devices with *qdev*.
  The *qdev* command displays the status of devices known to NQS. If no devices are specified, *qdev* displays the current state of each NQS device on the local host. See the UNICOS User Commands Reference Manual, publication SR-2011, for more information about the *qdev* command.

- Display supported batch limits and shell strategies for each host with *qlimit*.
  The *qlimit* command displays the batch request resource limit types that NQS can directly enforce. When you attempt to queue a batch request, each specified limit-value is compared against the limit-value configured for the destination batch queue. If the batch queue limit-value is greater than or equal to the corresponding batch request limit-value, your request can be successfully queued. The *qlimit* command also displays the batch request shell strategy defined by your system administrator. See the UNICOS User Commands Reference Manual, publication SR-2011, for more information about the *qlimit* command.

- Submit a hardcopy print request to NQS with *qpr*.
  The *qpr* command places files in an NQS queue to be printed by a device such as a line printer or a laser printer. See the UNICOS User Commands Reference Manual, publication SR-2011, for more information about the *qpr* command.

The following UNICOS data transfer commands are often used in NQS batch files:

acquire Stages a file from the front-end computer system to UNICOS. The *acquire* command searches the Cray system for a file of the same name before staging the file from the front-end system to the Cray computer. If the file is found on the Cray system, the file is not staged from the front-end system.

dispose Transfers a file from UNICOS to a front-end computer system. By default, the file is transferred to the front-end system that originated the transfer request.

fetch Stages a file from a front-end computer system to UNICOS. It does not first check the UNICOS file system for a file of the same name, as does the *acquire* command. The *fetch* command does not delete the file from the UNICOS file system after job completion. UNICOS does not prevent two users from staging files to the Cray system at the same time and with the same path name. If two users attempt this, the file that is transferred last will overwrite the first one transferred.

A.2 GETTING STARTED WITH NQS

The first step in using NQS is to create a shell script of commands to execute the sequence of actions to by performed by the batch request. Within this file you can include flags that modify the *qsub* user command, as long as the flags appear before the shell commands and are preceded by #, @, and $ characters. For example, the following batch request file specifies that the request is to be sent after 11 p.m. on Tuesday and that a list of current system users is to be produced:

```
#@$-a 11pm Tuesday who
```

The following is a slightly more complex example of a batch request file:

```
#
# Slightly more complex
#
# @$-q queue1 # Queues request to queue1
# @$-lt 00:01:00 # Specifies a per-process CPU limit of 1 minute
#
qstat -l
cd junk
ls -l
cc -o test.homer ralph.c
test.homer
ls -l
rm test.homer
ls -l
```

You can use two methods to include data on which the shell commands must act. The first method is to create a separate data file to which the command is linked, as follows:

```
sort < sortinput
```

In this example, **sortinput** contains lines of text that you want sorted. The second method of including data within your batch request is to use the here document operator as follows:
(For information on the here document operator, see either subsection 4.1.4, or 5.1.4, Shell Scripts Containing Their Own Input: Here Documents)

```
sort << EOF Robert Cohn was once middleweight boxing champion of Princeton. EOF
```

This example sorts the specified lines of text. In contrast, the following shell script will not sort these lines:

```
sort Robert Cohn was once middleweight boxing champion of Princeton.
```

In this example, the *sort* command encounters an immediate end-of-file indicator when reading the standard input file **stdin**, which defaults to **/dev/null.**

After creating a batch request file, use *qsub* to send the batch request for execution. The *qsub* command lets you specify several controling factors, including per-process CPU time limits, time the batch request begins execution, and the queue to which the batch request is submitted. For example, the following command submits file **request1** to queue1 at 11 p.m. on the following day, and exports all environment variables:

```
qsub -a "11pm Tom." -q queue1 -x request1
```

Specify limits no larger than those required to execute your request, because NQS uses these limits for batch request scheduling. For example, batch requests requiring large amounts of CPU time are generally run less often than batch requests requesting small amounts of CPU time.

The *qsub* command also lets you interactively enter the commands to be executed by the batch request. Exclude the script-file from the *qsub* command line and press RETURN. All lines that you enter in the standard input buffer are then executed as the batch request. Signal the end of the standard input file with a CONTROL-d, as follows:

```
$ qsub -a "11pm Tom." -q queue1
ls
who
(CONTROL-d)
$
```

If your batch request is successfully submitted, NQS returns a message that displays the request id and destination queue of your batch request. For example, the following message indicates that NQS assigned your request a sequence number of 125, you are working on machine MH-VAX, and your request was sent to queue1.

```
Request 125.MH-Vax submitted to queue: queue1.
```

By default, NQS also assigns a request-name to your batch request. The default NQS request name is equivalent to the name of the script file you specified on the command line. The request name and the request-id are associated with your request throughout the network.

After submitting the request, use the *qstat* command to display the queue and batch request status, as follows:

```
qstat queue1
```

If this command is entered soon after the previously submitted batch request is sent, it produces the following output:

```
------------------------------
cray2: NQS BATCH QUEUE SUMMARY
------------------------------
QUEUE NAME       LIM TOT ENA STS QUE RUN  WAI HLD ARR EXI  QUEUE COMPLEXES
---------------  --- --- --- --- --- ---  --- --- --- ---  --------------
A_little          10  14 yes  on   3  10    0   1   0   0  comp1 comp2
A_medium           5   4 yes  on   0   3    1   0   0   0  comp1
A_monster          2   5 yes off   4   0    0   0   1   0  comp2
---------------  --- --- --- --- --- ---  --- --- --- ---  --------------
    <TOTAL>       15  23              7  13    1   1   1   0
---------------  --- --- --- --- --- ---  --- --- --- ---  --------------
```

Figure A-2. Example of *qstat* Output for an NQS Batch Request

See the *qstat* entry in the UNICOS User Commands Reference Manual, publication SR-2011, for more information on displaying the status of NQS batch requests.

Use the *qdel* command with its **-k** option, to delete a running NQS batch request. To delete a batch request, you must be the owner of that request, unless you have super user privileges or are an NQS manager. To delete a batch request, specify the request-id, as follows:

```
qdel 4.cray2
```

This command deletes the batch request with request-id 4.cray2.

You can also use the *qdev*, *qpr*, and *qlimit* commands to display the status of NQS devices, submit a hardcopy request, and display the NQS resource limits, respectively.

After the batch request completes processing on the executing machine, the output and error messages files are returned, by default, to your home directory on the originating machine. By default, the name of the output file contains the first 7 characters of the request name, followed by the characters **.o**, followed by the request sequence number of the request id. The error messages file has the same naming convention, except that the second set of characters begins with **.e**. For example, the output file of the batch request submitted above is named **request1.o4**, and the error messages file is named **request1.e4**.

You can also submit NQS batch job files using Cray station software facilities, as described in the next subsection.

A.3 USING CRAY STATION SOFTWARE TO SUBMIT NQS BATCH FILES

The following three general steps outline the process for using Cray station software to submit an NQS batch job from a local computer system to UNICOS. The next subsection supplies a specific example of the three steps necessary for submitting the batch job with the IBM/VM station.

The station manuals listed in appendix C, Cray Station Publications, provide specific details on using NQS in your environment.

1. *On the front-end computer system, prepare the job file or files that will be submitted to UNICOS.* The first element in the job file must be the NQS statements that provide explicit instructions for UNICOS on how to process the job. The job files you create on the front-end system can contain other elements, such as code and data to be used by the program.

2. *Use the job submission station command to send the job file or files from the front-end computer system to the Cray computer system for processing.* You can use other station commands to monitor and modify the job as it runs. The UNICOS job output is returned to the front-end computer system and placed in a location specified by the station software.

3. *Use the front-end computer system to manage the output from the submitted job.* For example, you can read the file at your terminal, print a copy of it, or modify it if it contains errors.

A.4 SUBMITTING A BATCH JOB FROM THE IBM/VM STATION

This subsection presents an example of a Fortran job submitted to a Cray mainframe from an IBM/VM station, following the steps outlined in Figure 4-1. Examine the file shown, then read the explanation that follows it, for information about what each of the lines in the file does. The line numbers in this file are only for reference purposes; the files you create cannot contain line numbers.

Step 1. Preparing the job and data files on your IBM/VM station

To perform this step, you will need the following batch job file, named **crayjob** of type **job** on your IBM/VM front-end system. Before you actually create the following file on your local IBM/VM station (without the line numbers), read through the explanation in the paragraphs following:

```
1    # @$user=u4407 pw=bombo
2    # @$-r crayjob
3    fetch for.f -nfor -t'ft=t'
4    fetch indata -nindata -t'ft=data'
5    touch outdata
6    ln indata fort.7
7    ln outdata fort.8
8    cft77 for.f
9    segldr for.o
10   chmod 700 a.out
11   a.out
12   dispose outdata -noutdata -t'ft=data'
```

Lines 1 and 2 contain NQS directives, with the information necessary to submit your job (naming it **crayjob**) to the Cray mainframe.

Lines 3 and 4 contain the NQS *fetch* commands that copy your Fortran source file, **for.f**, and datafile, **indata**, from your IBM/VM station to the Cray mainframe.

Line 5 contains the UNICOS *touch* command that creates a blank file, **outdata** to contain the output of your Fortran program.

Lines 6 and 7 use the UNICOS *ln* command to link the Fortran units **fort.7** and **fort.8** to the UNICOS files, **indata** and **outdata**.

Line 8 compiles the Fortran program file, **for.f**, creating the binary object file, **for.o**.

Line 9 loads the Fortran object file produced by line 8, and produces the executable file, **a.out**.

Line 10 uses the UNICOS command *chmod* to give you execute permission for the executable file, **a.out**.

Line 11 executes the Fortran program in the executable file, **a.out**, produced by line 9.

Line 12 disposes the UNICOS output file **outdata**, containing the output of the Fortran program, to your IBM/VM front-end computer.

In addition to the job file preceding, you need to create two other files on your IBM/VM front-end system. These are the Fortran program file, named **for** and of type **F**, and the input datafile, named **indata** and of type **data**. These two files are shown following

Fortran program file, **for.f**:

```
      PROGRAM TEST
      INTEGER IARR(10),JARR(10)
      READ(7,*)(IARR(I),I=1,10)
      DO 10, J = 1, 10
          JARR(J)=IARR(J)/2
10    CONTINUE
      WRITE (8,*) (JARR)(K),K=1,10)
      END
```

Input data file, **indata**:

```
1,3,7,9,6,5,2,4,8,10
```

Step 2. Using IBM/VM station commands to submit and monitor the job

To submit your batch job file, **crayjob** to the Cray mainframe from your IBM/VM station, use the following station command:

```
crsubmit crayjob
```

The system responds with output similar to the following:

```
PUN FILE 2561  TO  S218     COPY 001   NOHOLD

R; T=0.06/0.15 11:20:36
```

To check on the status of your job, you can use the *crstatus* station command, which provides output similar to the following:

```
11:29:57              Cray  System  Status     (E I O R S)      S218       Frame  1
                                    CSDN = nqs_job
Jobname    Seq  DC Status            Class     Pri    FL      CPU   Limit   MF   TID
-------  -----  -- ----------------  -------  ----  -----  ------  ------   --  --------
U4407    22298  IN RUNNING           little   8.0     14      17  ******   V4  U4407
JUNK4    22300  IT send              O-strm   0.0    ---     ---     ---   V4  U4734
JUNK3    22301  IT send              O-strm   0.0    ---     ---     ---   V4  U4734
JUNK2    22302  IT send              O-strm   0.0    ---     ---     ---   V4  U4734
JUNK     22303  IT send              O-strm   0.0    ---     ---     ---   V4  U4734
                                     End Of Data
```

Figure A-3. Example of Output from *crstatus*

When your job is done and the Cray system sends your output file back to your virtual reader, you will get a message similar to this:

```
R; T=0.02/0.10 11:21:18

PRT FILE 2569 FROM S218    COPY 001   NOHOLD
PRT FILE 2571 FROM S218    COPY 001   NOHOLD
```

To transfer the output file from your reader to permanent storage, type *receive* at your terminal and you will see a response similar to the following:

```
File OUTDATA DATA A received from S218 at MHVM1 sent as OUTDATA DATA A

R; T=0.06/0.17 11:21:52
```

Step 3. Use your front-end computer system to manage the output from the submitted job

Once the data file **outdata** has been disposed to your IBM/VM front-end station, you can read, print, or modify it as you would any other file, using xedit and your local printing command.

B. UNICOS INTERACTIVE COMMUNICATIONS FACILITIES

This appendix supplies a brief introduction to the following products, which provide communication with UNICOS:

- TCP/IP, which runs on most UNIX systems, allows Cray computer systems to participate as peers in TCP/IP network environments. TCP/IP provides file transfer applications, virtual terminal access, and tools upon which distributed networking applications are built.

- Station software provides access to proprietary protocol implementations (SNA, DECnet, CDCNET, and so on) through gateways resident on front-end systems. Station software provides data conversion and reformatting to allow the Cray system to act as a natural extension to your environment.

Because detailed information about system communication is supplied in other CRI publications, this appendix provides a brief introduction, referring you to the appropriate manuals for more specific information.

See the TCP/IP Network User Guide, publication SG-2009, for complete user information regarding the TCP/IP network. For complete user information on Cray station products, see the list of publications in appendix C.

B.1 THE TCP/IP PROTOCOL

The Transmission Control Protocol/Internet Protocol (TCP/IP) is a set of computer networking protocols that allows two or more individual computers, called *hosts*, to communicate over a network. TCP/IP was originally defined by the Defense Advanced Research Projects Administration (DARPA), an agency of the U.S. Department of Defense (DoD). It is now a government standard implemented on a wide variety of computing equipment.

TCP/IP includes a variety of commands that provide communication facilities, not just with Cray computer systems, but with any systems that support the TCP/IP protocol.

TCP/IP provides the following basic communications facilities (the commands associated with them are listed also, and they can be found in the UNICOS User Command Reference Manual, publication SR-2011):

- Remote login and execution (*telnet*, *rlogin*, and *remsh*)
- File transfer utilities (*ftp* and *rcp*)
- Network mail

For example, the *telnet* and *rlogin* utilities let you use hosts on your network as if they were directly connected to your terminal. The computer system to which your terminal is hard-wired is your *local host;* the Cray computer system and other computer systems on the network are considered *remote hosts.*

The *telnet* utility uses the DoD TCP for reliable virtual terminal communication. One important virtue of *telnet* is that it can connect you to any host on your network that supports DoD standard TCP/IP, regardless of the resident operating system. Actual use of a remote host, however, requires knowledge of its operating system, because *telnet* does not perform command translation.

When you use *telnet*, the remote host may prompt you for your login name and password. It then checks the system authorization before granting you access.

In the following example, the user is prompted for a login name on the *name* host.

```
$ telnet name
login: loginame
password:

$
```

The *name* argument to the *telnet* command is specific to each system and is set by your system administrator. It is the name by which your front-end computer system knows the Cray mainframe. See your system administrator for the *name* of the Cray system(s) at your site that you want to log on to. After typing this command, you will be at the logon screen of the Cray system as discussed in subsection 2.1.1. Logging in, where *loginame* is explained.

B.1.1 THE ftp COMMAND

The *ftp* command is the user interface to the ARPANET standard File Transfer Protocol (FTP). This lets users transfer files to and from a remote network site. Use the *ftp* command as follows:

```
ftp name
```

The *name* argument is the same as the *name* you would use with the *telnet* command. This name is system-specific, so if you do not know what it is, ask your system administrator.

Once you have logged in to the Cray mainframe with the *ftp* command, you can use a number of commands specific to *ftp* to accomplish file transfers. Some of the more commonly used *ftp* commands include the following:

bye Terminate the FTP session with the remote server and exit *ftp*.

close Terminate the FTP session with the remote server, and return to the command interpreter.

delete *remote-file*
Delete the file *remote-file* on the remote machine.

get *remote-file* [*local-file*]
Retrieve the *remote-file* and store it on the local machine. If the local file name is not specified, it is given the same name it has on the remote machine.

mdelete [*remote-files*]
Delete the *remote-files* on the remote machine.

mget *remote-files*
Expand the *remote-files* on the remote machine and do a *get* for each file name thus produced.

mput *local-files*
Transfer multiple *local-files* from the current local directory to the current working directory on the remote machine.

put *local-file* [*remote-file*]
Store a local file on the remote machine. If *remote-file* is left unspecified, the local file name is used in naming the remote file. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

quit A synonym for *bye*.

recv *remote-file* [*local-file*]
A synonym for *get*.

send *local-file* [*remote-file*]
A synonym for *put*.

user *user-name* [*password*] [*account*]
Identify yourself to the remote FTP server. If the password is not specified and the server requires it, *ftp* will prompt the user for it (after disabling local echo). If an account field is not specified, and the FTP server requires it, the user is prompted for it. Unless *ftp* is invoked with "auto-login" disabled, this process is done automatically on initial connection to the FTP server.

B.1.2 THE rcp COMMAND

The *rcp* command performs a remote file copy between two machines. The allowed syntaxes for the command are as follows:

rcp *file1 file2*
rcp [-r] *file ... directory*

The *rcp* command copies files between machines. The *file* arguments may refer to remote files, local files, or directories; arguments may consist of either absolute or relative path names. Remote files are specified in the form *rhost:file*, where *rhost* is a remote hostname or alias (described in *hosts*(4N)). The local file name *file* may not contain a colon (:) unless it is preceded anywhere in the name by a slash (/).

If the -r option is specified and any of the source files are directories, *rcp* copies each subtree rooted at that name; in this case the destination must be a directory.

If *path* is not a full path name, it is interpreted relative to your login directory on *rhost*. A path name on a remote host may be quoted (using \ ", or ') so that metacharacters are interpreted remotely.

The *rcp* command does not prompt for passwords; your current local user name must exist on *rhost* and must permit remote command execution via *remsh*(1).

The *rcp* command handles third party copies, where neither source nor target files are on the current machine. Hostnames may also take the form *rhost.rname* to use *rname* rather than the current user name on the remote host.

B.2 USING STATION SOFTWARE

If you are using a station software product to communicate with UNICOS, you should have access to a set of publications describing how to use that product with UNICOS. This section is not intended to replace the publications listed in appendix C, Cray Station Publications, but rather to provide a brief introduction to UNICOS station facilities.

Station software products are a means of connecting Cray computer systems to those of other vendors. CRI provides communications software products that link UNICOS to the following front-end environments:

- DEC VAX/VMS
- IBM MVS
- IBM VM
- CDC NOS
- CDC NOS/BE
- Apollo DOMAIN
- AT&T UNIX System V and Berkeley UNIX 4.2BSD running on several hardware systems, including the following:

  AT&T 3B20
  DEC VAX
  Pyramid
  Sun Microsystems
  Hewlett-Packard
  Silicon Graphics IRIS workstation

These products provide connectivity and extensive functional capabilities, including the following:

- Interactive access
- File transfer
- Job transfer, status, and control

Stations can be used to integrate a Cray system into a multivendor, multiple-protocol environment. Stations are connected using either the direct channel front-end interface (FEI) attachment, or the Network System Corporation (NSC) HYPERchannel attachment for multiple connections.

Interactive access through station software can be accomplished by entering a single station command, followed by entering your UNICOS user name and password. For example, you can initiate and terminate a UNICOS interactive session through the VAX/VMS station as follows:

```
$ cint
 CINT> inter

Welcome to Mendota as tty00

login:  xxx
password:  xxx

Be sure to read the news --

$ <CTRL/Z>

CINT> quit
bye
CINT> exit
$
```

Several additional interactive station features let you perform special functions, depending on the version of station software you are using. For example, the UNIX station interactive facility lets you redirect output from a UNICOS command to a file on a UNIX front-end system.

The following is a list of station commands used to start an interactive UNICOS session from different local computer systems.

Station	Command	
UNIX	ias	
VAX/VMS	cinteractive	
IBM/MVS	cray	
Apollo	CINT	
NOS	HELLO,ICF /LOGON	(note the *two* lines required)
IBM/VM	crint	

Station software also provides batch job submission to UNICOS. As a station user, the first step in submitting a batch job through the station is to prepare a shell script containing NQS commands, shell commands, and program data. (NQS is the UNICOS batch system.) See appendix A or the publications listed in appendix C, Cray Station Publications, for information on submitting a batch job to UNICOS with a Cray station.

If you are accessing UNICOS through station software on a front-end system, you can transfer files with a set of UNICOS commands that provide for data transfer between systems. You can include these commands in a batch file or you can enter them during an interactive session. The data transfer commands are *acquire*, *dispose*, *fetch*, and *scpqsub*. These commands conform to the syntax of UNICOS commands and are executed by your UNICOS shell.

The *acquire* command transfers a file from the front-end computer system to UNICOS, searching for a file of the same name before transferring the file to Cray memory from the front-end system. If the file is found on the Cray system, the front-end file is not transferred. Depending on how your site installs USCP, you may have to specify the full path name of this command, which is **/usr/bin/acquire**.

Format:

acquire sdn -l *localpath* [-i *termid*] [-m *mainframe*] [-d *dc*] [-f *fm*][-t'*text field*']

sdn	Name of the source file on the front-end system
localpath	Location of the file on UNICOS; can be a relative or full path name. This is a required parameter.
-i *termid*	Terminal identifier for front-end verification. The default is the terminal identifier from the front-end system from which the job was submitted or through which the interactive session was initiated.
-m *mainframe*	A 2-character, front-end identifier; the default is the originating mainframe.
-d *dc*	Disposition code. All COS disposition code values are supported; the default is PR.
-f *fm*	File format; it can be one of the following:
	CB Character blocked; the file is sent to the front-end system in COS blocked format. CB is the default.
	TR Transparent; the data is treated as a continuous byte string.
	UD UNICOS data; the format is mixed-case ASCII character data common to most UNIX systems.
-t'text field'	Front-end specific text to be passed to the front-end system. See the station reference manual for your

The *fetch* command transfers a file from the front-end computer system to UNICOS. The *fetch* command does not check the UNICOS file system for the existence of the specified file before sending the transfer request to the front-end system and it does not delete the file from the UNICOS system following job completion.

The *dispose* command transfers a file from UNICOS to a front-end computer system. By default, the file is transferred to the front-end system from which the request originated. Depending on how your site installs USCP, you may have to specify the full path name of this command, which is **/usr/bin/dispose.**

The *scpqsub*(1) command lets you spawn additional UNICOS jobs from within the original UNICOS job sent to the Cray computer system. You can route the output of the job to the front-end system of your choice. This process is called *recursive submission.*

For more information on the format of *fetch*, *dispose*, and *scpqsub*, see the station reference manual specific to your front-end system.

B.3 STATION SOFTWARE EXAMPLE PROGRAMS

The following examples provide an illustration of interactive and batch jobs you can run through station software. This subsection assumes that you are using a UNIX front end and the UNIX station; you will have to make minor modifications to the programs (mainly in the text fields of the *acquire*, *fetch*, and *dispose* commands) to make these programs run on your own front-end system.

The examples are shown in both interactive and batch mode. The interactive examples assume that you are already logged on interactively to UNICOS.

B.3.1 FILE TRANSFER EXAMPLES

Transferring a file from the front-end system to reside as a file on UNICOS is accomplished with the *acquire* command, as follows:

```
$ acquire data1 -mTZ  -t'/u/xyz/data1'
```

The **data1** argument is the UNICOS file to which the data is transferred; unless you specify a full path name, the file is placed in your current directory. The **-m** option specifies the mainframe id for your front-end system. The **-t** option specifies the front-end text, or the file on the front-end to be transferred.

The equivalent batch job is as follows:

```
JOB,JN=EX1.                              #USCP job
ACCOUNT,AC=xxx,US=xxx,UPW=xxx.           #USCP account validation
#   @$-r ex1                             #Job card
#   @$-eo                                #Combine standard error and output
#   @$-lt  10                            #Time limit of 10 seconds
acquire data1 -mTZ -t'/u/xyz/data1'
```

The following interactive command transfers a file from UNICOS to a front-end system:

```
$ dispose data1 -mTZ -dST -t'/u/xyz/data1'
$ exit
```

This command transfers a copy of file **data1** to **/u/xyz/data1**; if the front-end file already exists, it is overwritten. The format is similar to *acquire*, except that the **-d** option specifies the ST disposition code. The *exit* command causes the login prompt to reappear on the screen.

The following batch file has the same effect as the previous interactive command:

```
JOB,JN=EX2.
ACCOUNT,AC=xxx,US=xxx,UPW=xxx.
#   @$-r  ex2
#   @$
dispose  data1  -mTZ -dST -t'/u/xyz/data1'
```

B.3.2 JOB SUBMISSION EXAMPLES

This subsection provides examples on four types of programs in which you will use station software to submit jobs. The programs describe how to run a job on a Cray system when the source exists on UNICOS, on the front-end system, and in various combinations. The subsection assumes that you understand how to run a UNICOS Fortran program; if you do not, see subsection 3.5, Fortran Programs Under UNICOS.

The following batch file runs with both the program and data existing as files on the front end:

```
JOB,JN=EX3.
ACCOUNT,AC=xxx,US=XXX,UPW=xxx.
#  @$-r  ex3
#  @$
fetch source.f -mTZ -t'u/xyz/source'
fetch data -mTZ -t'u/xyz/data'
cft source.f
segldr source.o        #Output is placed in executable file a.out
a.out  <  data          #Runs a.out and takes input from file data
```

The following batch file runs with the program on UNICOS, and the data on the front-end system:

```
JOB,JN=EX4.
ACCOUNT,AC=xxx,US=XXX,UPW=xxx.
#  @$-r  ex4
#  @$
fetch data -mTZ -t'/u/xyz/data'
cft prog1.f
segldr prog1.o
a.out < data
```

The following batch file runs with the program on the front-end system, and the data on UNICOS:

```
JOB,JN=EX5.
ACCOUNT,AC=xxx,US=XXX,UPW=xxx.
#  @$-r  ex5
#  @$
fetch prog.f -mTZ -t'/u/xyz/prog'
ln data1 fort.10
cft prog.f
segldr prog.o
a.out
```

The following batch file runs with both the program and the data on UNICOS:

```
JOB,JN=EX6.
ACCOUNT,AC=xxx,US=XXX,UPW=xxx.
#  @$-r  ex6
#  @$
ln data1 fort.10
ln prog1 prog1.f
cft -eL prog1.f; cat prog1.l
segldr prog1.o
a.out
```

C. CRAY STATION PUBLICATIONS

The following publications contain detailed user-level information and examples for the indicated Cray station software:

Apollo DOMAIN Station Reference Manual, publication SA-0250

Apollo DOMAIN Station Differences for UNICOS Installations, publication SN-0253

UNIX Station User Guide, publication SU-0107

UNIX Station Summary of Differences for UNICOS Installations, publication SU-0103

CDC NOS Station Reference Manual, publication SR-0035

CDC NOS Station Summary of Differences for UNICOS Installations, publication SN-0237

CDC NOS/BE Station Reference Manual, publication SR-0034

CDC NOS/BE Station Summary of Differences for UNICOS Installations, publication SN-0240

DEC VAX/VMS Station Reference Manual, publication SV-0020

DEC VAX/VMS Station Summary of Differences for UNICOS Installations, publication SN-0239

IBM MVS Station Reference Manual, publication SI-0038

IBM MVS Station Summary of Differences for UNICOS Installations, publication SN-0149

IBM VM Station Reference Manual, publication SI-0160

IBM VM Station Differences for UNICOS Installations, publication SN-0166

D. UNICOS SIGNALS

UNICOS has 27 defined signals and 32 more available for users. These signals are defined in the *system header file*, **/usr/include/signal.h**, as well as being listed here.

Signal Name	Number	Description
SIGHUP	01*	Hangup; user logged off
SIGINT	02*	Interrupt (rubout); user pressed the interrupt key
SIGQUIT	03*	Quit (ASCII FS) user pressed CONTROL-c or the program failed and output a core dump
SIGILL	04*	Illegal instruction (not reset when caught); user requested unknown command (usually a typographical error)
SIGTRAP	05*	Trace trap (not reset when caught); a signal used by program debuggers such as *adb*
SIGHWE	06*	Hardware error (fp table, lm parity, double bit); abort
SIGERR	07*	Error exit
SIGFPE	08*	Floating-point exception; an arithmetic error
SIGKILL	09	Kill signal from the *kill* command (cannot be caught or ignored)
SIGPRE	10*	Program range error
SIGORE	11*	Operand range error
SIGSYS	12*	Bad argument to system call
SIGPIPE	13	Write on a pipe with no one to read it; the user piped a command's output, but specified no recognized command to receive it as input
SIGALRM	14	Alarm clock
SIGTERM	15	Software termination signal from the *kill* command
SIGUSR1	16	User defined signal 1
SIGUSR2	17	User defined signal 2
SIGCLD	18	Termination of a child process; a command and its invoked subshell terminated
SIGPWR	19	Power failure
SIGMT	20	Multitasking wake-up signal
SIGMTKILL	21	Multitasking kill signal
SIGBUFIO	22	Fortran asynchronous I/O completion
SIGRECOVERY	23	Recovery signal (advisory)
SIGUME	24	CRAY X-MP and CRAY-1 systems: Uncorrectable memory error
SIGCRAY8	24	CRAY-2 systems: Reserved for Cray Research, Inc.
SIGDLK	25	CRAY X-MP and CRAY-1 systems: True deadlock detected
SIGCRAY7	25	CRAY-2 systems: Reserved for Cray Research, Inc.
SIGCPULIM	26	CPU time limit exceeded (see *limit*(2))
SIGSHUTDN	27	System shutdown imminent (advisory)

Signals marked with * produce a core dump if they are not caught with the Bourne shell *trap* command.

The following alternative definitions are also available:

SIGIOT	06
SIGEMT	07
SIGBUS	10
SIGSEGV	11

Signals 33 through 64 are available for users.

E. ON-LINE MANUAL SECTION ABBREVIATIONS

To retrieve entries with the *man* command from only one particular section of the UNICOS on-line documentation set, specify the section abbreviation before the name of the entry as follows:

man *section entry*

The following example displays only the *name* command, and not a system call, library routine, or other entry that may have the same name:

man 1 *name*

The complete set of section abbreviations for the UNICOS on-line documentation set are as follows:

1 User command entries in the UNICOS User Commands Reference Manual, publication SR-2011

1bsd User command entries from UNIX 4.2 BSD in the UNICOS User Commands Reference Manual, publication SR-2011

1m Administrator command entries in the UNICOS Administrator Commands Reference Manual, publication SR-2022

2 System call entries in the UNICOS System Calls Reference Manual, publication SR-2012

3c C library routine entries in the CRAY-2 UNICOS Libraries, Macros, and Opdefs Reference Manual, publication SR-2013, for UNICOS running on a CRAY-2 computer system, or in the CRAY X-MP and CRAY-1 C Library Reference Manual, publication SR-0136, for UNICOS running on a CRAY X-MP or CRAY-1 computer system

3db CRAY X-MP and CRAY-1 SYMDEBUG library routine entries in the Programmer's Library Reference Manual, publication SR-0113, for UNICOS running on a CRAY X-MP or CRAY-1 computer system

3f Fortran library routine entries in the CRAY-2 UNICOS Libraries, Macros, and Opdefs Reference Manual, publication SR-2013, for UNICOS running on a CRAY-2 computer system, or in the Programmer's Library Reference Manual, publication SR-0113, for UNICOS running on a CRAY X-MP or CRAY-1 computer system

3io CRAY X-MP and CRAY-1 I/O library routine entries in the Programmer's Library Reference Manual, publication SR-0113 for UNICOS running on CRAY X-MP or CRAY-1 computer systems.

3m Math library routine entries in the CRAY-2 UNICOS Libraries, Macros, and Opdefs Reference Manual, publication SR-2013, for UNICOS running on a CRAY-2 computer system, or in the Programmer's Library Reference Manual, publication SR-0113, for UNICOS running on a CRAY X-MP or CRAY-1 computer system

3n TCP/IP network library routine entries in the TCP/IP Network Library Reference Manual, publication SR-2057

3q CRAY-2 calling sequence entries in the CRAY-2 UNICOS Libraries, Macros, and Opdefs Reference Manual, publication SR-2013, for UNICOS running on a CRAY-2 computer system, or

3rpc RPC library routine entries in the TCP/IP Network Library Reference Manual, publication SR-2057

3s CRAY-2 standard C library routine entries in the CRAY-2 UNICOS Libraries, Macros, and Opdefs Reference Manual, publication SR-2013, for UNICOS running on a CRAY-2 computer system

3sci SCILIB (libsci) library routine entries in the CRAY-2 UNICOS Libraries, Macros, and Opdefs Reference Manual, publication SR-2013, for UNICOS running on a CRAY-2 computer system, or in the Programmer's Library Reference Manual, publication SR-0113, for UNICOS running on a CRAY X-MP or CRAY-1 computer system

3u Utility library routine entries in the CRAY-2 UNICOS Libraries, Macros, and Opdefs Reference Manual, publication SR-2013, for UNICOS running on a CRAY-2 computer system, or in the Programmer's Library Reference Manual, publication SR-0113, for UNICOS running on a CRAY X-MP or CRAY-1 computer system

3w TCP/IP socket compatibility library routine entries in the TCP/IP Network Library Reference Manual, publication SR-2057

3x CRAY-2 Miscellaneous library routine entries in the CRAY-2 UNICOS Libraries, Macros, and Opdefs Reference Manual, publication SR-2013, for UNICOS running on a CRAY-2 computer system

3z CRAY-2 macros and opdef entries in the CRAY-2 UNICOS Libraries, Macros, and Opdefs Reference Manual, publication SR-2013, for UNICOS running on a CRAY-2 computer system

4d Special file (device) entries in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

4f File formats entries in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

4n TCP/IP file and facility entries in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

GLOSSARY

GLOSSARY

The following list defines terms and acronyms used in this manual that may not be familiar.

A

Absolute path name - See full path name.

Argument - Words following the command name on a command line that provide information necessary to execute a program. Command arguments are very often file names.

ASCII - American Standard Code for Information Interchange

B

Background - A mode of program execution when the shell does not wait for the command to terminate before prompting for another command.

Block - A unit of disk storage space equal to 4096 bytes.

Bourne shell - One of two UNICOS command intepreters that acts as an interface between users and the UNICOS operating system. It can be thought of as a layer over the operating system (hence "shell") providing some commands, command interpretations, and utilities as an interface to the operating system. The Bourne shell is derived from AT&T UNIX System V. See also, C shell.

C

C language - A general-purpose, medium-level programming language used to write programs (such as numerical, text processing, and database) and operating systems.

C shell - One of two UNICOS command intepreters that acts as an interface between users and the UNICOS operating system. It can be thought of as a layer over the operating system (hence "shell") providing some commands, command interpretations, and utilities as an interface to the operating system. The C Shell is derived from the 4.2 release of the Fourth Berkely Software Distribution (4.2BSD). See also, Bourne shell.

Child process - A duplicate of the parent process created with the system call *fork*(2).

Command - The first word of a command line. It is the name of an executable file that is a compiled program, shell built-in command, or shell procedure.

Command line - A sequence of nonblank arguments separated by blanks or tabs typed in by a user. The first word usually specifies the name of a command and the other words are options and arguments to the command.

Command list - A sequence of one or more simple commands separated or terminated by a new-line character or a semicolon.

Command procedure - See shell script.

Command substitution - When the shell reads a command line, any command or commands enclosed between accent grave characters, as in `*command*`, are executed first, and the output from these commands replaces the whole expression, `*command*`.

Core file - A file produced by a running process when it receives a particular signal. This file contains a memory image of the process and is named **core**. See appendix C, UNICOS Signals.

Current directory - The directory in which the user is currently located. It is the value returned by the *pwd* command. The current directory is used as the point of reference for accessing data within the file structure when relative path names are used.

D

Delimiter - Any character(s) that mark the beginning and ending of any separate unit: piece of text, program input, or program block; the symbol(s) that mark an item as separate from its environment.

Directory - A type of file that is used to group and organize files and other directories.

E

echo - A UNICOS command that repeats the text arguments given it. It can also evaluate metacharacters, performing variable substitution, and command substitution. See the *echo* entry in the UNICOS User Commands Reference Manual, publication SR-2011, for more information on the *echo* command.

Environment variables - predefined shell variables that determine some of the characteristics of your shell.

EOF - The end-of-file character. EOF is used to terminate a shell; if the login shell is terminated, the user is logged off the system. By default, an EOF is generated at a keyboard by CONTROL-d.

EOT - The end-of-transmission character. This is the same as the ASCII EOF character. See EOF.

esac - Indicates the end of the UNICOS *case* command; no more conditions are listed.

Exit status - A number returned to the system by a command once it completes execution, indicating either that the command executed successfully or that the command encountered some sort of error. See subsection 4.4.1, Error Handling and Command Exit Statuses.

F

File - An organized collection of information containing data or programs or both, which allows users to store, retrieve, and modify information. A file is either a regular file or a special file. A simple file name is a sequence of characters other than a slash (/).

File descriptor - A nonnegative integer returned by the *open*(2) or *create*(2) system call. By convention, UNICOS commands use file descriptor 0 for standard input, file descriptor 1 for standard output, and file descriptor 2 for standard error. Upon login, standard input, standard output, and standard error are assigned to the user's terminal.

Filter - A command that reads its standard input, transforms it in some way, and displays the result on its standard output.

Foreground - A mode of program execution when the shell waits for the command to terminate before prompting for another command.

fork - The system call that duplicates a parent process, creating a child process.

Full path name - The path name between the root directory (/) and a specific file. This is a list of the names of all directories on a direct path between the root directory and the file, beginning with a slash for the root directory, and ending in the name of the file, with slashes between all of the names. See Path name.

G

Group identification number (gid) - A unique number, assigned to one or more logins, that is used to identify groups of related users.

H

Here document - A combination of a command and input to that command that has the following format:

```
command << delimiting_string
text
delimiting_string
```

This causes the shell to read subsequent lines as standard input to the command, beginning on the line following the first occurrence of the *delimiting_string* and continuing to the second occurrence of the *delimiting_string*. You can use any character(s) for the *delimiting_string*.

HOME - Another name for the login directory. It is a predefined shell variable; the value of this variable is accessed with $HOME.

Home directory - The default working directory for a user; the user is placed in this directory after logging in.

I

In-line input documents - See *here* document.

K

Keyword parameters - An argument to a Bourne shell command or script, with the following format, where *name* is the keyword parameter:

```
name = value command arg1 arg2 . . .
```

It allows shell variables to be assigned values when a shell procedure is called. The value of *name* in the invoking shell is not affected, but the value is assigned to *name* before execution of the procedure. The arguments (*arg1 arg2* ...) are available as positional parameters ($1 $2 ...).

L

Link - A link allows a file to be accessed from more than one directory. It is not a copy of the file, but a path to it, like an implicit version (that you cannot see) of a path name that is continuously in effect.

Log in - A procedure to connect a user to a UNICOS system.

Log off - A procedure to disconnect a user from a UNICOS system.

Login - A means by which a user can gain access to a UNICOS system.

Login directory - See Home directory.

Login name - A unique string of letters and numbers used to identify a login

M

Metacharacters - Characters that have a special meaning to the shell, such as <, >, *, ?, |, &, $, ;, (,), , ", ', *, [,]

Mode (of a file or directory) - The permissions for the file or directory (read, write, and execute for owner, group, and other). The mode is referenced by either an octal number (absolute mode) or a sequence of characters (relative mode). The mode is used in conjunction with the *chmod* command to change the permissions of files and directories.

N

New-line character - The character that appears on a screen as a combination of a carriage return and line feed. To indicate a new-line character on an ASCII keyboard, press RETURN. The character is commonly represented in code as \n.

Null string - A string of nothing, specified with empty double or single quotes: "" or ''.

P

Parent directory - The directory immediately above another directory. A ".." is the shorthand name for the parent directory. To make the parent directory of your current directory your new current directory, enter the command "cd ..".

Parent process - A process that has created a child process with the *fork*(2) system call.

Password - A string of 6 to 13 characters chosen from a 64-character alphabet (., , 0-9, A-Z, a-z). If a user choses a password with more than 8 characters, any characters beyond the first 8 are ignored.

Path name - A sequence of directory names separated by the / character and ending with the name of a file. The path name defines the connection path between a directory and a file. See also Full path name and Relative path name.

Pipe - A simple way to connect the output of one program to the input of another program so that each program will run as a sequence of processes.

Pipeline - A series of programs, filters, or commands separated by the character |. The output of each filter becomes the input of the next filter in the line. The last filter in the line writes to its standard output.

Positional parameters - Arguments supplied with a command procedure that are placed into variable names $1, $2, . . . when the command procedure is invoked by the shell. The name of the file being executed is positional parameter $0.

Primary prompt - A notification (by default "$" or "%") to the user that the UNICOS shell is ready to accept another request.

Process - A program that is in some state of execution. The execution of a computer environment including the process' status, (runnable, sleeping swapped), the process ID (PID), the user's ID (UID), the contents of memory (if any), register values, the current directory, status of open files (if any), and various other items.

R

Relative path name - The path name between the current working directory and a specific file. See Path name.

Root directory - The directory at the apex of the hierarchical directory tree structure of the UNICOS file system. It is the single directory from which all other files and directories branch.

S

Secondary prompt - A notification (by default ">" or "?") to the user that the command typed in response to the primary prompt is incomplete.

Shell - A UNICOS program written in the C language that handles the communication between the system and users. The shell accepts commands and causes the appropriate program to be executed. There are two shells in UNICOS: the Bourne shell (*sh*) and the C shell (*csh*). The Bourne shell is the "standard," or default shell; hence "the UNICOS shell" means the Bourne shell. The term "the shell" is used when referring to either shell; used in those cases where the Bourne shell and the C shell are the same.

Shell environment - The set of characteristics that determine how you interact with the shell and how it appears to you. These characteristics include what shell prompt is displayed, where in the directory system you can access variables, and how your terminal is defined for the system.

Shell procedure - See shell script.

Shell script - An executable file that is not a compiled program and is composed of shell commands and input to them. A shell script is a call to the shell to read and execute commands contained in a file. A sequence of commands may thus be preserved for repeated use by saving it in a file that may also be called a shell program, shell script, command procedure, command file, or runcom according to local preference.

sort - A UNICOS command that orders the lines of files alphabetically, numerically, or in ASCII order. See the *sort* entry in the UNICOS User Commands Reference Manual, publication SR-2011, for more information on the *sort* command and its many options.

Standard error - Error messages produced by most commands are sent to an open file, which is normally connected to the printer or screen. Standard error is file descriptor 2; the output may be redirected by an argument to the shell of the form "2>file", which opens the specified file as standard error.

Standard input - The standard input of a command is sent to an open file, which is normally connected to the keyboard. An argument to the shell of the form "< *file*" opens the specified file as the standard input, thus indicating that input is to come from the file named instead of the keyboard. Standard input is file descriptor 0.

Standard output - Output produced by most commands is sent to an open file, which is normally connected to the printer or screen. This output may be redirected by an argument to the shell of the form "> *file*", which opens the specified file as the standard output. Standard output is file descriptor 1.

String - A sequence of keyboard characters bounded by spaces, tabs, or returns.

Super user - A UNICOS user who has special permissions to access and alter the system defaults; usually the system administrator.

T

Trap - The process of using the Bourne shell *trap* command to conditionally perform some action(s), based on the exit status of a preceding command. See subsection 4.4.3, Using Signals: The trap Command.

U

UID - See user identification number.

User-defined variables - A user variable can be defined using an assignment statement of the form *name=value*, where *name* must begin with a letter or underscore and may then consist of any sequence of letters, digits, or underscores up to 512 characters. The *name* is the variable. Positional parameters cannot be in the name.

User identification number (uid) - A unique number assigned to each login that is used to identify users and the owner of information stored on the system.

V

Variables - A variable is a name representing a string value. Variables that are normally set only on a command line are called parameters (positional parameters and keyword parameters). Other variables are simply names to which the user (user-defined variables) or the shell itself may assign string values.

W

Working directory - See current directory.

INDEX

INDEX

READER'S COMMENT FORM

UNICOS Primer SG-2010 B

Your reactions to this manual will help us provide you with better documentation. Please take a moment to check the spaces below, and use the blank space for additional comments.

1) Your experience with computers: ____ 0-1 year ____1-5 years ____5+ years
2) Your experience with Cray computer systems: ____0-1 year ____ 1-5 years ____5+ years
3) Your occupation: ____ computer programmer ____ non-computer professional
____ other (please specify): _____
4) How you used this manual: ____ in a class ____as a tutorial or introduction ____ as a reference guide
____ for troubleshooting

Using a scale from 1 (poor) to 10 (excellent), please rate this manual on the following criteria:

5) Accuracy _____
6) Completeness ____
7) Organization ____
8) Physical qualities (binding, printing) ____
9) Readability ____
10) Amount and quality of examples ____

Please use the space below, and an additional sheet if necessary, for your other comments about this manual. If you have discovered any inaccuracies or omissions, please give us the page number on which the problem occurred. We promise a quick reply to your comments and questions.

Name _____ Address _____
Title _____ City _____
Company _____ State/ Country _____
Telephone _____ Zip Code _____
Today's Date _____

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

Attention: PUBLICATIONS
1345 Northland Drive
Mendota Heights, MN 55120