ADVANCED ConvexOS
Training Course
June 21, 1990

CONVEX Computer Corporation

Table of Contents

Appendices

List of Tables

Preface

Objectives and Intended Audience

The CONVEX UNIX system administrator training materials address technical professionals having prior experience with the UNIX operating system. This document, which is intended as a training aid, contains a copy of the instructor's transparency presentation and examples of the major points. In addition, concepts/functions are reinforced with exercises.

It covers commands and utilities that take the user beyond the basics. The materials illustrate the advanced text manipulation commands, provide instructions for writing and executing C shell scripts, and discusses the use of the CXbatch system. Additionally, the utilities that aid in inter-machine communications are introduced.

Organization

- Chapter 1 provides a review of fundamental commands.
- Chapter 2 introduces commands available for finding files that meet specific criterion.
- Chapter 3 introduces the facilities of the C shell. It provides instructions for writing and executing C shell scripts. A variety of shell script examples are provided.
- Chapter 4 provides detailed instruction for using the text manipulation commands, *sed* and *awk*.
- Chapter 5 introduces the utilities that aid in inter-machine communications. It discusses the use of remote login, file transfer program (*ftp*), and UNIX to UNIX copy (*uucp*).
- Chapter 6 discusses the advantages of using the CXbatch system. Information is provided to set up the user's environment for the CXbatch system. Additionally, user commands are covered.
- Chapter 7 introduces the commands available for archiving files on magnetic tape.
- Chapter 8 illustrates how to submit problem reports to CONVEX via *contact*.
- The appendix contains a summary of basic UNIX commands, *vi* commands, and illustrates awk and shell scripts.

Notational Conventions

The following conventions are used in this document:

- *Italics* within text indicate commands, filenames, or programs.
- Within command sequences and text, **boldface** type indicates literals. Words appearing in **boldface** should be typed just as they appear. *Italics* within command sequences indicate generic commands or filenames. Substitute actual commands or filenames for

the *italicized* words. For example, the command sequence:

ld [*switches*] [*object files*] [*libraries*]

instructs you to type the command *ld*, followed by your choice of switches, object files, and/or libraries.

- Text appearing in `typewriter` font represents the text as it appears on the terminal.

Associated Documents

The following documents, available from CONVEX Computer Corporation, are recommended to the CONVEX UNIX user:

- *CONVEX UNIX Primer* describes and illustrates the use of simple UNIX commands.
- *UNIX Tutorial Papers* contains a section which discusses advanced UNIX applications.

1
Fundamentals

Objectives for Chapter 1

After completing this chapter, you will be able to:

1. Enter a command in the correct format.
2. Correct typing errors when entering a command.
3. Describe the UNIX hierarchical system indicating your directory in the hierarchy.
4. Differentiate between relative and absolute pathnames.
5. Specify the number of command lines you want to display with *history*.
6. Display a listing of previous command lines.
7. Repeat previous command lines or parts of previous command lines.
8. Modify selected words and events of previous command lines.
9. Use appropriate online help function to find information on UNIX commands.
10. Determine your terminal characteristics and modify those characteristics as specified.
11. Add an entry that initializes your terminal (if appropriate) to your *.login* file.
12. Determine environment variables in effect and modify as needed.

COMMAND STRUCTURES

Commands:

- Are case sensitive
- Use the format:

  command -[options(s)] [filenames(s)]

  The options:
  - are single letters prefixed by a dash and followed by a space

    % `ll -d -g /mnt`
  - can usually be combined and prefixed by one dash

    % `ll -dg /mnt`

- Use the semicolon (;) to separate multiple commands on the same line.

  % `cd ; rm -r practice; lf`

- Continue commands to the next line by placing a backslash (\) at the end of the line.

  % `rm -r\`
  `/mnt/snoopy/practice`

Command Structures

Sample command formats:

`lf`	Command only
`date`	Command only
`lf -la`	Command with two options
`lf -l .login`	Command with an option and an argument
`cal 1989`	Command with one argument
`cal 8 1989`	Command with two arguments
`cal 81989`	Argument is invalid; no spaces
`cal 8\` `1989`	Command continued on another line
`date; cal 8 1989`	Multiple commands separated with semicolon (;)

Tryout the *date* command by entering:

date

The screen displays:

```
% date
Tue Aug  8 15:09:09 CDT 1989
%
```

Display the date and a calendar for August by typing:

date; cal 8 1989

Both the date and calendar are displayed:

```
% date; cal 8 1989
Tue Aug  8 15:09:09 CDT 1989
   August 1989
 S  M Tu  W Th  F  S
        1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
%
```

PATHNAMES

- An absolute pathname:
  - Leads from the root directory (/)
  - Includes names of parent directories
  - Contains slashes separating the directory names

    `/docs/andy/proj/data`

- Relative Pathnames:
  - Relative to your current location in the hierarchical structure
  - Begin without a slash
  - Begin with subdirectory name or filename in current working directory

    For example, if your working directory is:

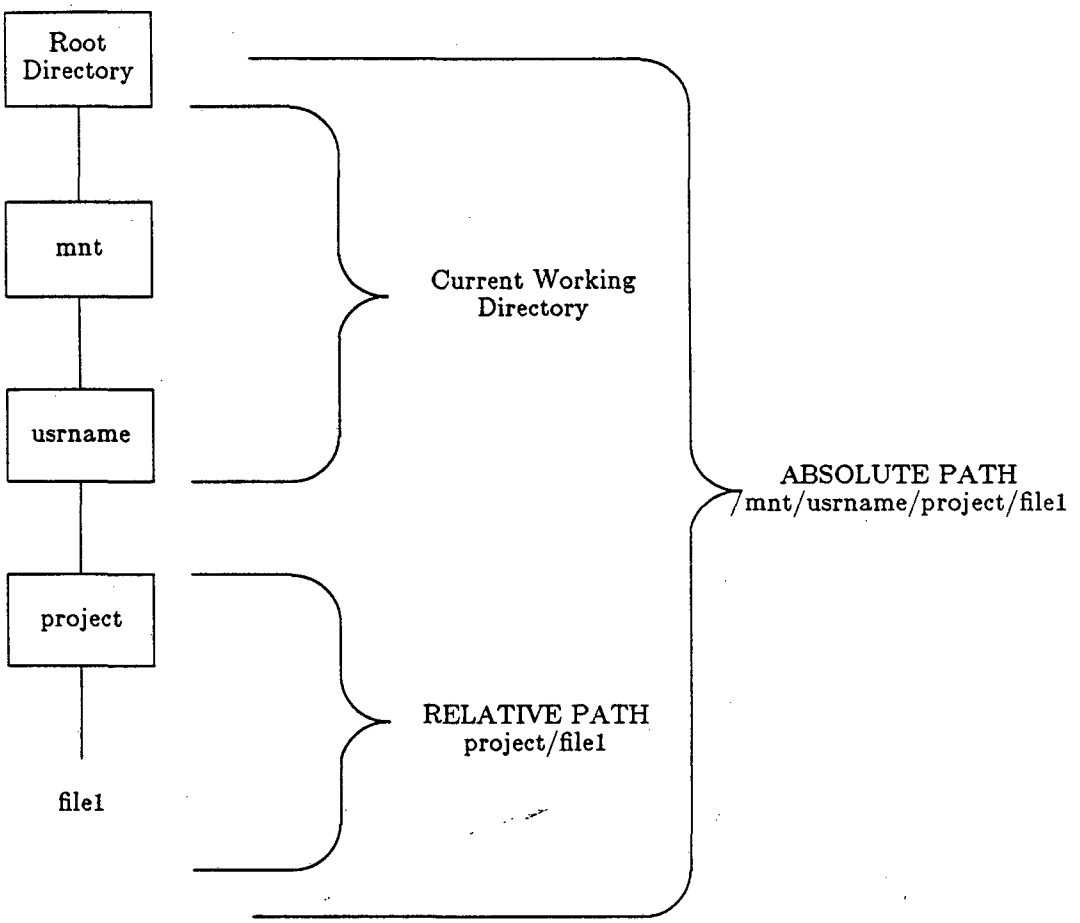
    /docs/andy

    then you can refer to the file *data* in the *proj* subdirectory as:

    `proj/data`

Absolute/Relative Pathnames

Assume that your current working directory is */mnt/usrname*, and that you want to work with a file named *file1* in the directory *project*. What is the relative pathname to the file named *file1*?



FILENAMES AND METACHARACTERS

- Asterisk (*) - matches any number of characters

  % `lf dat*`
  matches *dat*, *date*, *data1*, and *date.old*

- Question Mark (?) - matches any single character

  % `lf file?`
  matches *file1*, *fileA*, and *filed*

- Square Brackets [] - List of ASCII characters inside the brackets matches any character in the list; separate a range of characters with a dash (-).

  % `lf *[135]`
  matches any filenames that end with a "1", a "3", or a "5"

  % `lf [A-Z0-9]*`
  matches any filenames that begin with an uppercase letter or a digit

Using Metacharacters

Use the metacharacters with the *ls* command. First list the file in your home directory that begin with *t*. Your entry:

```
% ls t*
test1   test1.date  test1.echo test2  test3
%
```

List the files *test1, test2, test3.* Your entry:

```
% ls test?
test1   test2   test3
%
```

List all the files in your home directory that end with *1* or *3*. Your entry:

```
% ls *[13]
states1   test1   test3
%
```

List all the files in your home directory that begin with uppercase characters A-E. Your entry:

```
% ls [A-E]*
no match.
%
```

COMMAND LINE CORRECTIONS

Correction keys (before pressing the RETURN key):

- Backspace key or CTRL-h – erases the last character or several characters typed
- CTRL-w – erases the last word you typed
- CTRL-u – erases the entire input line
- Can be changed with the *stty* command in your *.login* file

```
stty kill ^k
```

sets ^*k* as the key sequence to erase the entire input line.

Correcting Typing Errors on the Command Line

Enter the following command incorrectly as shown; do not press the RETURN key.

daet

Use the BACKSPACE key or CTRL-h to erase the *et*; then type the **te**. When you strike the BACKSPACE key or CTRL-h two times, your entry looks like:

```
% daet (incorrect entry)
% da (entry after deletions)
% date (corrected entry)
Tue Aug  8 15:09:09 CDT 1989
%
```

Enter the following command incorrectly as shown; do not press the RETURN key.

cal 8 29866

Use CTRL-w to erase the last word typed; change the year to **1986**. Your entry looks like this:

```
% cal 8 29866 (incorrect entry)
% cal 8 (entry after deleting 29866)
% cal 8 1986 (entry after making correction)
   August 1986
 S  M Tu  W Th  F  S
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
%
```

Enter the following command but do not press the RETURN key.

daet; col 2986

Erase the entire entry with the CTRL-u key. Your entry will look like this:

```
% daet; col 2986 (incorrect entry)
% (deletion leaves a blank line following the prompt)
```

COMMAND LINE EDITING

- Using ^ at the beginning of a line indicates you wish to modify the preceding command

- Provide a search/replace pattern; use a ^ before and after the search string, followed by the correct characters

- Substitution occurs at the first instance of the pattern

- To change *pc* to *cp* in the command:

```
% pc file1 file2
```

at the *c shell* prompt (%), type:

```
^pc^cp
```

`cp file1 file2` is displayed and the command is executed

Command Line Editing

Enter the following command (press RETURN after entering the command):

who | sort | moer

Correct the command by substituting the correct pattern; your entry looks similar to:

```
% who | sort | moer
moer: Command not found
% ^er^re
who | sort | more
user1   tty01   Sep 23 10:16
user2   tty02   Sep 23 10:13
user3   tty04   Sep 23 10:10
user4   tty05   Sep 23 08:09
. . .
. . .
%
```

HISTORY FUNCTION

- The ! character alerts the c shell that you want to edit or execute a previous command line that has been saved in a *history* buffer

- To execute previous commands (at the c shell prompt):

!!	Executes the last command
!4	Executes command 4
!l	Executes the most recent command beginning with 'l'
!-4	Executes command that goes back four commands
!?s?	Executes the most recent command pattern with 's'

- To make an addition to the preceding command (last command executed), use ! ! followed by additional arguments or commands

```
% !! | less
```

- To make an addition to a specific command, use ! *command number* followed by the additional arguments

```
% !4 | less
```

Repeating History Events

Repeat the last command (*who | sort | more*) you entered. Your entry looks similar to:

```
% !!
who | sort | more
user1   tty01   Sep 23 10:16
user2   tty02   Sep 23 10:13
user3   tty04   Sep 23 10:10
user4   tty05   Sep 23 08:09
...
...
%
```

Change your history setting to 10; then do a history listing. Your entry looks similar to:

```
% set history=10
% history
 7 date
 8 wc states1
 9 history
10 set history=5
11 history
12 who | sort | moer
13 who | sort | more
14 who | sort | more
15 set history=10
16 history
```

Repeat event 8 (*wc states1*). Your entry looks like:

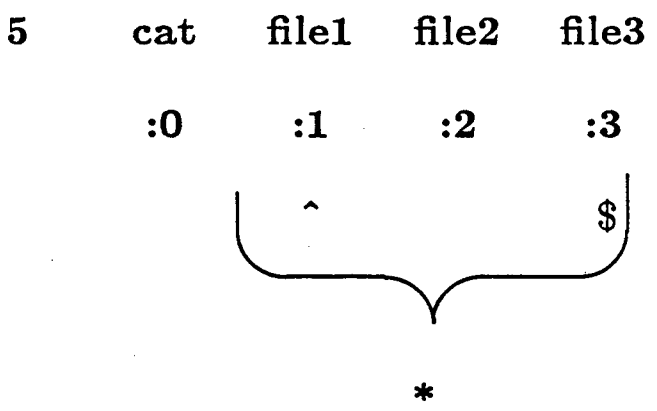
```
% !8
wc states1
  5  5  37 states1
%
```

Send the output of the previous event (*wc states1*) to a file named *mod*. Your entry looks similar to:

```
% !! > mod
wc states1 > mod
% cat mod
Maine
Montana
Nebraska
Illinois
Iowa
%
```

HISTORY FUNCTION (cont.)

- *csh* divides a command into separate "words" at spaces or tabs:



- History references for this command could be:

```
% !5              cat file1 file2 file3
% more !5:2-3     more file2 file3
% ex !5:$         ex file3
% more !5^        more file1
% lpr !5:2        lpr file2
% lpr !5*         lpr file1 file2 file3
```

History Word Identifiers

Do a listing of the previous commands. Your entry:

```
% history
10 set history=5
11 history
12 who | sort | moer
13 who | sort | more
14 who | sort | more
15 set history=10
16 history
17 wc states1
18 wc states1 > mod
19 history
%
```

Select a previous event (*wc states1 > mod*) that allows you to edit the file *mod*. Add this line to the *mod* file:

The numbers indicate lines, words,
and characters contained in the file.

Your entry, using the *vi* editor:

```
% vi !18:$
vi mod
"mod" 1 line, 33 characters
                The numbers indicate lines, words,
                and characters contained in the file.
                .
:wq
"mod" 3 lines, 123 characters
%
```

Modify the same event (18); display *states1* using the *more* command. Your entry:

```
% more !18:1
more states1
Maine
Montana
Nebraska
Illinois
Iowa
%
```

HISTORY FUNCTION (cont.)

- To specify substitutions for a previous command line:
  - Identify the history event
  - Supply a search/replace pattern
  - Use a ^ before and after the search string, followed by the correct characters

- For example, to change the *filename* of event 4 to *myfile*:

```
4 sort filename | awk '{print $1, $2}' | lpr
% !4:s^filename^myfile
```

Other delimiters may be used instead of ^:

```
% !4:s/filename/myfile
```

- To verify that the command is correct before executing it, use :p immediately following the event identifier

```
% !4:p:s/filename/myfile
4 sort myfile | awk '{print $1, $2}' | lpr
```

If the command is correct, to execute it, type:

```
!!
```

History Substitutions

Change event 14 (*who* | *sort* | *more*) so that it prints the sorted output of *who*. Verify that your command is correct; then execute it. Your entry:

```
% !14:p:s/more/lpr
% !!
who | sort | lpr
%
```

THE *info* COMMAND

- Provides information by topic on ConvexOS
- Provides pointers to written documentation
- Provides online examples of ConvexOS commands
- Has the form:

  % `info`

  - Main menu displays
  - Make selection by number or command name
  - Exit with `q`

Using *info*

Invoke the *info* system. Select number **6** from the main menu. Then select number **1** on the submenu "CHECK USER, JOB, OR SYSTEM STATUS." Explore the various topics listed on the "CHECK USER STATUS" menu. After viewing several topics, exit *info*.

The first time you execute *info*, you will see four screens of introductory information; then the main menu is displayed. The following example assumes that you have previously viewed the introductory information.

```
%info
                        CONVEX INFO SYSTEM MAIN MENU

 1. Contact other users or machines
 2. Use UNIX online help
 3. Execute commands
 4. Edit, find, print, modify, analyze,
    and archive files
 5. Develop programs
 6. Check user, job, or system status
 7. Modify system and file accessibility
 8. Perform arithmetic calculations
Enter <1..8>, <q>uit, <?>help, <t>opic/command list, a command name, a topic
Please type in your selection and press <RETURN>: 6

CHECK USER, JOB, OR SYSTEM STATUS Submenu of Main Menu

 1. Check user status
 2. Check job status
 3. Check system status

Please type in your selection and press <RETURN>: 1

CHECK USER STATUS - Check User, Job, or System Status Submenu

 1. Get personal user information
 2. List group memberships for a user
 3. Show the most recent user logins
 4. Show the most recently executed commands
 5. Remind yourself when you need to leave
 6. Display the current directory name
 7. Display your login shell (Bourne or C shell)
 8. Display disk usage and limits
 9. Summarize disk usage
10. List users on the system
11. Print the pathname of the user's terminal
12. Display your environment settings

Please type in your selection and press <RETURN>: q
%
```

THE *man* COMMAND

- Displays online documentation about ConvexOS commands
- Has the form:

man *command*

% man cat

```
CAT(1)          ConvexOS Programmer's Manual          CAT(1)

NAME
     cat - catenate and print

SYNOPSIS
     cat [ -u ] [ -n ] [ -s ] [ -v ] file ...

DESCRIPTION
     Cat reads each file in sequence and displays it on the
     standard output.  Thus

          cat file

     displays the file on the standard output, and

          cat file1 file2 >file3

     concatenates the first two files and places the result
     on the third.
```

(more output appears in the actual display)

Displaying Manual Pages

View the manual page for *man*. Your entry:

```
% man man
MAN(1)          UNIX Programmer's Manual              MAN(1)



NAME
   man - find manual information by keywords; print out the
   manual

SYNOPSIS
   man -k
   man -f
   man [ - ] [ -t ] [ section ] title ...

DESCRIPTION
   Man is a program which gives information from the program-
   mers manual.  It can list one-line descriptions of commands
   specified by name, or all commands whose descriptions con-
   tain the specified keyword.  It can also provide online
   access to sections of the printed manual.
...
...
%
```

THE *man -k* COMMAND

- Displays the page that includes the specified keyword

- Has the form:

  man -k *keyword* or apropos *keyword*

  % man -k print or apropos print

```
banner (see banner(6))     - print large banner on printer
bigcal (see bigcal(1))     - print out calendar of month
cal (see cal(1))           - print calendar
cat (see cat(1))           - catenate and print
cpr  (see cpr(1))          -  print 'C' files
date (see date(1))          - print and set the date
```

  (more output appears in the actual display)

Accessing Manual Pages

Display a listing from the man pages for the topic *editors*. Your entry:

```
% man -k editor
a.out (see a.out(5))    - CONVEX assembler and link editor output
ed (see ed(1))          - text editor
emacs (see emacs(1))    - a screen editor
ex, edit (see ex(1))    - text editor
ld (see ld(1))          - link editor
sed (see sed(1))        - stream editor
vi, view (see vi(1))    - screen oriented (visual) display editors based on ex
%
```

THE *man -f keyword* COMMAND

- Lists all commands that have *keyword* as an index in the man pages
- May also be invoked as *whatis*
- Prints a one-line description (from table of contents of "man page") for the specified *keyword*
- Prints the "man page" name for the keyword, including its section number
- Has the form:

  `man -f` *keyword*

  - *keyword* can be:
    - a command
    - a system file
    - a device name
    - a system call
- To display a one-line description of *print*:

  % `man -f print`

  `print (see print(1))- pr to the line printer`
- The *whatis keyword* command is the same as *man -f keyword*

Online Manual Page Command Descriptions

Use the *man -f* command to determine where you can find information on the command *tty* in the "man pages," as well as display a one-line description of the command. Your entry:

```
% man -f tty
tty (see tty(1))      - get terminal name
tty (see tty(4))      - general terminal interface
%
```

Use the *whatis command* to describe *emacs*. Your entry:

```
% whatis emacs
emacs (see emacs(1))  - a screen editor
%
```

ENVIRONMENT VARIABLES

- Initialized by login
- Maintained by the shell
- Passed to all commands and programs run from within the current *shell*
- Store information that certain programs need to know about
- Usually in uppercase (to distinguish them from ordinary *shell* variables):

PATH	HOME
TERM	TERMCAP
SHELL	USER
PRINTER	EDITOR
PAGER	LESS

- To display a list of environment variables and their values, type: `printenv`

Environment Variables

Display a listing of the environment variables. Your entry:

```
% printenv
HOME=/mnt/username
SHELL=/bin/csh
PATH=.:/mnt/username/bin:/usr/convex:/usr/ucb:/bin:/usr/bin
TERM=vt100n
USER=username
TERMCAP=d0|vt100n:cr=^M:do=^J:nl=^J:bl=^G:.....
%
```

ENVIRONMENT VARIABLES (cont.)

- To set an environment variable for a terminal session, use the format:

  `setenv` *ENV_VARIABLE_NAME value*

  ```
  % setenv PRINTER swip
  ```

- To permanently set an environment variable, place the instruction in your *.login* file

  ```
  setenv EDITOR /usr/convex/edt
  ```

- To remove the definition of an environment variable for a terminal session, use the format:

  `unsetenv` *ENV_VARIABLE_NAME*

  ```
  % unsetenv PRINTER
  ```

Setting Environment Variables

Set your editor for this terminal session to emacs; you will need to use the pathname: */usr/convex/emacs* to specify the editor. Your entry:

```
% setenv EDITOR /usr/convex/emacs
%
```

Verify that the editor is emacs by displaying the environment variables.

```
% printenv
HOME=/mnt/username
SHELL=/bin/csh
PATH=.:/mnt/username/bin:/usr/convex:/usr/ucb:/bin:/usr/bin
EDITOR=/usr/convex/emacs
TERM=vt100n
USER=username
TERMCAP=d0|vt100n:cr=^M:do=^J:nl=^J:bl=^G: .....
%
```

TERMINAL SETUP

The *tset* command:

- Is usually placed in your *.login* file
- Sets up your terminal when you first log in
- Does terminal dependent processing:
  - Setting erase and kill characters
  - Setting or resetting delays
  - Sending any sequences needed to properly initialize the terminal
- An example of a common terminal setup:

```
set noglob
eval "tset -Q -s -m dialup:vt52"
```

This form:

- Initializes the terminal
- Places the termcap entry in the environment - the environment variable *TERMCAP*
- Sets up for a vt52 style terminal if the login terminal type is "dialup"

Using the *tset*(1) Command

Determine if any options have been set for your terminal with the *tset* command by viewing your *.login* file. Your entry:

```
% more .login
set ignoreeof
set mail = /usr/spool/mail/$user
set path = ( . ~/bin /usr/convex /usr/ucb /bin /usr/bin)
umask 022
stty crt erase ^H kill ^U
set noglob
eval 'tset -Q -s'
%
```

How can you determine what the command *tset -Q -s* means?

TERMINAL SETUP (cont.)

The *stty* command:

- Allows you to set terminal options:
  - Parity
  - Echo
  - Key characteristics
- To display a listing of the baud rate and option settings that are currently in effect use:

```
% stty
```

- To display all the options currently in effect, use:

```
% stty all
```

- To display everything *stty* knows, use:

```
% stty everything
```

Using the *stty*(1) Command

Display the baud rate and terminal options that are different than the defaults. Your entry:

```
% stty
new tty, speed 9600 baud; tabs ff1 crt
erase = ^H
%
```

Display the options that show which control characters can be used for corrections at the command line. Your entry:

```
% stty all
new tty, speed 9600 baud; tabs ff1
crt
erase kill werase rprnt flush lnext susp  intr quit stop  eof
 ^H    ^U    ^W     ^R     ^O   ^V   ^Z/^Y  ^c   ^\  ^S/^Q  ^D
%
```

TERMINAL SETUP (cont.)

- To change a terminal setting for a work session, use the format:

  stty *function new_assignment*

  % stty erase ^e

- To permanently setup a terminal option, place the instruction in your *.login* file:

  stty erase ^e

Setting Terminal Options

Change the erase line (kill) character to CTRL-k. Your entry:

```
% stty kill ^k
%
```

Type some characters and erase them with the new setting. Now change erase back to CTRL-u. Your entry:

```
% stty kill ^u
%
```

Exercises for Chapter 1

1. What is the general format for UNIX commands?

2. When multiple options are desired with a command, how are they entered?

3. What is the control sequence to continue a command onto the next line?

4. What will the following command cause: *lf- al*?

5. What is the command to erase the last word of the command you are entering.

6. You want to execute the following commands without entering each command on a separate command line - print the date and sort the contents of the file named *addresses*. Write the command(s) that will allow you to complete this task.

7. What is indicated whenever a pathname begins with /?

8. What is the difference between absolute and relative pathnames?

9. Enter the command *printenv*. What is the purpose of the *path* that is displayed?

10. What does the command *!!* do?

11. What does the command *!cd* do?

12. In the history file is the following commands: *220 cd /docs/smiht/class/courses*. Write a command that would change *smiht* to *smith*.

13. Which environmental variable sets your terminal characteristics?

14. How many options are there for the *lf* command?

15. What is the command to display the on-line documentation for the *grep* command?

16. What does the command *lf -l a R* display?

17. A user wants to be notified with a beep when mail arrives. What command must the user set?

18. A user does not want to be interrupted with *talk* messages from other users? What command must the user set?

19. Write a command that reminds you of a meeting today at 3:45 p.m.; you want to be reminded about this meeting a few minutes before it is time to leave. What command can you use?

20. What is the purpose of the *passwd* command?

Review Exercises

1. Write a command that displays the protection scheme of the files in your home directory.

2. What are the permissions on your home directory?

3. What determines the default permissions on any files that you create?

4. What is the maximum number of characters for a user name?

5. What command will always return you to your home directory?

6. What does the symbol "*" indicate after the displayed file names when you do an *lf* command?

7. Write a command that copies a file named *comments* from the home directory of user *smith* to your home directory.

8. List all the files and directories in your account. What command did you use?

9. Make a directory called programs. What command did you use?

10. Move the file *sub1.f* in your home directory to the subdirectory *programs*. What command did you use?

11. List the hidden files in your home directory. What command did you use? What are the files . and ..

12. Write the command that will display the total number of files in the */etc* directory.

13. You have submitted the file *main.f* to the printer. Write the command to remove this file from the print queue.

14. What is standard input?

15. What is standard output?

16. What is standard error?

17. Write a command that places the contents of all the files of your current directory ending with *.c* into a file called *all.prog*.

18. How many people are logged on to your local system? What command did you use to get this information?

19. You need to redirect the error messages from the command *cat fun new old > holder* to a file. Write a command that would accomplish this task.

20. Write a command that will display the total number of files in your home directory no matter where you are in the directory structure.

21. Write a command that adds today's date to the file *output* in the *programs* directory.

22. Do you have any jobs running in the background? How did you determine this information?

23. You need to execute a job in the background. How can you accomplish this task?

24. How do you suspend a job that is currently executing?

25. How do you restart a suspended job and bring it to the foreground.

26. List the command to kill job number 3.

2

Search Utilities

Objectives for Chapter 2

After completing Chapter 2, you will be able to:

1. Use the *whatis* command.
2. Find files by name.
3. Find files by last access.
4. Execute commands on files that meet specific specifications.
5. Print out lines that fit specified descriptions from files, using *grep*, *egrep*, or *fgrep*.
6. Use appropriate options when doing searches for patterns, words, lines, strings, etc., using *grep*, *egrep*, or *fgrep*.
7. Use *grep* in multiple word searches of one or more files.
8. Search through several files for a specified reference using a search string as the criterion.
9. Search files for a specified reference using patterns that contain characters recognized by UNIX as being *special* characters, i.e., *, !, ?, etc.

LOCATING COMMANDS

The *which* command:

- Locates a program file, aliases, and paths (*csh* only)
- Looks for the file which would have been executed had that name been given as a command
- Searches along the path from the user's current path
- Displays the pathname of a given command
- Has the form:

  `which` *command*

```
% which emacs
/usr/convex/emacs
```

Using the *which*(1) Command

Locate the program file *ex*. Your entry:

```
% which ex
/usr/ucb/ex
%
```

LOCATING FILES WITH *find*

- Searches for files that meet conditions you specify:
  - Filename matches specified pattern
  - Creation date in given range
  - Date of last use
  - Specific owner or permission
  - Specific file characteristic
  - Any combination of above characteristics
- Begins searching at directory specified and searches recursively
- Has the format:

  `find` *dir_pathname criterion action*

  % `find /mnt/user -name states1 -print`

Locating Files

Determine if you have a file named *states1* in your directory; start the search at your home directory. Your entry:

```
% find /mnt/username -name states1 -print
/mnt/username/states1
/mnt/username/bin/states1
%
```

LOCATING FILES WITH *find* (cont.)

Commonly used search criteria:

- By name: `-name` *desired files*

-name file1	Looks for files named *file1*
-name 'file?'	Looks for files named *file* followed by one character
-name 'file*'	Looks for any files beginning with *file*
-name '*.?'	Looks for all files with the next to last character a period

```
% find /mnt/person -name file1 -print

% find /mnt/person -name 'file?' -print

% find /mnt/person -name 'file*' -print

% find /mnt/person -name '*.?' -print
```

Finding Files Using Metacharacters

To do this exercise, you will need five files in your home directory; create the empty files:

touch file1 file2 filea file.1 file.2

Determine the pathname(s) for any *file* followed by one character that are located in any of your directories.

```
% find ~ -name 'file?' -print
/mnt/username/file1
/mnt/username/file2
/mnt/username/filea
%
```

If you have a directory *project*, add an empty file name *file.test* to this directory. To create this file type:

touch project/file.test

Determine all the files in your directories (beginning at the home directory) that begin with *file*. Your entry:

```
% find ~ -name 'file*' -print
/mnt/username/project/file.test
/mnt/username/file1
/mnt/username/file2
/mnt/username/filea
/mnt/username/file.1
/mnt/username/file.2
%
```

LOCATING FILES WITH *find* (cont.)

- By last access: `-atime` [-/+] *number_of_days*

-atime 5	Files last accessed exactly 5 days ago
-atime -5	Files accessed more recently (less) than 5 days ago
-atime +5	Files accessed more than 5 days ago

```
% find /usr/person -atime 5 -print

% find /usr/person -atime -5 -print

% find /usr/person -atime +5 -print
```

- By last modification: `-mtime` [-/+] *number of days*

-mtime 10	Files last modified 10 days ago
-mtime -10	Files last modified less than 10 days ago
-mtime +10	Files last modified more than 10 days ago

```
% find /usr/person -mtime 10 -print

% find /usr/person -mtime -10 -print

% find /usr/person -mtime +10 -print
```

Locating Files By Last Modification

Determine which files (if any) in your directories have been modified less than two days ago. Your entry:

```
% find ~ -mtime -2 -print
pathname
pathname
. . .
. . .
%
```

LOCATING FILES WITH *find* (cont.)

- For files modified more recently than a given file, use: `-newer` *filename*

  % `find /usr/person -newer file1 -print`

- By username or ID: `-user` *username* or *ID*

  % `find /usr -user snoopy -print`

- To *exec* a command, the action format:

  `-exec` *command* `{} \;`

  For example, the command:

  % `find /usr/person -atime +45 -exec rm {} \;`

  removes all the files in the named directory that haven't been used in over 45 days; the brackets ({}) represent the name of files to be removed

- For information on other options, see *find*(1) in the *CONVEX UNIX Programmer's Manual*

Running a Command on Found Files

Use the *find* command to find all the files beginning with *st* in your directories and list them. Your entry:

```
% find ~ -name 'st*' -exec ls -l {} \;
%
```

Determine if user *bayne* has any files in the directory */tac/bin*. Your entry:

```
% find /tac/bin -user bayne -print
/tac/bin/bayne/states1
/tac/bin/bayne/states2
. . . .
. . .
%
```

PATTERN SEARCHING IN A FILE

grep(1)

- Actually consists of a group of three utilities -- *grep, egrep, fgrep*
  - *grep* matches regular expressions in the style of *ex*
  - *egrep* matches extended regular expressions
  - *fgrep* matches fixed strings
- Searches named files for lines containing given pattern - strings or expressions
- Prints out the matching line(s)
- Can be used with pipes
- Has the form:

  `grep` [*option ...*] *pattern/expression filename ...*

  `egrep` [*option ...*] *pattern/expression filename ...*

  `fgrep` [*option ...*] *string filename ...*

```
% grep Phony /etc/passwd
% egrep 'shaef+er' phone /etc/passwd
% fgrep 'digits wide' /mnt/username/*
```

Searches With the *grep(1)* Family

To complete the exercises, you will need the following file. Create the file *poem* with your favorite UNIX text editor; type the lines exactly as shown.

In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
My Xanadu checkline.
Down to a sunless sea.*

*From Kubla Khan by Samuel Taylor Coleridge.

grep, *egrep*, and *fgrep* Options

-c	Prints only the total number of lines that match
-f *file*	The regular expression (*egrep*) or string list (*fgrep*) is taken from the *file*
-i	Ignores case in matches (*grep* and *fgrep* only)
-n	Precedes found line(s) with its line number in the file
-v	Prints all lines which do not match
-w	Searches for pattern as an isolated word (*grep* only)
-x	Prints only lines which match in their entirety (*fgrep* only)

```
% grep -c Maine /mnt/username/*
% grep -i Phony/etc/passwd
% grep -n animal vet cattle barn
% grep -w man infofile
% fgrep -x 'this old man' poem
% egrep -v '^Mississippi' book
```

Using *grep*(1) Options

Search for your username in the */etc/passwd* file. Your entry:

```
% grep username /etc/passwd
username:/caK581Grpf/E:245:51:username,,,:/tac/username:/bin/csh
%
```

Use *egrep* to print all lines that do not match the string *Xanadu* in the file *poem*. Your entry:

```
% egrep -v 'Xanadu' poem
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
%
```

Print only the line *A stately pleasure dome decree:* along with its line number from the file *poem*. Your entry:

```
% fgrep -n -x 'A stately pleasure dome decree:' poem
2:A stately pleasure dome decree:
%
```

SINGLE- VS MULTIPLE-WORD SEARCHES

Using single- and multiple-word strings:

- A single-word string matches:
  - String by itself
  - String embedded in a larger string (when *-w*) option is not used
- Multiple-word searches:
  - Must use single or double quotes around string or *grep* becomes confused:

```
grep 'Colorado Springs' cities
```

Using *grep* in Multiple-Word Searches

Print out the line(s) in the file *poem* that contain *In Xanadu.* Your entry:

```
% grep 'In Xanadu' poem
In Xanadu did Kubla Khan
%
```

SEARCH PATTERN ELEMENTS

`grep 'thing' file1`	Matches *thing, something,* or *everythingelse*
.	Matches any single character
`grep 's.nd' file1`	Matches *sand* or *send*
[...]*chars*	Matches single character or range in the bracketed list
`grep '[l-s]end' file1`	Matches *lend, mend* or *send*
`grep '[Mm]end' file1`	Matches *Mend* or *mend*
[^...]*chars*	Matches any character not in the bracketed list
`grep '[^l-s]end' file1`	Matches *bend* or *tend*
`grep '[^abc]' file1`	Matches strings not containing an individual character or combination of characters within the brackets
^	Matches any line beginning with expression following ^
`grep '^by' file1`	Matches any line beginning with *by*
$	Matches any line ending with the expression preceding the $
`grep 'test$' file1`	Matches any line ending with *test*
*	Matches zero or more occurrences of the preceding character
`grep 'Pr*' file1`	Matches any line containing *P* followed by zero or more *r*'s
\	Escapes special meaning of next character
`grep '\!' file1`	Prints any line containing an exclamation mark

Note: The single quotes can be omitted on examples 1, 2, and 8.

Search Pattern Elements Recognized by *grep*

Print only those lines of the file *poem* that end with *an*. Your entry:

```
% grep 'an$' poem
In Xanadu did Kubla Khan
Where Alph, the sacred river, ran
Through caverns measureless to man
%
```

Print all the lines in the file *poem* that contain the word *man* or *ran*. Your entry:

```
% grep '[mr]an' poem
Where Alph, the sacred river, ran
Through caverns measureless to man
%
```

Print only the line(s) that contain a period in the file *poem*. Your entry:

```
% grep \.' poem
My Xanadu checkline.
Down to a sunless sea.
%
```

Exercises for Chapter 2

1. Locate the program file *vi*. What is the pathname? What command did you use to find this information?

2. Assume you have several directories and somewhere you have a file named *secrets*. Write a command that would find (and print) the pathname for that file.

3. Write a command that displays the pathname for any files in your directories that begin with *sa*.

4. Write a command that displays the filenames that are in your directory that have not been accessed for more than 45 days.

5. Write a command that displays the filenames that user *snoopy* has in the *mnt* directory.

6. Write a command that displays a listing of only your directory names.

7. Write a command that finds all the files in */mnt/snoopy* that were last modified more than 45 days ago and sends that output to a file named *old.45*.

8. What does the command **find ~ -name 'test?' -ok rm** {} \; accomplish? Hint: You may need to read the man page for *find*.

9. User snoopy has not been cleaning up his directories. Write a command that removes all files in the */mnt/snoopy* directory that have not been accessed for more than 45.

10. Write a command that locates all the files in the directory of *snoopy* that begin with *super* and displays the filenames on the terminal.

11. What is the purpose of the *which* command?

12. What would happen if you entered the command **find ~ -name 'st*'**? What filenames would be displayed? Why?

13. What is the difference between the commands *whatis* and *which*?

14. What are the differences and similarities between *grep*, *fgrep*, and *egrep*. Which one is the fastest? Which one is the most flexible?

15. What are the two different ways the character ^ (caret) is used in defining patterns in *grep*?

16. How can you designate a * as an ASCII character without its meta meaning being used in *grep*?

17. What are the differences between the following three commands?

```
grep 'a.*a' awkdata
grep 'a.a' awkdata
grep 'a*a' awkdata
```

18. Devise grep patterns to match the following:

    a. Smith or smith
    b. 3-character string starting with *f* and ending with *t*
    c. all lines beginning with *BEGIN*
    d. all lines in which the first nonspace character is a *B*

    Verify that the patterns are correct by executing them on a file containing the appropriate entries.

19. Write a command that determines if *lonely* appears in the files *lonesome, nonsense,* and *poetry*.

20. Write a command that lists all the programs in your current directory that contain the line *header.h*. (Hint: Header files are usually found in files that end with *.c*).

21. Write a command using *grep* which will list all the files in your directory which do not end in *.s*.

22. Write a command that prints only the lines containing a question mark (?) in the file *clown*.

23. Assume that one of your files contains the line "Through caverns measureless to man", but you do not remember the filename. Write a command that searches for that line, prints the filename containing that line, as well as the line.

24. Write a command that determines if a password exists for user *Snoopy* or *snoopy*. (Passwords are found in the */etc/passwd* file.)

25. Write a command that prints the current input line number of the line beginning with *My* in the file *poem*. Verify that the command works correctly.

26. Write a command to determine whether *smith* is logged on the system (without reading the entire output of *who*).

27. Write a command to determine if *lpd* is active.

3

Creating C Shell Scripts

Objectives for Chapter 3

After completing Chapter 3, you will be able to:

1. Identify characters with special meaning to the C shell.
2. Assign values to variables.
3. Identify variables initialized by the C shell.
4. Create C shell scripts requiring no flow control.
5. Create C shell scripts using basic control structures: if and foreach.
6. Execute a C shell script.

SHELLS

- Shells are programs that read commands and execute them
- Logging in creates your initial *shell*
- Entering a command forks a process which may be a new *shell*
- Entering *csh* at the *shell* prompt creates a new *shell*
- Executing a C Shell script creates a new *shell*

%	Login *shell*
% `csh`	Requests another *shell* (2nd-level)
% `my.script`	Forks a *shell* (3rd-level)
%	Script completes (returns to 2nd-level)
% `^D`	Terminates 2nd-level *shell*
%	Back to original login *shell*

The C Shell

Put the line:

echo "I am a new shell"

in your *.cshrc* file. When you fork a new *shell*, the line *I am a new shell* displays. Verify that this is true by forking a new *shell* with **csh**. Terminate the *shell* with CTRL-d. Your entry (after editing your *.cshrc* file):

```
% csh
I am a new shell
%%
```

When you type CTRL-d, it does not display, but a second *shell* prompt is displayed. Remove the line from the *.cshrc* file so that it is not displayed each time you execute a script.

C SHELL SCRIPTS

A C shell script:

- Is created with a text editor
- Contains a # in column 1 of line 1 which indicates that the C shell should be used to execute the commands; if # is not included, the Bourne Shell is assumed

  Use the line `#!/bin/csh -f` (in place of #) for fast startup; *.cshrc* file is not read
- Performs a set of UNIX commands contained in a file
- Must be executable to perform the commands
  - Change the protection mode to be executable:

    % `chmod 755` *scriptname*

    or

    % `chmod a+x` *scriptname*
- Can be executed by typing the filename

  % *scriptname* [*arguments*]
  % `sample.script`

Creating C Shell Scripts

You will be working with a variety of scripts throughout this module. Create the following *data.script* file with your favorite text editor; change the mode to executable.

```
#!/bin/csh -f
echo "This script prints information related to your login shell"
echo "The files in your current directory are:"
ls
echo "The environmental variables that are set are:"
printenv
```

Your entry to make the file executable:

```
% chmod 755 data.script
%
```

Now execute the script. Your entry looks similar to:

```
% data.script
This script prints information related to your login shell
The files in your current directory are:
example.1     example.3     sample.2      test1         test3
example.2     sample.1      test          test2
The environmental variables that are set are:
HOME=/trn/snoopy
SHELL=/bin/csh
PATH=.:/trn/snoopy/bin:/usr/convex:/usr/ucb:/bin:/usr/bin:/usr/local/bin:
TERM=vt100n
USER=snoopy
PAGER=less
EDITOR=/usr/convex/emacs
PRINTER=trip
%
```

C SHELL VARIABLES

- A variable name begins with a letter and consists of letters, digits, and underscores

- To define the contents (value) of a variable use the *set* command followed by an equal sign and the value

```
set vname=value
set sample=1
```

- To show the variables that are set, type: `set` at the *shell* prompt

- Refer to variable values by preceding their names with a dollar sign ($)

```
% set sample=1
% echo $sample
1
```

Using the previous information in a script:

```
#!/bin/csh -f
set sample=1
echo $sample
```

Execute the script:

```
% sample.script
1
```

C Shell Variables

Set the following variables:

```
set sample = 1
set sample2 = hi
set sample3 = date
```

Display a listing of all the variables that are set. Finally display individual variable values using the *echo* command.

Your entry looks similar to:

```
% set sample = 1
% set sample2 = hi
% set sample3 = date
% set
argv      ()
autologout 0
cdpath    /mnt/username
history   20
home      /mnt/username
. . .
sample    1
sample2   hi
sample3   date
% echo $sample $sample2 $sample3
1 hi date
%
```

SETTING STRING VARIABLES

- To assign the contents literally, use single forward quotes:

```
% set a = 'Hi there'
% echo $a
Hi there
```

- Strings not contained in quotes may expand incorrectly:

```
% set b = hi there
% echo $b
hi
```

This actually sets b = hi and there = (null)

- Single or double quotes cause the contents to be considered as one element

- To determine the number of elements, use $#*variable_name*

```
% echo $#a
1
```

- Using double quotes causes all variables to be expanded except filenames

Setting String Variables

Write a script named *var.script* that sets the following variables:

sample4 = Hi there
sample5 = "Hi there"
sample6 = 'Hi there'

and displays each of their values and number of elements when the script is executed. Your script contents looks similar to:

```
#!/bin/csh -f
set sample4 = Hi there
set sample5 = "Hi there"
set sample6 = 'Hi there'
echo $sample4
echo $sample5
echo $sample6
echo $#sample4
echo $#sample5
echo $#sample6
```

Your executed results look similar to (remember to change the mode to executable):

```
% chmod 755 var.script
% var.script
Hi
Hi there
Hi there
1
1
1
```

Notice in the output that your value for *sample4* is incorrect; only the first item is read and displayed. Thus, you can see that quotation marks are necessary for string values.

SETTING ARRAY VARIABLES

- To assign more than one element, use parentheses around the elements; a space separates the elements:

```
% set b = (This is four elements)
                        or
% set b = ('This' 'is' 'four' 'elements')
% echo $#b
4

% set c = "This is one element"
% echo $#c
1
```

- To specify array elements, use the form:

$*variable_name*[*element_number-element_number*]

The following example illustrates using the value of the first three elements (range) of *b*:

```
% echo $b[1-3]
This is four
```

To use the value of specific elements of the variable *b*:

```
% echo $b[1] $b[4]
This elements
```

Setting Array Variables

Write a script named *test.script* that set variable *elements* to the following:

(How many elements does this contain?)

Your script looks similar to:

```
#!/bin/csh -f
set elements = (How many elements does this contain\?)
echo $#elements
```

Execute the script:

```
% chmod 755 test.script
% test.script
6
%
```

If you can't get your script to execute properly, put a back slash (\) before the question mark.

Note that you can not echo $elements. You must first set noglob.

VARIABLE EXPANSION AND COMMAND SUBSTITU WITH QUOTATION MARKS

- Backward single quote (`):

```
%set i = ls
%echo $i
ls
% echo `$i`
file1 file2 file3
% set i = `ls`
% echo $i
file1 file2 file3
```

- Forward single quote ('):

  Prevents variable expansion, wildcard expansion, or alias expansion

```
% set a = ls
% echo '$a'
$a
% set i = sample
% echo 'Here is $i'
Here is $i
```

Variable Expansion with Single Quotes

Experiment with using single quotes with a command. Enter:

set d = date

Now echo the value of variable *d* using no quotes, using single backward quotes, and single forward quotes. Note the differences in *d*'s value.

Your entry:

```
% set d = date
% echo $d
date
% echo `$d`
Wed Dec 17 13:19:18 CST 1986
% echo '$d'
$d
%
```

VARIABLE EXPANSION AND COMMAND SUBSTITU WITH QUOTATION MARKS (cont.)

- Double quotes (”):

  Groups characters into a single argument

```
% set i = "Hi there"
% echo $i
Hi there
```

  Permits variable expansion to take place

```
% set i = "my sample"
% echo "Here's $i"
Here's my sample
```

  No filename wildcard expansion takes place

```
% set i = "ls file*"
% echo "$i"
ls file*
```

  Filename wildcard expansion does take place if you do not use the double quotes:

```
% echo $i
ls file1  file2  file3
```

Variable Expansion with Double Quotes

Move to your home directory if you are not there. Write a *quote.script* that prints: **Here is my sample of files ending with script**. Assign **ls *script** to the variable *list*; assign **my sample** to the variable *s*.

Your script looks similar to:

```
#!/bin/csh -f
set s = "my sample"
set list = `ls *script`
echo "Here is $s of files ending with script"
echo $list
```

Verify that your script performs correctly. Your entry:

```
% quote.script
Here is my sample of files ending with script
data.script pre.script test.script var.script ...
%
```

FILENAME EXPANSION

file*	Matches *file* followed by zero or more characters `file*` matches *file*, *file1*, and *file.out*
file?	Matches *file* followed by any one character `file?` matches *file1*, *fileA*, and *filed*
file[123]	Matches any single character contained in the brackets `file[123]` matches only *file1*, *file2*, and *file3*.
file[a-z]	Matches *file* followed by any lowercase letter `file[a-z]` matches *filea*, *filef*, and *filez*

Filename Expansion

Change your directory to *samples*. Write a script named *list.script*:

1. that displays a message that says the files being listed are found in your current directory.
2. that lists all the files ending with *script* in your current directory.

Your script looks similar to:

```
#!/bin/csh -f
echo "The files in your current directory that end with script are:"
ls *script
```

Execute the script:

```
% list.script
The files in your current directory that end with script are:
data.script  test.script
pre.script   var.script
%
```

PREDEFINED VARIABLES

The C shell has the following predefined variables:

$ user	Current user `echo $user`
$ home	Home directory `echo $home`
$ shell	Shell you are currently running `echo $shell`
$ path	Your search path `echo $path`
$ term	Your terminal type `if ($term==adm3a) then`
$ status	Exit status of the last command `if ($status==1) then`
$ cwd	Current working directory `echo $cwd`

Predefined Variables

Write and execute a script name *pre.script* that tells you your home directory, your current working directory, and the *shell* you are running under. Your script looks similar to:

```
#!/bin/csh -f
echo $home
echo $cwd
echo $shell
```

Execute the script:

```
% chmod 755 pre.script
% pre.script
/mnt/username
/mnt/username/your_directory_name
/bin/csh
%
```

Make a directory called *samples* and move to that directory. Write a script named *dir.script* that displays the name of your current working directory and your home directory, changes to your home directory and lists all the files in your home directory.

Your script looks similar to:

```
#!/bin/csh -f
echo $cwd
echo $home
cd $home
ls
```

Verify that your script performs correctly:

```
% chmod 755 dir.script
% dir.script
/mnt/username/samples
/mnt/user/
data data.script pre.script test.script var.script
%
```

Change to your home directory.

SPECIAL VARIABLE FORMS

The C shell recognizes the following variables:

$?name — Replaced by 0 if name is not set
Replaced by 1 if name is set

```
set i = (a b c)
echo $?i
1
```

$#name — Replaced by number of elements in variable *name*

```
set i = (a b c)
echo $#i
3
```

$name[1-3] — Replaced by first 3 elements of array *name*

```
set i = (a b c)
echo $i[1-3]
a b c
```

$name[2-] — Replaced by 2nd through last element of *name*

```
set i = (a b c)
echo $i[2-]
b c
```

$name[*] — Replaced by all elements of *name*

```
set i = (a b c)
echo $i[*]
a b c
```

Special Variable Forms

At your terminal set the following variables:

a = my.sample
b = (c d e f g h)

Use the *echo* command to display the following:

1. Number of elements in each variable
2. First 3 elements of variable *b*
3. Last 2 elements of variable *b*
4. All the elements for each variable
5. Determine if *j* is set

Your entry looks similar to:

```
% echo $# a $# b
1 6
% echo $b[1-3]
c d e
% echo $b[5-]
g h
% echo $a[*] $b[*]
my.sample c d e f g h
% echo $?j
0
%
```

SPECIAL VARIABLE FORMS (cont.)

The C shell recognizes the following variables when used in a script:

$0	Replaced by name of script being executed
$1	Replaced by first argument to script
$2	Replaced by second argument to script
$*	Replaced by all arguments to script
$$	Replaced by process number of current shell
$<	Reads 1 line of input from terminal

If *my.script* contains:

```
#!/bin/csh -f
echo $0                          # gives names of script
echo $1 $2                       # gives first and second elements
echo $*                          # gives all elements
echo $$                          # gives the process ID
mv $1 $1.old                     # renames first argument
echo "Moved $1 to $1.old"        # displays first element renamed
echo 'Who are you'               # displays contents at terminal
set a = ($<)                     # reads line of terminal input
echo 'I am done' $a              # displays contents at terminal
```

Executing the script produces:

```
% my.script file1 file2 file3
my.script
file1 file2
file1 file2 file3
13369
Moved file1 to file1.old
Who are you
user1
I am done user1
```

Script Variables

Write a script named *back.script*:

1. that displays a message telling the user what the script does; include the name of the script in the message.
2. that copies a file entered as the first script argument to *file*.**back**.
3. displays a message that tells the user the task is completed.

Your script looks similar to:

```
#!/bin/csh -f
echo "$0 will copy $1 to $1.back"
cp $1 $1.back
echo "$1 is copied to $1.back"
```

Execute the script (did you make it executable?); your entry:

```
% back.script file1
back.script will copy file1 to file1.back
file1 is copied to file1.back
%
```

THE *argv* VARIABLE

- The *argv* variable receives the arguments sent to the script:

    script_name argv[1] argv[2] argv[3] ...

  argv receives its input from the variables that are included after the script name:

    % my.script a b c

  If *my.script* contains:

```
#!/bin/csh -f
echo $argv[1]      # first element ($1)
echo $argv[2]      # second element ($2)
echo $argv[3]      # third element ($3)
echo $argv         # all elements ($*)
echo $#argv        # number of elements
echo 'I am done'
```

  Executing the script produces:

```
% my.script a b c
a
b
c
a b c
3
I am done
```

The *argv* Variable

Modify your *back.script* to use the *argv* variable. Your modified script looks similar to:

```
#!/bin/csh -f
echo "$0 will copy $argv[1] to $argv[1].back"
cp $argv[1] $argv[1].back
echo "$argv[1] is copied to $argv[1].back"
```

Execute the script (did you make it executable?); your entry:

```
% back.script file2
back.script will copy file2 to file2.back
file2 is copied to file2.back
%
```

MATHEMATICAL OPERATIONS

Using the @command:

- Similar to the *set* command
- Used with mathematical calculations (+, -, /, *, etc.)

  Examples:

```
% set i = 10
% set j = 2
% @ k = ($i + $j)
% echo $k
12
% @ z = ($i / $j)
% echo $z
5
```

- Use for incrementing:

```
% set z = 1
% @ z = ($z + 1)
% echo $z
2
```

Mathematical Operations

Write *add.script* that does the following:

1. displays the message: "This scripts adds 2 integers".
2. asks the user for an integer.
3. asks user for a second integer.
4. adds the two integers.
5. displays the message: "Total is:" followed by the total.

Your script looks similar to:

```
#!/bin/csh -f
echo "This script adds 2 integers:"
echo "Type your first integer and press RETURN"
set a = ($<)
echo "Type your second integer and press RETURN"
set b = ($<)
@ total = $a + $b
echo "The total is: $total"
```

Verify that the script executes correctly. Your entry:

```
% add.script
I add two numbers
Type your first number and press RETURN
345
Type your second number and press RETURN
23
The total is: 368
%
```

FLOW CONTROL - FOREACH STATEMENTS

- foreach statement format:

```
foreach i (*.f)        foreach i (*.f)
  ...                    ...
  break                  continue
end                    end
```

Example script:

```
#!/bin/csh -f
#####################################################
# this script does the following:
#  compiles every FORTRAN file in the
#    directory which ends in .f
#  the object files *.o are built
#    but due to the -c option,
#    no executable file is built
#####################################################
foreach i (*.f)
  fc -O2 -c $i
end
```

foreach Statements

Enhance your *back.script* to handle more than one file at a time. Include the messages:

back.script is copying *filename* to *filename*.back
***filename* is copied to *filename*.back**

Your script looks similar to:

```
#!/bin/csh -f
foreach i ($*)
echo "$0 is copying $i to $i.back"
cp $i $i.back
echo "$i is copied to $i.back"
end
```

Verify that your script executes correctly. Your entry looks similar to:

```
% back.script file1 file2 file3
back.script is copying file1 to file1.back
file1 is copied to file1.back
back.script is copying file2 to file2.back
file2 is copied to file2.back
back.script is copying file3 to file3.back
file3 is copied to file3.back
%
```

FLOW CONTROL - IF STATEMENTS

Several commands exist that can be used to regulate flow control:

- if statement format:

```
if ($i == test) then
  . . .
else
  . . .
endif
```

Nested if statements are permitted

Example:

```
#!/bin/csh -f
set a = 5
set b = 6
if ( $a > $b) then
    echo $a is greater than $b
else
    echo $a is not greater than $b
endif
```

Execute the script:

```
% sample.script
5 is not greater than 6
```

if Statements

Write a *cmp.script* to perform the following:

1. Asks the user for two files to compare.
2. If two filenames are not entered, display the message: "You didn't give me two filenames! I am quitting."
3. If two filenames are entered, then display the message: "I will make a comparison of *file1 file2.*
4. Compare the files; use the option that returns only the status with the *cmp* command.
5. If the status equals zero, then display the message: "Identical files; I will remove *file1.*" Then remove *file1.*
6. If the status returns non-zero, then display the message: "Not identical; I won't remove either file."

Your script looks similar to:

```
#!/bin/csh -f
#
echo "Enter the name of two files to compare"
set ans = $<
#The following set will break the string into words
set num = ($ans)
if ($#num == 2) then
        echo "I will make a comparison of"
        echo $num[1]  $num[2]
        cmp -s $num[1] $num[2]
          if ($status == 0) then
            echo "identical files; I will remove $num[1]"
            rm $num[1]
          else
            echo "Not identical; I won't remove either file"
          endif
else
echo "You didn't give me two filenames"
echo "I am quitting."
endif
```

Verify that your script performs correctly.
Your entry:

```
% cmp.script
Enter the name of two files to compare"
test1 test.s
I will make a comparison of
test1 test.s
Not identical; I won't remove either file
% cmp.script
Enter the name of two files to compare
test
You didn't give me two file names!
I am quitting.
% cmp.script
Enter the name of two files to compare
sunny ugly
I will make a comparison of
sunny ugly
identical files; I will remove sunny
```

COMPARISON OPERATORS

=	Assignment operator `set a = 1`
==	Equal to `if ($i == 10) then`
!=	Not equal to `if ($1 != 0) then`
&&	Boolean AND
><	Greater than; less than `if ($i > 1 && $i < 10) then`
||	Boolean OR `if ($i == 0 || $i == 1) then`
!	Boolean NOT `if ( !($i <= 10) ) then`
=~	Similar To `if ($i =~ test?) then`
!~	Not Similar To `if ($i !~ test?) then`

EXAMPLE:

```
% set i = (a b c)
% if ($#i == 3) echo There are three
There are three
% if ($#i == 2) echo There are two
%
```

(Nothing is displayed as the expression is false)

Comparison Operators

Write a *assign.script* that

1. assigns 6 to *a*
2. assigns 10 to *b*
3. assigns 15 to *c*
4. uses Boolean AND to compare the value of *c* to both *a* and *b*
5. prints a message "C is greater than A or B" if it is

Your script looks similar to:

```
#!/bin/csh -f
set a = 6
set b = 10
set c = 15
if ($c > $a && $c > $b) echo C is greater than A or B
```

Execute your script:

```
% assign.script
C is greater than A or B
%
```

FLOW CONTROL - WHILE STATEMENTS

- while statement format:

```
while ($i != 13)        while ($i != 13)
  ...                     ...
break                   continue
end                     end
```

Example script:

```
#!/bin/csh -f
echo "Guess what number I am thinking of"
echo "Enter the number and press return"
set ans = $<
while ($ans != 5)
   echo "Enter another number"
   set ans = $<
end
echo "You guessed it"
```

while Statements

Write *cal.script* that prints a calendar for a year. If the user does not specify a year, ask for the year. Also, display a message that tells what year calendar is being printed.

Your script looks similar to:

```
#!/bin/csh -f
if ($#argv != 1) then
        echo "enter the desired year:"
        set j = ($<)
else
        set j = $1
endif
echo "The desired year is $j"
set i = 1
while ($i != 13)
        cal $i $j
        @ i = ($i + 1)
end
echo "Calendar is complete"
```

Execute your script:

```
% cal.script
Enter the desired year:
1987
The desired year is 1987
   January 1987
 S  M Tu  W Th  F  S
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
   (continues)
Calendar is complete
%
```

FLOW CONTROL - CASE STATEMENTS

- switch statement format:

```
switch ($variable)
case string:
  . . .
  breaksw
  . . .
default:
  . . .
endsw
```

Example script for selecting a printer:

```
#!/bin/csh -f
echo "This does an iprx on the file"
echo "Enter the file name"
set fn = ($<)
echo "Select the number of the printer you want"
echo "1 for swip, 2 for vaxip, or 3 for tacip"
set p = ($<)
switch ($p)
case 1:
  iprx -Pswip $fn
  breaksw
case 2:
  iprx -Pip $fn
  breaksw
case 3:
  iprx -Ptacip $fn
  breaksw
default:
echo "Nothing printed; invalid printer number"
endsw
```

case Statements

Write *case.script* that:

1. displays the message: "This script allows you to copy or move a file".
2. displays the message: "Enter the item number and press RETURN".
3. Asks the user to select 1 to copy a file or select 2 to move the file.
4. Displays the message(s): asking for the filename to copy or move.
5. Displays a message telling the user that file has been copied or moved.
6. Displays a message if the user enters any number other than 1 or 2.

Your script looks similar to:

```
#!/bin/csh -f
echo "This script allows you to copy or move a file"
echo "Enter the item number and press RETURN"
echo "1 for copy a file or 2 to move a file"
set ans = ($<)
switch ($ans)
case 1:
    echo "Enter the name of the file to be copied and press RETURN"
    set fn = ($<)
    echo "Enter the name of the file to copy to and press RETURN"
    set cf = ($<)
    cp $fn $cf
    echo "$fn copied to $cf"
    breaksw
case 2:
    echo "Enter the name of the file to be moved and press RETURN"
    set mf = ($<)
    echo "Enter the new name of file and press RETURN"
    set nf = ($<)
    mv $mf $nf
    echo "$mf is moved to $nf"
    breaksw
default:
  echo "You didn't give me a valid number; aborted"
endsw
```

Test your script:

```
% case.script
This script allows you to copy or move a file
Enter the item numoer and press RETURN
1 for copy a file or 2 to move a file
1
Enter the name of the file to be copied and press RETURN
file1
Enter the name of the file to copy to and press RETURN
file1.sample
file1 copied to file1.sample
% case.script
This script allows you to copy or move a file
Enter the item number and press RETURN
1 for copy a file or 2 to move a file
5
You didn't give me a valid number; aborted
%
```

FLOW CONTROL - GOTO STATEMENTS

goto statement format:

```
goto label
...
label: . . .
```

Example script for selecting a printer:

```
#!/bin/csh -f
echo "This does an iprx on the file"
echo "Enter the file name"
set fn = ($<)
echo "Select the number of the printer you want"
again:
echo "1 for swip, 2 for vaxip, or 3 for tacip"
set p = ($<)
echo $p
switch ($p)
case 1:
  iprx -Pswip $fn
  breaksw
case 2:
  iprx -Pip $fn
  breaksw
case 3:
  iprx -Ptacip $fn
  breaksw
default:
echo "Nothing printed; invalid printer number"
echo "Please renter your selection"
  goto again
endsw
```

goto Statements

Modify your *add.script* to perform the following:

1. Ask the operator "Do you want to try again (y or n)?"
2. If the answer is *y*, repeat sequence.
3. If the answer is *n*, clear the screen and display "Thank you"

Your script looks similar to:

```
#!/bin/csh -f
echo "I add two numbers"
again:
echo "Type your first number and press RETURN"
set a = ($<)
echo "Type your second number and press RETURN"
set b = ($<)
@ total = $a + $b
echo "The total is:  $total"
echo "Do you want to do this again?"
echo "Enter y or n and press RETURN"
set r = ($<)
   if ($r == y) then
      goto again
   endif
   if ($r == n) then
     clear
     echo "Thank you"
   endif
```

Test your script:

```
% add.script
I add two numbers
Type your first number and press RETURN
5
Type your second number and press RETURN
978
The total is:  983
Do you want to do this again?
Enter y or n and press RETURN
y
Type your first number and press RETURN
78
Type your second number and press RETURN
89
The total is:  167
Do you want to do this again
Enter y or n and press RETURN
n
(screen clears)
Thank you
%
```

FLOW CONTROL – SHIFT WITH A LOOP

- shift causes members of *argv* to be shifted to the left, discarding *argv[1]*

```
#
# This shows how shift works in a loop
set zed = (Test this again)
echo "Before using shift, there are $#zed arguments"
while ($#zed > 0)
    echo "another arg is $zed[1]"
    shift zed
end
echo "After using shift, there are $#zed arguments"
```

Execute the script:

```
% shift.script
Before using shift, there are 3 arguments
another arg is Test
another arg is this
another arg is again
After using shift, there are 0 arguments
```

- shift causes members of any *array* to be shifted to the left, discarding *array[1]*

Shift with a Loop

Write a *shift.script* that performs the following:

1. Uses the *shift* command.
2. *vi* or *emacs* on all files named on the command line.
3. includes a message that explain what the script does.

Your scripts looks similar to:

```
#!/bin/csh -f
echo "This script allows you to edit all of the files you named on the command line"
while ($#argv > 0)
     emacs $1
     shift
end
```

Test your script:

```
% shift.script pre.script data
#
echo $home
echo $cwd
echo $shell
(save the file)
John Foo 456
Sally Jones 234
Fred Smith 123
Edgar Thompson 345
(save the file)
%
```

FILENAME MODIFIERS

To extract a portion of a file pathname, use filename modifiers:

:r Retrieves the root of the filename

:e Retrieves the extension of the filename

:h Retrieves the head of the filename

:t Retrieves the tail of the filename

EXAMPLES:

```
% set p=/mnt/jsmith/sample/my.script
% echo $p:r
/mnt/jsmith/sample/my
% echo $p:e
script
% echo $p:h
/mnt/jsmith/sample
% echo $p:t
my.script
```

To build a new filename:

```
% echo $p:r.newfile
/mnt/jsmith/sample/my.newfile
```

Note: You can only use 1 : operator per command.

Filename Modifiers

Using the filename */mnt/foo/sample.text*, write a script name *ext.script* that:

1. asks the user for the absolute pathname.
2. displays the absolute pathname of the file and a message explaining what is being displayed.
3. displays the root of the filename and a message explaining what is being displayed.
4. displays the head of the filename and a message explaining what is being displayed.
5. displays the tail of the filename and a message explaining what is being displayed.

Your script looks similar to:

```
#!/bin/csh -f
echo "What is the absolute pathname of the file to be examined?"
set ans=($<)
echo "The absolute pathname is $ans"
echo "The root of the filename you gave me is $ans:r"
echo "The extension of the filename you gave me is $ans:e"
echo "The head of the filename you gave me is $ans:h"
echo "The tail of the filename you gave me is $ans:t"
```

Execute your script:

```
% ext.script
What is the absolute pathname of the file to be examined?
/mnt/foo/sample.text
The absolute pathname is /mnt/foo/sample.text
The root of the filename you gave me is /mnt/foo/sample
The extension of the filename you gave me is text
The head of the filename you gave me is /mnt/foo
The tail of the filename you gave me is sample.text
%
```

(Note: These filename modifiers will not work with $0.)

FILE OPERATORS

-d *filename* — True if *filename* is a directory
```
if (-d /mnt/usr) then
```

-e *filename* — True if *filename* exists
```
if (-e /mnt/usr/myfile) then
```

-f *filename* — True if *filename* is a regular file
```
if (-f /mnt/usr/myfile) then
```

-o *filename* — True if you are the owner of *filename*
```
if (-o /mnt/usr/myfile) then
```

-r *filename* — True if *filename* is readable
```
if (-r myfile) then
```

-w *filename* — True if *filename* is writable
```
if (! -w myfile) then
```

-x *filename* — True if *filename* is executable
```
if (-x a.out) then
```

-z *filename* — True if *filename* is empty
```
if (! -z myfile) then
```

EXAMPLE:

```
% if (-e view.script) echo yes
yes
% if (-e poem) echo yes
%
```

(Nothing is displayed because the file does not exist)

File Operators

Write a *check.script* that:

1. asks the user for a filename.
2. prints a message when the file exists in the home directory.
3. prints a message if the file is executable.
4. prints a message if the file is writable.

Your script looks similar to:

```
#!/bin/csh -f
echo "What file do you want me to check?"
set a = ($<)
cd $home
if (-e $a) echo "$a exists"
if (-x $a) echo "$a is executable"
if (-r $a) echo "$a is readable"
if (-l $a) echo "$a is a link"
if (-p $a) echo "$p is a pipe"
if (-w $a) echo "$a is writable"
```

Verify that the script works correctly. Your entry:

```
% check.script
What file do you want me to check?
data
data exists
data is readable
data is writable
%
```

Supply the name of a file that does not exist. As you have not used any flow control (if-else statements), the response is the *shell* prompt.

DEBUGGING SCRIPTS

Use any of the following to debug script output:

-x Causes commands to be echoed just prior to execution

-X Causes commands, including *.cshrc* commands, to be echoed just prior to execution

-v Causes the command to be echoed after history substitution

-V Causes commands, including *.cshrc* commands, to be echoed after history substitution

The options can be combined:

```
% csh -x -v scriptname arg1 arg2
```

Note: these options can be included on the #!/bin/csh line.

Debugging C Shell Scripts

Experiment with the debugging options. Determine the difference in using *-x* and *-v*. Use these options with your *back.script*. Your entry looks similar to:

```
% csh -x back.script file1 file2
foreach i ( file1 file2 )
echo back.script is copying file1 to file1.back
back.script is copying file1 to file1.back
cp -i file1 file1.back
echo file1 is copied to file1.back
file1 is copied to file1.back
end
echo back.script is copying file2 to file2.back
back.script is copying file2 to file2.back
cp -i file2 file2.back
echo file2 is copied to file2.back
file2 is copied to file2.back
end
% csh -v back.script file1 file2
foreach i ( $* )
echo "$0 is copying $i to $i.back"
back.script is copying file1 to file1.back
cp $i $i.back
echo "$i is copied to $i.back"
file1 is copied to file1.back
end
echo "$0 is copying $i to $i.back"
back.script is copying file2 to file2.back
cp $i $i.back
echo "$i is copied to $i.back"
file2 is copied to file2.back
end
%
```

BUILT-IN COMMANDS

echo -n — Writes contents of *string* to standard output, suppressing a newline

```
echo -n "Newline suppressed"
```

exit — *shell* exits with the value of the $*status* variable - any number other than 0 indicates an error

```
if ($1 == "") then
  echo "no username specified"
  exit
endif
```

exit *(val)* — The *shell* exits with the value specified as *val*; allows you to determine where a script failed

```
if ($1 == "") then
  echo "no username specified"
  exit 1
endif
set colon = ":"
grep "^$1$colon" /etc/passwd > name_$$
wc -l name_$$ > lcnt_$$
set ncnt = `awk' {print $1}' < lcnt_$$`
if ($ncnt == 0) then
  echo "user name is not known"
  rm -r name_$$
  rm -r lcnt_$$
  exit 2
endif
...
```

Built-in Commands

Modify the *case.script*:

1. Give the user the opportunity to copy or move additional file(s) before the script "quits" by adding a message that asks if another file should be copied or moved.

   a. Repeat the sequence if yes
   b. If no, display the message: "Goodbye".
   c. If invalid answer, display the message: "Invalid answer; aborted".
2. Retain the message: "You didn't give me a valid number; aborted"; (be sure the script actually "quits" if 1 or 2 is not entered)
3. Replies to each question should be placed on the same line as the question.

Your script looks similar to:

```
#!/bin/csh -f
echo "This script allows you to copy or move a file"
again:
echo "Enter the item number and press RETURN"
echo -n "1 for copy a file or 2 to move a file: "
set ans = ($<)
switch ($ans)
case 1:
    echo -n "Enter the name of the file to be copied and press RETURN: "
    set fn = ($<)
    echo -n "Enter the name of the file to copy to and press RETURN: "
    set cf = ($<)
      cp $fn $cf
    echo "$fn copied to $cf"
breaksw
case 2:
    echo -n "Enter the name of the file to be moved and press RETURN: "
    set mf = ($<)
    echo -n "Enter the new name of file and press RETURN: "
    set nf = ($<)
      mv $mf $nf
    echo "$mf is moved to $nf"
breaksw
default:
  echo "You didn't give me a valid number; aborted"
  exit
endsw
echo "Do you want to move or copy another file"
echo -n "Enter y or n and press RETURN: "
 set r = ($<)
if ($r == y) goto again
    if ($r == n) then
        echo "Goodbye"
    else
        echo "Invalid answer; aborted"
    endif
```

Verify that your script performs correctly for all three cases, i.e, asks you to copy or move another file, if "yes", repeats sequence; if "no", displays "Goodbye" or aborts (displaying an appropriate message) when an incorrect reply is given.

BUILT-IN COMMANDS (cont.)

onintr - Ignore all interrupts during the execution of the *shell* script; (must use *kill -9* to terminate)

```
#
onintr - ignores ctrl. C
loop:
echo "Try to kill me with ^C"
goto loop
```

onintr Restores the default action by the *shell* for detected interrupts

```
#
onintr -
echo "You can't get me here"
sleep 2
echo "You can't get me here"
onintr
loop:
  echo "But you can here"
  sleep 2
goto loop
```

onintr *label* Go to the section of the script labeled *label* upon detection of an interrupt

```
#
onintr intr
loop:
 echo "Stop me with ^C"
sleep 1
goto loop
intr:
  echo "You did it"
```

Working with Interrupts

Modify the *add.script* to display the message: **interrupt detected; aborted** when an interrupt (CTRL-c) is detected.

Your script looks similar to:

```
#!/bin/csh -f
onintr intr
echo "I add two numbers"
again:
echo "Type your first number and press RETURN"
set a = ($<)
echo "Type your second number and press RETURN"
set b = ($<)
@ total = $a + $b
echo "The total is:  $total"
echo "Do you want to do this again?"
echo "Enter y or n and press RETURN"
set r = ($<)
    if ($r == y) then
        goto again
    endif
    if ($r == n) then
      clear
      echo "Thank you"
      exit
    endif
intr:
    echo "Interrupt detected; aborted"
```

Test your script:

```
% add.script
I add two numbers
Type your first number and press RETURN
5
Type your second number and press RETURN
^C
Interrupt detected; aborted
%
```

BUILT-IN COMMANDS

dirs — Displays the directory stack with the current directory leftmost.

```
% dirs
/usr  /etc
```

pushd *dirname* — Keeps a stack of directories; pushes the current directory onto a directory stack and changes the current directory to *dirname*

```
% pushd /usr/bin
/usr/bin  /usr  /etc
```

pushd — Exchanges the top 2 elements of the directory stack

```
% dirs
/usr/bin  /usr  /etc
% pushd
/usr  /usr/bin  /etc
```

pushd +*n* — Rotates the stack *n* items

```
% dirs
/usr  /usr/bin  /etc
% pushd +2
/etc  /usr  /usr/bin
```

popd — Discards the top of the directory stack and changes the current *dir* to the next element of the stack

```
% dirs
/etc  /usr  /usr/bin
% popd
/usr  /usr/bin
```

Directory Stack

Experiment with the directory stack commands. First change directories to */usr*, */etc*, and */usr/bin* using *pushd*. Display the directory stack. Then exchange the top 2 elements of the directory stack. Finally discard the top of the directory stack.

Your entry looks similar to:

```
% pushd /usr
/usr ~
% pushd /etc
/etc /usr ~
% pushd /usr/bin
/usr/bin /etc  /usr~
% dirs
/usr/bin /etc /usr ~
% pushd
/etc /usr/bin /usr ~
% popd
/usr/bin /usr ~
%
```

BUILT-IN COMMANDS (cont.)

eval — Executes a command built from variables using *eval*

```
% set a = ls
% echo $a
ls
% eval $a
file1  file2  file3
```

time *command* — Prints a summary of the time used for *command*

```
% time ls
file1  file2  file3
0.2u 01s 0:01 33% 1+2k 1+1io 1pf + Ow
```

0.2 seconds of user time
0.1 seconds of system time
0:01 seconds of elapsed real time
33% of the cpu used
1+2k memory + stack used
1+1io 1 each input and output (disk)
1pf+Ow one page faulted in; none written

source -h — Reads commands from a file and appends them to your history list

```
% source -h .cshrc
% history
4 source -h ~/.cshrc
5 set cdpath = ( ~ )
6 set history = 20
7 set notify
8 alias cd 'set old=$cwd; chdir \!*'
```

Using *eval*, *time*, and *source -h*

Assign *date* to the variable *d*. First echo the variable's value; then execute the command using the *eval* command. Your entry looks similar to:

```
% set d = date
% echo $d
date
% eval $d
Tue Dec 23 10:06:16 CST 1986
```

Execute the command assigned to *d* and display a summary of the time used to execute it. Your entry:

```
% time eval $d
Tue Dec 23 10:14:26 CST 1986
0.0u 0.0s 0:00 50% 0+0k 0+2io 1pf+0w
%
```

The following exercise demonstrates how commands can be read from a file and appended to the history list. For this exercise, read in the commands you have set in your *.logout* file; then display the history list. Your entry looks similar to:

```
%  source -h .logout
% history
     1 source -h /tac/username/.logout
     2 clear
     3 set path = /usr/games
     4 fortune
     5 /usr/local/bin/micom
     6 echo See you tomorrow!
     7 history
%
```

RESOURCE LIMITS

The built-in command *limit*:

- Allows you to control the resources:

cputime	maximum process cpu-seconds
filesize	largest single file created
datasize	maximum growth of data + stack
stacksize	maximum size of stack region
coredumpsize	size of largest cored dump
memoryuse	maximum resident set size
concurrency	maximum parallel threads

- Sets the default maximum-use scale to *k* (Kbytes) except for cputime which is seconds

- To print a list of all current limits (except cputime), type: `limit` at the *shell* prompt

- To print the current limit of a specific resource, type: `limit` *resource* at the *shell* prompt

```
% limit stacksize
stacksize    512 kbytes
```

- To set/change a resource maximum-use limit, type: `limit` *resource maximum-use* at the *shell* prompt

```
% limit stacksize unlimited
stacksize    229344 kbytes
```

Resource Limits

Determine the resource limits that are currently set. Your entry:

```
% limit
filesize        unlimited
datasize        229344 kbytes
stacksize       512 kbytes
coredumpsize    32768 kbytes
memoryuse       unlimited
%
```

Notice the limit for *cputime* does not appear in the listing. Determine what the limit is for *cputime*. Your entry:

```
% limit cputime
cputime         unlimited
%
```

Change *stacksize* to unlimited and verify that it is set to unlimited. Your entry:

```
% limit stacksize unlimited
% limit stacksize
stacksize       229344 kbytes
%
```

Exercises for Chapter 3

1. Write a script that displays: `Hi UNIX user. Here are the date and time.` Then display the output of the *date* command. Test the script.

2. Write a script that sets the variable *first* to the value of your first name; the variable *last* to the value of your last name. Using these variables, display the following line as standard output:

   `Your name is` *first last.* Verify the script.

3. Write a script named *printer* that can be executed by any user to determine the names of the printers available on the current system. Include directions on how to print a file using these printers. Have another user test your script.

4. Write a script that displays: `The date is` *display the actual date from the* **date** *command. The next line should display:* `The time is` *display the actual time as provided by the* **date** *command.* Test the script.

5. Modify the previous script so that it displays as the first line `Have a good day, Mr./Ms.` *username.* The user name should be obtained from the predefined user variable. Have another use verify the accuracy of your script.

6. Write a script that can be executed by anyone on the system. This script will display the message: `Your login name is` *username.* Your home directory is *home directory.* You have *number_of_files* in your home directory. Have some of your colleagues execute the script.

7. Write a script that displays the first argument entered by the user and the total number of arguments entered. Your script will display the message: `The first argument you entered is` *arg* and the total number of arguments you entered is *number_of_args.*

8. Make a script that changes a file to executable mode. Test the script on files in your home directory.

9. Write a script using a *foreach* statement that reads a word list from a file named *wdlist* and prints a line for each word in the word list: `I know a little girl who swallowed a` *word from wdlist.*

   The file *wdlist* contains the following list of words: *bug, frog, nickel, pizza.* Test the script.

10. Modify the previous script to use the arguments on the command line as the word list. Print the same reply for each word as you did in the previous exercise.

11. Write a script that will supply phone numbers for specified persons. First create a data base file containing the names and phone numbers (John Foo 234-9080). Your script will ask the user: `Whose phone number to you want? Enter only the first name:"`

    The script should then print out the first and last name of the person, as well as the phone number. Verify that your script works correctly.

12. Write a script that will determine the day (Monday, Tuesday, etc.) and if it is Friday, display the message `Time cards are due today!` If it is not Friday, display the message: `Time cards are due on Friday.` Test the script.

13. Using a *while* loop, write a script that asks `What do you get when it snows on Easter?`. Accept the answer *snow bunnies*. Test your script.

14. Write a script that copies one filename (argument) to *file.old*. Verify that the script works correctly.

15. Enhance the the previous script. Modify the script so that it checks to make sure that only one argument is given; use the *exit* statement if this isn't the case. If more than one argument is given, display a message that says "aborting, more than one argument given." Verify that the script works correctly.

16. Now enhance the previous script to handle more than one file at a time. If no argument is given, display a message that no argument was given and exit. Have one of your colleagues verify that your script is accurate.

17. Using the same script, check to see if *file.old* exists; if it does, refuse to copy over the existing file (but don't exit until all *file* (arguments) have been processed). Also, make sure the *file* supplied as the argument actually exists before copying. Test the script.

18. Modify the same script again. If *file.old* already exists, ask the user if it should be overwritten. Accept any answer given except "yes" as being "no." Test the script.

19. Now modify the script so that it gets the extension for the new filename from the script's name. That is, if a link is added to the script called "new," then *new_file* should copy *file* to *file.new*. (Note: Setting a variable to $0:t will not work!)

20. Write a script that renames all your fortran (.for) files to file name .f. If the name of the file is *main.for*, the new name will be *main.f*. Assume that you have several files in the directory ~/programsfc that need to be renamed. (You can make test files by using the command *touch main.for sub1.for*, etc.)

21. Write a script that displays a menu of choices for the user: copy a file; move a file; remove a file; print a file. For each menu item, make sure the script takes appropriate action. Ask another user to execute the script.

22. Write a script that:

    a. Clears the screen.
    b. Asks the user for a number.
    c. Accepts the first number and requests another number.
    d. Adds the two numbers.
    e. Display the message: `The total is:` followed by the total.
    f. Exit the program with a message to the user: `That is all for now; bye.`
    g. Verify the correctness of your script.

23. Write a script that:

    a. Clears the screen.
    b. Asks the user for two numbers.
    c. Adds the two numbers.
    d. Displays the message: `The total is:` followed by the total.
    e. Displays the message: `Do you want to try again?`
    f. If the answer is *yes*, repeat the sequence. If the answer is *no*, clear the screen and exit the program with message to the user: `Thank you!`
    g. Test the script.

24. Write a script that:

    a. Generates a menu that allows the user to select a mathematical operation. Use the code: A = add; S = subtract; M = multiply; D = divide.
    b. Ask the user to make a selection from the menu.
    c. Ask the user for 2 numbers.
    d. Verify that the user entered just two numbers. If two numbers were not given, ask the for the numbers again.
    e. Perform the selected option.
    f. Display the result of the operation.
    g. Ask the user if he/she wants to do another calculation. If *yes*, repeat the preceding sequence. If *no*, exit with a message to the user: `Calculations complete. Exiting.`
    h. Verify that the script works.

4

Advanced Text Manipulation Commands

Objectives for Chapter 4

After completing Chapter 4, you will be able to:

1. Use *sed* on the command line and in a *sed* script file.
2. Make selection of lines to edit by number or pattern using *sed*.
3. Determine which lines match a specified context address for *sed*.
4. To delete, change, replace, or append text using *sed*.
5. Write editing changes made with *sed* to a file.
6. Use *awk* on the command line and in an *awk* script file.
7. Determine which lines match a specified regular expression for *awk*.
8. Make selections with *awk* using one or two patterns.
9. Output specified records and fields using *awk*.
10. Redirect the standard output to one or more output files using *awk*.
11. Change the field or record separator (when appropriate) using *awk*.
12. Print *awk* output using either the *print* or *printf* command.
13. Use predefined variables in *awk* scripts.

USING THE NON-INTERACTIVE TEXT EDITOR - *sed*

sed is:

- Designed to edit files too large for comfortable interactive editing
- Designed to edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode
- Able to perform multiple global editing functions in one pass through the input
- Allows using a command file or online editing commands; most effective when using editing scripts as a command file
- Based on the *ed* and *ex* editors BUT *sed* has critical differences
  - By default, *sed* will process all the data in the input file.
  - Therefore, *sed* has commands which allow you to control the sequence of editing performed.

Using the *sed* Editor

To complete the following exercises, you will need to create a file named *sample* with your favorite text editor. The file should contain the following contents exactly as shown:

```
0123456789:
32:10
0 1 2 3 4 5 6 7 8 9
1:
2
3
a
abcdefg test
aaaa
aaa
aa
a
za
zzzzah:
carriage return (blank line)
```

You will also need to create a file named **datafile** containing the following contents:

```
      PROGRAM SAMPLE

C THIS IS INCLUDED
INCLUDE PROG.FOR

* THIS IS A COMMENT

      WRITE(6,*) a, b, c
      WRITE(6,100)
      WRITE(6,200)
      WRITE(6,300)
      WRITE(6,400)
  100 FORMAT(' abc ')
  200 FORMAT(" def ")
  300 FORMAT(* zzz *)
  400 FORMAT(' ***** ')
      END
```

sed FORMAT

- Has the form:

  sed [*options*] [*address1*][,*address2*]] *function* [*arguments*]

  The options:

  -e Allows you to specify a single editing command on the command line

  sed -e *expression file_to_edit*

  % sed -e 's/an/and/' sample

  -f Allows you to specify the name of a script containing editing commands

  sed -f *scriptfile file_to_edit*

  % sed -f edit.script sample

  -n Specifies "nocopy"; copy only those lines specified by the *p* function; by default all lines go to output

  sed -n /*expression*/p *file_to_edit*

  % sed -n '/an/p' sample

  % sed -n 's/an/and/p' sample

  % sed -n -f edit.script sample

sed Options

The following two commands lines show a substitution of *an* for *and.*

Enter the following two lines at the *shell* prompt; then compare the output.

sed -e 's/an/and/' poem
sed -n 's/an/and/p' poem

```
% sed -e 's/an/and/' poem
In Xandadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, rand
Through caverns measureless to mand
My Xandadu checkline.
Down to a sunless sea.
% sed -n 's/an/and/p' poem
In Xandadu did Kubla Khan
Where Alph, the sacred river, rand
Through caverns measureless to mand
My Xandadu checkline.
%
```

Notice that the use of the *-n* option changes the output. How was the output changed with the *-n* option?

ADDRESSES - LINE SELECTION

Select lines for editing by line number or context

- Addressing by number:
  - Line numbers are consecutive throughout the entire stream; not restarted with each file during multiple file edits
  - Can be a dollar sign ($), which represents the last line of the last input file
  - Has the format:

line_number function [*arguments*]

```
% sed -e '3s/an/and/' editfile
% sed -n '3 p' editfile
% sed -e '$s/sun/fun/' editfile
% sed -n '$ p' editfile
% sed -f scriptfile editfile
```

Selecting Lines for Editing By Number

Using *sed*, print the last line of the file *poem*. Your entry:

```
% sed -n '$ p' poem
Down to a sunless sea.
%
```

What would be printed if you use the command:

sed -n '$ p' sample poem

Would the last line of each file be printed? Enter the command to see what happens. Your entry:

```
% sed -n '$ p' sample poem
Down to a sunless sea.
%
```

ADDRESSES - LINE SELECTION (cont.)

- Context address:
  - Is a pattern enclosed in slashes
  - Are constructed as follows:

`/file/`	Matches lines containing a string of *file*
`/[0123456789]/` or `/[0-9]/`	Matches lines containing strings of any digit(s)
`/[Uu]nix/`	Matches lines containing strings of either *Unix* or *unix*
`/[^abcd]/`	Matches lines containing strings of any symbols other than *a*, *b*, *c*, or *d* (individually or combination)
`/^abc/`	Matches lines containing strings of *abc* at the beginning; does not match *" abc"*
`/abc$/`	Matches lines containing string(s) of *abc* at the end
`/^$/`	Matches null lines
`/^.$/`	Matches a line containing exactly one character
`/x.y/`	The dot . matches any single character, i.e., *x+y*, *x-y*, *x y*, or *x.y*

Context Addresses

Using the file *poem*, what line(s) will be printed when you enter the following command:

sed -n '/^an/p' poem

Enter the command to see if your answer was correct. Your entry:

```
% sed -n '/^an/p' poem
%
```

As you can see, no lines match; there are no lines beginning with *an*.

Enter a command using *sed* that prints all lines in *sample* except those containing only the characters *a*, *b*, or *c*.

```
% sed -n '/[^abc]/p' sample
0123456789:
32:10
1 2 3 4 5 6 7 8 9
1:
2
3
abcdefg test
za
zzzzah:
%
```

The line containing "abc" should be printed as it contains a character NOT (^) in the list *a*, *b*, *c*.

Note: The circumflex (^) as the first character in the bracket reverses the sense. It tries to match any one character NOT in the list.

Enter a command using *sed* that removes all lines in *sample* containing trailing blanks.

```
% sed -e 's/ *//' sample
0123456789:
32:10
1 2 3 4 5 6 7 8 9
1:
2
3
abcdefg test
za
zzzzah:
%
```

sed COMMANDS

- Commands are named by a single character
- *sed* recognizes:

a\
<*text*>
	Causes the *text* to be written to output after the line matched by its address; *a* must be followed by \; *text* may contain any number of lines, with each line, except the last, ending with \
	`$ a\`
	`These two lines are appended\`
	`to the end of the file.`

b*label*	Branches to command : *label* used with *sed* script file

c\
<*text*>
	Deletes the lines selected by its address(es) and replaces them with the lines in *text*; *c* must be followed by \; *text* may contain any number of lines, with each line, except the last, ending with \
	`2 c\`
	`Two new lines of code\`
	`are replacing line 2.`

d	Deletes from the named file (does not write to output) lines matched by its address(es) `sed -e /tryout/d sample`
g	Replaces all occurrences `sed -e 's/can/may/g' sample`

sed Commands

Delete the line that begins with *My Xanadu* in the file *poem*. Your entry:

```
% sed -e '/^My/d' poem
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
%
```

Experiment with global substitution. The following exercise is for demonstration purposes only to show you how global substitution takes place. Substitute globally an *i* for an *a* in the file *poem*. Your entry:

```
% sed -e 's/a/i/g' poem
In Xinidu did Kibli Khin
A stitely pleiusre dome decree:
Where Alph, the sicred river, rin
Through civerns meisureless to womin
My Xindidu checkline.
Down to i sunless sei.
%
```

If you had not included the *g* with the command, only the first occurrence of *a* in each line would have been changed to an *i*.

sed COMMANDS (cont.)

i\
<*text*>
	Causes the *text* to be written to output before the line matched by its address; *i* must be followed by \; *text* may contain any number of lines, with each line, except the last, ending with \
	`2i\`
	`Inserts two lines of characters\`
	`before line 2.`

p	Prints only the line(s) that the commands were executed on when used with the *-n* option `sed -n -e 's/can/may/p' sample`
q	Causes current line to be written to output (if it should be) and execution to terminate `sed -e '/abc/q' sample`
r *file*	Reads *file* into output following the specified address `sed -e '/^$/r note1' sample`

s/*old*/*new*/*f*	Substitutes *new* for *old*.
	If searched by pattern, replaces first occurrence in each line; if *f=g*, replaces all occurrences; with *-n*, if *f=p*, prints lines where replacement occurred; if *f=w file*, writes lines with changes to *file*.
	`sed -e '/\.\./s/\.\./\./' sample`
	`sed -e 's/\.\./\./g' sample`
	`sed -e '/\.\./s//\./gp' sample`
	`sed -e 's/\.\./\./gw test' sample`

Making Substitutions with *sed* Commands

Substitute *Phoney* for *Khan* in the file *poem*. Your entry:

```
% sed -e 's/Khan/Phoney/' poem
In Xanadu did Kubla Phoney
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
My Xanadu checkline.
Down to a sunless sea.
%
```

Again change *Khan* to *Phoney*; instead of displaying the entire file, use the command that prints only the line that you changed. Your entry:

```
% sed -n 's/Khan/Phoney/p' poem
In Xanadu did Kubla Phoney
%
```

For this exercise, again change *Khan* to *Phoney* and write the changed line to a file named *poem.test*. Your entry:

```
% sed -n 's/Khan/Phoney/w poem.test' poem
%
```

Now verify your results by using viewing the contents of *poem.test*:

```
% more poem.test
In Xanadu did Kubla Phoney
%
```

sed COMMANDS (cont.)

t*label*	Tests whether any successful substitutions have been made on current input line; if so, branches to *label*; otherwise, does nothing Used with *sed* script file
w *file*	Writes only the addressed line(s) to the named file `sed -e 's/can/may/w file1' sample`
y/*str1*/*str2*/	Replaces each character from *str1* with corresponding character from *str2* `sed -e 'y/abc/ABC/' sample`
=	Prints the current input line number `sed -n '/abc/=' sample`
!*cmd*	Causes the next command to be applied only to those lines not selected by the address `sed -e '/^You/\!s/can/may/' sample`
:*label*	Sets label for *b* and *t* commands Used with *sed* script file
{	Treats commands up to matching } as a group Used with *sed* script file

Making Replacements with *sed* Commands

Print the current input line number for those lines containing *abc* in file *sample*. Your entry:

```
% sed -n '/abc/=' sample
8
%
```

Replace every *a*, *b*, or *c* with *ABC* in the file *sample*. Your entry:

```
% sed -e 'y/abc/ABC/' sample
0123456789:
32:10
0 1 2 3 4 5 6 7 8 9
1:
2
3
A
ABCdefg test
AAAA
AAA
AA
A
zA
zzzzAh:

%
```

CREATING A *sed* SCRIPT FILE

- Open a file with a ConvexOS text editor
- Enter the commands, each on a separate line
- Begin comment lines with a # in the first column of a line
- To execute the *sed* commands contained in a file, use the form:

  `sed -f` *sed.script file_to_edit* `>` *new_file*

  ```
  % sed -f sed.script sample > test
  ```

- A sample *sed.script* file:

```
# This is a small example
# of a sed script
2i\
This line inserts before line 2.
# The following command deletes
# the line(s) containing ran
/ran/d
$ a\
This is appended as 2 lines\
to the end of the file.
# The following command changes
# each occurrence of Khan
s/Khan/Phoney/g
```

Creating a *sed* Script File

Create a *sed* script file *poem.script* that performs the following in the file *poem*:

1. Capitalize all characters in the file.
2. Delete the line that begins with *MY*.
3. Change *ALPH* to DELPHIA

Send the output to a file named *test*.

Your entry looks similar to using the *ex* editor:

```
% ex poem.script
"poem.script" [New file]
:a
# This line capitalizes.
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/

# This deletes the line.
/^MY/d

# This changes ALPH to DELPHIA
s/ALPH/DELPHIA/g
.
:wq
"poem.script" [New file] 8 lines, 151 characters
% sed -f poem.script poem > test
%
```

Verify the results by viewing the contents of *test* using the *more* command. Your entry:

```
% more test
IN XANADU DID KUBLA KHAN
A STATELY PLEASURE DOME DECREE:
WHERE DELPHIA, THE SACRED RIVER, RAN
THROUGH CAVERNS MEASURELESS TO MAN
DOWN TO A SUNLESS SEA.
%
```

sed SCRIPT FILE

The following illustrates using labels and command grouping:

```
# sample sed script file
# processing is in batch mode

y/ABCDEF/abcdef/

# if above test is true, go to down

t down

/[012]/{
 s/0/0.0/g
 t mid
 }

s/@/ /g

:mid

/\$ -/bdown

/-\$/s//\$ -/

:down
```

Executing *sed* Scripts

Create and execute a *sed* script named *ftrn.script* that:

1. Converts VAX Include statement at beginning of statement to read #include instead of INCLUDE
2. Converts uppercase to lowercase letters
3. Converts double quotes to single quotes.
4. Begins all comments with 'c'.
5. Replaces asterisk with single quote; if asterisk(s) are inside single quotes, do not change them.
6. Removes all empty (null) lines.

Test your script on the file named *datafile* sending the output to a file named *ftrn*; verify the results by viewing the contents of the file. Your program looks similar to:

```
# convert VAX include statements into CONVEX
# statements (only at beginning of line)
s/^INCLUDE/#include/
# if 'include' statement, avoid all other tests
t jump
# convert uppercase letters to lowercase
y/ABCDEFGHIJKL/abcdefghijkl/
y/MNOPQRSTUVWXYZ/mnopqrstuvwxyz/
# convert double quotes into single quotes
y/"/'/
# begin all comments with 'c' instead of '*'
/^\*/s//c/
# convert non-standard format statements
/format/{
# do not change asterisks inside single quotes
     /'.*\*.*'/bjump
# replace asterisk with single quote
     s/\*/'/g
}
:jump
# do not output empty lines
/^$/!p
# end of sed command script
:end
```

Verify your program:

```
% sed -n -f ftrn.script datafile > ftrn
% more ftrn
    program sample
c this is included
#include PROG.FOR
c this is a comment
    write(6,*) a, b, c
    write(6,100)
    write(6,200)
    write(6,300)
    write(6,400)
  100 format(' abc ')
  200 format(' def ')
  300 format(' zzz ')
  400 FORMAT(' ***** ')
    end
```

THE *awk* PATTERN SCANNING AND PROCESSING LANGUAGE

- Is a programming language designed to handle text manipulation with the form:

  `awk '/pattern/ {action}' foofile`

- Searches one or more files for patterns
  - Each line of input is matched against each of the patterns in turn
  - For matches, the associated action is executed

- Does not alter input files

- Either the pattern or the action may be left out, but not both
  - No action - matching line copied to output
  - No pattern - action is performed for every input line

- Used in a variety of operations:
  - Generating reports
  - Matching patterns
  - Validating data
  - Filtering data for transmission

Pattern Scanning and Processing Language

If you do not have a file called *sample* with the following, contents, create the file.

0123456789:
32:10
0 1 2 3 4 5 6 7 8 9
1:
2
3
a
abcdefg test
aaaa
aaa
aa
a
za
zzzzah:
carriage return (blank line)

Create a file named *inventory* with the following contents:

smith:bears:10:8
jones:table:3:1200
wilson:bats:100:10
james:batteries:1000:1
moss:books:500:12

What will happen when you try to use *awk* without putting {} around the action? Experiment; try the command:

awk '{print NR, $0}' sample

without the brackets. Your entry:

```
% awk 'print NR, $0' sample
awk: syntax error near line 1
awk: bailing out near line 1
%
```

What happens if you leave the slashes off a pattern? Experiment; try the command:

awk 'THIS' datafile

```
% awk 'THIS' datafile
awk: syntax error near line 1
awk: bailing out near line 1
%
```

RECORDS AND FIELDS

- *awk* divides input into records
  - Each line is considered a record

- Each record is terminated by a record separator; the default is a newline
  - By default *awk* processes input one line at a time

- Each record is divided into fields
  - Fields are normally separated by blanks or tabs

    The record:

    Name City State

    has three fields, each separated by a tab

  - The field separator can be changed either with *-Fseparator* option or with a predefined variable, *FS*

Records and Fields

You have a file with the following contents. How many records does it contain? What is the record separator?

```
Freddy Foo
Greedy Smith
Sally Greed
```

How many fields are contained in the above listing? What is the field separator?

awk USAGE

- There are two methods of using *awk*:
  - Supply pattern/actions on command line:

    awk [*option*] '*/pat/{act}*' *file(s)*

    ```
    % awk '/abc/{print $0}' sample
    ```

    awk [*option*] '*/pat/*' *file(s)*

    ```
    % awk '/this/' sample
    ```

    awk [*option*] '*{act}*' *file(s)*

    ```
    % awk ' {print NR, $0} ' sample
    ```

  - Place the *awk* commands in a file; use the *-f* option to indicate the commands are in the named file

    awk -f *awk.file file(s)*

    ```
    % awk -f awk.script sample
    ```

- To indicate a field separator other than the default (blank), use *-F* followed by the actual field separator

  ```
  % awk -F'*' -f awk.script sample
  % awk -F: -f awk.script sample
  % awk -F: '{print NR, $0}' sample
  ```

awk Usage

Experiment with the *awk* command by entering the following two command lines:

```
awk '/abc/{print $0}' sample
awk '{print NR, $0}' sample
```

Your entry and the results:

```
% awk '/abc/{print $0}' sample
abcdefg test
% awk '{print NR, $0}' sample
1 0123456789:
2 32:10
3 0 1 2 3 4 5 6 7 8 9
4 1:
5 2
6 3
7 a
8 abcdefg test
9 aaaa
10 aaa
11 aa
12 a
13 za
14 zzzzah:
15
%
```

PREDEFINED VARIABLES

`$0`	The entire input record `awk '{print $0}' datafile`
`$1`	The first field of the input record `awk '{print $2, $1}' datafile`
`$n`	The *n*th field of the input record `awk '{print $1, NF}' datafile`
`length`	Length of the current input record `awk '{print length ($0), $0}' datafile`
`FILENAME`	Name of the current input file `awk 'BEGIN {print FILENAME}' datafile`
`NR`	Number of the current record First input record is 1 `awk '{print NR, $0}' datafile`
`NF`	Number of fields in the current record First input field is one `awk '{print NR, NF}' datafile`
`FS`	Input field separator (default blank) `awk 'BEGIN {FS=":"} {print NF}' datafile`
`RS`	Input record separator (default newline) `awk '{RS=":"; print NR}' datafile`
`OFS`	Output field separator (default blank) `awk '{OFS=":"; print $1,$2,$3}' datafile`
`ORS`	Output record separator (default newline) `awk '{ORS=":"; print $0 }' datafile`
`OFMT`	Output format for numbers (Default %.6g) `awk '{OFMT="%.5g"; print $2 }' datafile`

awk Predefined Variables

Determine how many fields each of the records contain in the file *sample*; specify a colon (:) as the field separator. Print the number of fields, as well as each record. Your entry:

```
% awk -F":" '{print NF, $0}' sample
2 0123456789:
2 32:10
1 0 1 2 3 4 5 6 7 8 9
2 1:
1 2
1 3
1 a
1 abcdefg test
1 aaaa
1 aaa
1 aa
1 a
1 za
2 zzzzah:
0
%
```

Using the same file, determine how many records are in the file when you specify a colon as the record separator. Print the record number and the contents of each record. Your entry:

```
% awk '{RS=":"; print NR, $0}' sample
1 0123456789:
2 32
3 10
0 1 2 3 4 5 6 7 8 9
1
4
2
3
a
abcdefg test
aaaa
aaa
aa
a
za
zzzzah
5

%
```

REGULAR EXPRESSIONS

A regular expression can be used as the pattern that controls the action in an *awk* program. The pattern must be enclosed in slashes.

`/file/`	Matches a string of *file*, *file1*, etc.
`/[a-zA-Z0-9]/`	Matches a string containing any character or digit within the brackets
`/[Uu]nix/`	Matches string containing either *Unix* or *unix*
`/x.y/`	The dot . matches any single character, e.g., *x+y*, *x-y*, *x y*, or *x.y*
`/file[a|b]/`	Matches strings containing *filea* or *fileb*, e.g., *fileac* and *filebxxx*
`/fil?/`	Matches the string *fil* followed by zero or more characters, e.g., *fil*, *fill*, *file*, *fileabc*
`/Ze*/`	Matches zero or more occurrences of *e*; matches all strings beginning with *Z*
`/0+9/`	Matches any number of zeroes followed by a *9*
`/[^0123456789]/`	Matches all strings that do not contain only digits
`/^\*/`	Matches all lines beginning with *
`/(a|b)c+/`	Matches strings containing *ac*, *bc*, and *abc* but not *a*, *b*, or *c* individually

Regular Expressions

Write a command using *awk* that prints only those lines containing digits; then execute the command on the file *sample*. Your entry:

```
% awk '/[0-9]/' sample
0123456789:
32:10
0 1 2 3 4 5 6 7 8 9
1:
2
3
%
```

Now write and execute a command that prints all lines in the file *sample* except those containing only digits. Your entry:

```
% awk '/[^1-9]/' sample
0123456789:
32:10
0 1 2 3 4 5 6 7 8 9
1:
a
abcdefg test
aaaa
aaa
aa
a
za
zzzzah:
%
```

Yes, lines containing digits are printed; however, those lines contained characters other than digits, i.e., colons (:) and spaces.

SELECTION CRITERIA

- Selection can be made by one or two patterns:

  awk '*/pat1/ , /pat2/ {act}*' *filename*

  awk '*/pat/ {act}*' *filename*

  The line:

  ```
  % awk '/start/, /stop/' sample > newfile
  ```

  outputs all lines inclusively, beginning with *start* and ending with *stop*, to the file *newfile*; the default action is print

  The line:

  ```
  % awk '/start/' sample
  ```

  prints all lines containing *start*

- Selection can also be made by matching the value of a variable:

  ```
  % awk 'NR==100, NR==200' sample
  ```

  prints lines 100-200, inclusively

  The line:

  ```
  % awk 'NF < 3' sample
  ```

  prints the entire record when the fields in the record are less than 3

Selection Criteria

Print the lines in the *sample* file beginning with the line *32:10* and ending with the line *a*. Your entry:

```
% awk '/32:10/, /a/' sample
32:10
0 1 2 3 4 5 6 7 8 9
1:
2
3
a
%
```

awk OPERATORS

The *awk* operators in decreasing order of precedence (operators in the same group have the same precedence with regard to each other)

`++a  a++` `--a  a--`	Increments (decrements) *a*, either before or after using *a*'s value
`* / %`	Multiply, divide, and remainder `NF % 2 == 0` matches lines with even numbers of fields
`+  -`	plus, minus `tot = $1 + $2` (sets *tot* equal to the sum of $1 and $2)
`no symbol`	String concatenation `long = $1 $2` (sets the variable *long* equal to the concatenation of the first two fields)
	Relational operators:
`>  >=`	GT, GE
`<  <=`	LT, LE
`==  !=`	EQ, NE
`~  !~`	Similar to, Not similar to; `$2 ~ /A|B|C/` specifies a match of all lines with *A*, *B*, or *C* in the second field

awk Operators

Assume each of the following is a "small" *awk* program; determine the results for each.

```
a = 2
++a
```

What is the value of *a* before and after executing the program?

```
b = 5
c = b/2
```

What is the value of *c*?

```
b = 5
d = b % 2
```

What is the value of *d*?

awk OPERATORS (cont.)

!	Negate value of expression `'$1 != prev {print; prev = $1}'` prints all lines in which the first field is different from the previous first field
&&	Boolean AND `$1 >="s" && $1 < "t"` is evaluated left to right; selects lines where the first field begins with the letter *s*
||	Boolean OR `$1 == "a" || $1 == "b"` is evaluated left to right; selects lines where the first field begins with either the letter *a* or *b*
=	Assignment operators:
+= -=	*var op= expr* is equivalent
*= /=	to *var = var op expr*;
%=	*s1 += 1* sets variable *s1* equal to the value of *s1* plus one

More *awk* Operators

What does the following statement mean?

```
if ( $1 > max) max = $1
```

What does the following statement mean?

```
if ($2 != "stop") icont += 1
```

awk PRINT COMMAND

To print some or all the records, use the *awk* print command

- Copy the input file to the standard output file

```
% awk '{print}' sample
```

- Print first two fields in reverse order separated by the output field separator

```
% awk '{print $2, $1}' sample
```

- Print first two fields concatenated together; no space between printed fields

```
% awk '{print $1 $2}' sample
```

- Print each record preceded by the record number and the number of fields

```
% awk '{print NR, NF, $0}' sample
```

The *print* Command

First print the first two fields in file *poem*; then print the first two fields concatenated together, without a space between printed fields. Your entry:

```
% awk '{print $1, $2}' poem
In Xanadu
A stately
Where Alph,
Through caverns
My Xanadu
Down to
% awk '{print $1 $2}' poem
InXanadu
Astately
WhereAlph,
Throughcaverns
MyXanadu
Downto
%
```

VARIABLE ASSIGNMENTS

- Values set according to context:

  - numeric

    ```
    x = 1
    ```

  - string

    ```
    x = "smith"
    ```

- By default variables are initialized to the null string - a numeric value of zero

- Strings are converted to numbers and numbers to strings whenever context demands it:

  ```
  x = "3" + "4"
  ```

  assigns 7 to *x*

- Strings that cannot be interpreted as numbers generally have a numeric value of zero

Variable Assignment

In the following list of variables indicate the value of *d*, *k*, *l* and *m*.

```
a = "some"
b = "thing"
c = "" (blank)
d = acb
i = "1"
j = 4
k = i + j
l = a + b
m = j + a
```

awk PROGRAM FILE

awk is an interpretive language; there are no declarations. A variable may take on different types of values within the same program.

- Creating an *awk.script* program file:
  - Begin comment lines with a # in the first column of a line or append comments to any line in the program by preceding the comment with a #
  - Multiple statements on a line are permitted; separate them with semicolons
  - To continue statements on successive lines, terminate the line with a \
  - Surround string constants with double quotes; to place a double quote in a string, precede it with \
  - A record may contain multiple lines, if RS (record separator) is not a newline

The *awk* Program File

Which of the following lines are not acceptable in an *awk* program file? Why?

```
The following adds the second and fourth fields
#and prints the sum.
{sum == $2 + $4 print s}
#The following prints each record, the total
#of items (fields 3 and 4) and the total sales for
#each item (amount in field 2 times total items)
{total == $3 + $4}
{print $0, total, total*$2}
```

INITIALIZATION AND WRAPUP

BEGIN and *END* provide a way to handle initialization and wrapup (within the *awk* script file):

- *BEGIN* matches the beginning of the input (before the first input record is read);
  - If *BEGIN* is present, it must be the first pattern

    The line:

    ```
    BEGIN { FS = ":" }
    . . .
    ```

    sets the field separator character to a colon
- *END* matches the end of input (after the last record has been processed)
  - If *END* is present, it must be the last pattern

    The line:

    ```
    . . .
    END { print NR }
    ```

    prints the number of records processed

Using *BEGIN* and *END* in *awk* Programs

Use your favorite editor to create an *awk* program called *number.script* that can be used to determine the number of words in a file. (Do not use the UNIX command *wc* as your program.) Use *BEGIN* and *END* in your program. Test your script on the file *poem*. Your program looks similar to:

```
BEGIN       {nw = 0}

        {nw += NF   #Determines number of words
        }

END     {print "number of words =     ",nw
        }
```

Your entry for testing looks like:

```
% awk -f number.script poem
number of words = 29
%
```

PROGRAM FLOW CONTROL

for

for (*expr1*;*condition*;*expr2*) *stmnt*

```
{   for (i=1;i<=NF;i++)
    print $1
}
```

for

for (*i* **in** *array*) *stmnt*

```
{  for(i in array) {print i, array[i]}  }
```

if

if (*cond*) *stmnt1* **else** *stmnt2*

```
{ if (maxpop < $3) {
    maxpop = $3
    country = $1}
}

{ if ($1 >= 0) {
    val1 /= $1
    val2 *= $1 }
  else {
     val1 = 1 ;val2 = $1 }
}
```

while

Uses the form: **while** (*cond*) *statment*

```
{   i = 1
    while (i <= NF) {
      print $ i; ++i}
}
```

Program Flow Control

Using either a *for* or *while* loop, print out the file *poem* with each field (word) on a separate line. Create either a file named *for.script* or *while.script*. Your program looks similar to either of the following:

```
{   for (i=1;i<=NF;i++)
    print $i
}

{   i = 1
    while (i <= NF) {
      print $ i; i++}
}
```

To test your program:

```
% awk -f while.script poem
In
Xanadu
did
Kubla
Khan
A
stately
pleasure
dome
. . .
. . .
. . .
```

PROGRAM FLOW CONTROL (cont.)

break	Exit immediately from a *for* or *while* loop
	`for (i=1;i<=NF;i++) {`
	`  if ($1 == "stop") break`
	`}`

continue	Keep going until the loop is completed
	`for (i=1;i<=NF;i++) {`
	`  if ($1 == "end") continue`
	`}`

exit	If found in the *BEGIN* section, the program stops and does not execute the *END* section. If found in the main body, the program stops and the *END* section is executed. An exit in the *END* section causes immediate termination.
	`for (i=1;i<=NR;i++) {`
	`  icont += 1`
	`  if (icont > 100) exit`
	`}`

getline	Replaces current record with next input record and continue processing
	`for (i=1;i<=NF;i++) {`
	`  if (NF < 3) getline`
	`}`

next	Skips to next input record and begins processing from top of program
	`for (i=1;i<=NF;i++) {`
	`  if (NF == 1) next`
	`}`

More on Program Control

Write an *awk* script named *awk.voc* that prints out the contents of *poem* as a vocabulary list. The list can be one column but identical words can appear only once in your listing.

Your program:

```
BEGIN {
        icnt = 0
}

{
        for (i=1; i<=NF; i++) {
                iflag = 0
                for (j=1; j<=icnt; j++) {
                        if ($i == word[j]) {
                                iflag = 1
                                break
                        }
                }
                if (iflag == 0) {
                        icnt = icnt + 1
                        word[icnt] = $i
                }
        }
}

END {
        for (i=1; i<=icnt; i++) print word[i]
}
```

Test your script; your entry:

```
% awk -f awk.voc poem
In
Xanadu
did
Kubla
Khan
A
stately
pleasure
dome
decree:
Where
Alph,
the
sacred
river,
ran
Through
caverns
measureless
. . .
. . .
%
```

awk ARRAYS

- Array elements:
  - Are not declared
  - Come into existence by being mentioned
- Array subscripts
  - Are enclosed in straight braces [ ]
  - May have any non-null value, including non-numeric strings

```
x[NR] = $0
```

assigns the current input record to the *NR*th element of the array *x*

The following script prints the lines from an input file in reverse order:

```
{line[NR] = $0}
  END   { for (i = NR;i>0;i--)
          print line[i]
        }
```

Using Arrays with *awk*

Write an *awk* program that prints out any file in reverse order. Create the file *rev.file* with your favorite editor. Verify that your program works by using it on the file *poem*.

Your program looks similar to:

```
{ line[NR] = $0}
END   { for (i = NR;i>0;i--)
        print line[i]
}
```

Your entry to verify that the program works:

```
% awk -f rev.script poem
Down to a sunless sea.
My Xanadu checkline.
Through caverns measureless to man
Where Alph, the sacred river, ran
A stately pleasure dome decree:
In Xanadu did Kubla Khan
%
```

awk ARRAYS (cont.)

- Array elements may be named by non-numeric values

- The program:

```
/apple/ {x["apple"]++]
/orange/ {x["orange"]++]
END {print x["apple"],x["orange"]}
```

  increments counts for the named array elements and prints them at the end of the input

- The following script sums the values for each person; the data is stored in an associative array:

```
{sum[$ 1] += $2}
END { for (name in sum)
    print name, sum[name]
}
```

Non-Numeric Array Values

Enter the following data in a file named *people*.

Sally 300
Fred 100
Joe 60
Fred 150
Sally 50
Joe 100

Now write an *awk* script that prints each person's name and sums the values for each. Name your file *sum.script*.

Your program looks similar to:

```
{sum[$ 1] += $2}
END { for (name in sum)
        print name, sum[name]
}
```

Test the script:

```
% awk -f sum.script people
Sally 350
Fred 250
Joe 160
%
```

printf COMMAND

- The *awk* utility provides the *printf* statement for output formatting:

```
printf "format_st", expr1, . . .
```

- The program:

```
{printf "%8.2f", $1}
```

prints $ *1* as a floating-point number, 8 digits wide, with two digits after the decimal point.

- The program:

```
{printf "%10d\n", $1}
```

prints $ *1* as a 10-character decimal number, followed by a newline.

- The program:

```
{printf "%s", $2}
```

prints $ *2* as a character string.

- The program:

```
{printf "%.6g", $1}
```

prints $ *1* in the style that gives the most precision in minimum space.

The *printf* Command

For the following exercise, you need a file named *numbers* with the following contents:

```
1234567
456.7890132
908.7654
876785.4326
```

Write an *awk* script named *awk.printf* that prints the numbers in the format of a floating-point number, 6 digits wide, with 3 digits after the decimal point and prints each number on a newline. Your script:

```
{printf "%6.3f\n", $0}
```

```
% awk -f printf.script numbers
1234567.000
456.789
908.765
876785.433
%
```

STRING OPERATIONS

length(*e1*)	Returns length of *e1*; if *e1* is omitted, returns the length of current record
substr(*string,pos,length*)	Returns a substring of *string* that begins at position *pos* and is *length* characters long. If *length* is omitted, *substr* returns the suffix of *string* starting at position *pos*.

The following program folds lines into 20-character lines. A "\" is appended if the lines is folded; the residue is then processed. It is assumed that the input file does not contain tabs.

```
BEGIN {N = 20                    #fold at column 20
  for (i = 1;i <= N; i++)        #string of blanks
     blanks = blanks " "
}
{  if ((n = length($0)) <= N)
      print                      #fold is unnecessary
   else {
       for (i = 1; n > N; n -=N) {
        printf "%s\\\n", substr($0,i,N);i += N
       }
       printf "%s%s\n",substr(blanks,1,N-n),
        substr($0,i)}
}
```

String Operations

Write an *awk* script that returns the number of characters in a file; do not use the UNIX *wc* command. Name your script *char.script*; test the script on the file *poem*.

Your script looks similar to:

```
BEGIN       {nc = 0}

        {nc += length($0) + 1       #Determines number of characters
        }

END     {print "number of characters =    ",nc
        }
```

Your entry to test the script:

```
% awk -f char.script poem
number of characters =    170
%
```

STRING OPERATIONS (cont.)

`index(`*e1,e2*`)`	Returns position in *e1* which contains *e2*; returns 0 if not contained
`split(`*e1,array,sep*`)`	Splits the string *e1* into fields that are stored in *array[1]*, using *sep* as the field separator. If *sep* is omitted, the current FS is used.

This example will determine the usage count for each of the vowels contained in a section of text. The location of a first period in each line will then be determined.

```
BEGIN          {
     # string "x" contains a list of the vowels
     x = "a e i o u"
     split(x, vowels)
     # "vowels" contains usage count of each vowel
}

{ for (i=1; i<= length($0); i++) {
     if ($ i=="a") vowel["a"]++
     if ($ i=="e") vowel["e"]++
     if ($ i=="i") vowel["i"]++
     if ($ i=="o") vowel["o"]++
     if ($ i=="u") vowel["u"]++
}

{ period[NR] = index($0,".") }
```

Using *split* and *index*

Create an *awk* program called *awk.time* that allows you to pipe the output of the *date* command through a script that prints out the time—hours, minutes and seconds, each on a separate line.

Your program:

```
{
        split($4, time, ":")
        print time[1],"hours"
        print time[2],"minutes"
        print time[3],"seconds"
}
```

Verify that your program works:

```
% date | awk -f awk.time
09 hours
02 minutes
45 seconds
%
```

MATHEMATICAL OPERATIONS

`cos(`*e1*`),sin(`*e1*`)`	trig functions
`exp(`*e1*`)`	Exponential function
`log(`*e1*`)`	Natural log function
`int(`*e1*`)`	Integer function
`sqrt(`*e1*`)`	Square root function

The following *awk.script* computes the square root of the sum of the squares of a set of numbers. The set of numbers is provided as input - one number to a record.

```
BEGIN {
   sum_of_squares = 0
}

{
   sum_of_squares += $1 *$1
}
END {
   root_mean_square = sqrt(sum_of_squares/NR)
   print root_mean_square
}
```

Mathematical Operations

Write an *awk* program *awk.math* that sums a series of numbers found on one record, prints the sum, and zeroes out for the next record. Your program looks similar to:

```
BEGIN       {
        sum = 0
}

{
        for (i=1; i<=NF; i++) {
                sum = sum + $i
                if (i == NF) {
                        print sum
                        sum = 0
                }
        }
}
```

Create a *mathdata* file with this information:

```
1 2 3 4
1 1 1 1
1 100 102
2 2
```

Verify that your program works; your entry:

```
% awk -f awk.math mathdata
10
4
203
4
%
```

REDIRECTING *awk* OUTPUT

- Can be redirected from standard output to one or more output files – maximum 10 files
- Can be piped to other ConvexOX utilities

The following *awk* examples demonstrate I/O redirection:

```
% awk '{print $1 >> out}' datafile
```

appends output to end of file named by variable *out*

```
% awk '{print $1 > "file1"; \
print $2 > "file2"}' datafile
```

directs output to multiple files

```
% awk '{print $1 > $2}' datafile
```

uses contents of field 2 as the filename for the output of field 1.

```
% awk '{print $1}' file1 | mail user
% awk '{print $1 | "mail user"}' file1
```

both examples mail the first field of *file1* to the *user*

Redirecting Output

Write an *awk* program that you can enter on the command line that appends the output of the file *poem* to the file *datafile*. Verify your program.

```
% awk '{print $0 >> "datafile"}' poem
% more datafile

    PROGRAM SAMPLE

C THIS IS INCLUDED
INCLUDE PROG.FOR

* THIS IS A COMMENT

    WRITE(6,*) a, b, c
    WRITE(6,100)
    WRITE(6,200)
    WRITE(6,300)
    WRITE(6,400)
  100 FORMAT(' abc ')
  200 FORMAT(" def ")
  300 FORMAT(* zzz *)
  400 FORMAT(' ***** ')
    END


In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
My Xanadu checkline.
Down to a sunless sea.
%
```

Exercises for Chapter 4

1. What is the main difference between *sed* and *ed*?

2. Write a *sed* command which will delete all the null lines in a file.

3. Write a *sed* command which will substitute a single dot for all pairs of dots (substitute "." for "..".

4. Write a *sed* command which will change all strings ending in *.f* to *.o*.

5. What is the basic profile of an *awk* script?

6. What are the default field and record separators used in *awk* scripts?

7. How to you accomplish initialization in an *awk* script?

8. Name three commands used in controlling the program flow in an *awk* script.

9. What does the *split* string operator in an *awk* script do? What is its format?

10. Write a command that prints only the lines containing a question mark (?) in the file *clown*.

11. Write a command that changes all occurrences of the word *write* to *WRITE* in *myprog.f*. Send the corrected output to a file named *cormyprog.f*.

12. Write an *awk* or *sed* scripts (whichever is appropriate) that:

    a. Inserts **My Experiment** as the first line of the file.
    b. Replaces all occurrences of *Xanadu* with **XANADU**.
    c. Deletes the fifth line of the file.

    If you have the file *poem*, verify that you script works correctly.

13. Write an *awk* or *sed* scripts (whichever is appropriate) that:

    a. Converts uppercase to lowercase
    b. Begins comments with *c* instead of #
    c. Removes all blank (null) lines

    If you have the file *sample*, verify that your script works correctly.

14. Write a command that writes lines 1–3 of the file *poem* to a file named *short.poem*. Verify that the command works correctly.

15. Write a command that prints the current input line number of the line beginning with *My* in the file *poem*. Verify that the command works correctly.

16. Write a command that prints the number of records in the file named *poem*.

17. Write a command that prints the number of records in a file named *poem*; a colon (:) is the record separator.

18. Write a command that will print out the third and fourth field in a file named *data*.

19. Assume you have a data file that contains inventory information. Write a script (*awk* or *sed*) that prints the first field of each record (item name), followed by an amount determined by multiplying field 2 (number of units) by field 3 (our cost).

20. To complete the following exercises, you need a file named *inventory* with the following contents:

```
smith:bears:10:8
Smith:toys:5000:6
jones:table:3:1200
wilson:bats:100:10
james:batteries:1000:1
moss:books:500:12
```

Using the *inventory* file to verify the results, write individual *awk* commands that:

a. Print lines that match either *Smith* or *smith*.
b. Print lines that contain *table* as the second field.
c. Print lines that have a count (field 3) greater than 500.
d. Print the line only if *Smith* is the first field and *toys* is the second field.
e. Print out the message `The name is` followed by the name given in the first field for all the records in the file.
f. Add a column to the output that gives the running total of items on hands—10, 5010, 5013, etc.)
g. Print the line that contains both *smith* and *bears*.
h. Print the record number preceding each record.
i. Changes the record separator to an asterisk (*).

21. Write an *awk* script that gives the total value of all the items in the file *inventory*—field 3 multiplied by field 4, printing the final value only.

22. Write an *awk* script that prints the line that contains *moss* and *books*; also print out the total value for the *books* (field 3 multiplied by field 4).

23. Write a script that extracts the names and addresses from files ending in *.let* in your home directory and outputs that information to a file named *address.list*. Assume the names and address are always located on the lines 7–10.

24. Assume you have several files named in the following manner:

```
01-01t   01-02t   01-03t
02-01t   02-02t   02-03t
03-01t   03-02t   03-03t
04-01t   04-02t   04-03t
05-01t   05-02t   05-03t
```

The `filenames must be changed so that the` *t* is the last character of the name and the suffix and prefix must be interchanged. Thus, the filenames would be:

01-01t	02-01t	03-01t
01-02t	02-02t	03-02t
01-03t	02-03t	03-03t
01-04t	02-04t	03-04t
01-05t	02-05t	03-05t

Write a script that will build a new filename as described.

25. All of your file and directories have been moved from */mnt* to */usr*. Some of your files have the reference *.so /mnt/username/dirname*. The reference now needs to be *.so /usr/username/dir*. Write a script that changes the incorrect reference (*/mnt*) to the correct reference (*/usr*). The script should change only those references of *mnt* that are preceded with *.so*.

5

Inter-machine Communication

Objectives for Chapter 5

After completing Chapter 5, you will be able to:

1. Copy files between two different machine using *rcp*.
2. Transfer files to and from a remote network site using *ftp*.
3. Explain the purpose of the *ftp* commands.
4. Transfer files asynchronously between two UNIX machines using *uucp*.
5. Monitor the progress of *uucp* by means of *uulook*, *uulog*, and *uusnap*.
6. Initiate a *uucp* phone call to a neighboring site using *send*.
7. Use *rlogin* to remotely log on to another machine.

FILE TRANSFER PROGRAM

ftp(1C)

- Allows a user to transfer files to and from a remote network site (host)
- Local host: machine on which you invoke *ftp*
- Remote host: machine to which you connect using *ftp*
- Format of *ftp*

  ftp [*-v*] [*-d*] [*-i*] [*-n*] [*-g*] [*host*]

- *ftp* options which may be placed on the command line or given to the command interpreter

  -v – turns on a verbose mode
  -d – enables debugging mode
  -i – turns off interactive prompting during multiple file transfers
  -n – prevents *ftp* from attempting auto-login
  -g – disables filename globbing (processing filenames according to the *csh*(1) metacharacters)

File Transfer Program

Since the Convex machine you are currently on may not be connected to any other machine, you may not be able to complete the exercises on the following pages. To complete the exercises, use the *hostname* of you remote machine as the name of the client machine.

FILE TRANSFER PROGRAM (cont.)

- The remote *host* may either be specified on the command line or designated with the *open* command

```
% ftp system2
% ftp
ftp> open system2
```

- When the remote *host* has been specified, *ftp* attempts to establish a connection to an FTP server on that host

- If the user desires the auto-login option, the *.netrc* file must exist in his home directory. This file contains information concerning the host machines, user logins, and optionally a password to be supplied during the connection.

- *ftp* supplies the *ftp>* prompt when it is in the command interpreter mode and awaiting instructions from the user

- Command arguments which have embedded spaces may be quoted with quote (") marks

Beginning an *ftp* Session

You will initiate an *ftp* session and answer the prompts with your login name and password. Your entry:

```
% ftp convext
Connected to convext.
220 convexc FTP server (Version 4.102 Wed May 14 22:48:39 CDT 1986) ready.
Name (convext:): username
Password (convexc:username): password
331 Password required for username.
230 User username logged in.
ftp>
```

ftp COMMANDS

Commands which are recognized by *ftp*:

- Invoke shell on local machine
  ! [*command*]
  ftp> ! ls -l

- Describe the command. If no argument is given, print a list of the known commands.
  ? [*command*] or help [*command*]
  ftp> ?
  ftp> ? glob
  ftp> help glob

- Append local file to remote file
  append *local-file* [*remote-file*]
  ftp> append newfile logfile

- Set file transfer type to network ASCII (default)
  ftp> ascii

- Set file transfer type to binary
  ftp> binary

- Terminate FTP session and exit
  ftp> bye
  ftp> quit

ftp Commands

List the files in your current directory. Your entry:

```
ftp> ! ls
file1 file2 temp1 temp2
ftp>
```

Request a list of the known commands with *help*. Your entry:

```
ftp> help
Commands may be abbreviated.  Commands are:

!               dir             mget            quit            trace
append          form            mkdir           quote           type
ascii           get             mls             recv            user
bell            glob            mode            remotehelp      verbose
binary          hash            mput            rename          ?
bye             help            open            rmdir
cd              lcd             prompt          send
close           ls              sendport        status
delete          mdelete         put             struct
debug           mdir            pwd             tenex
```

Request information on the command *glob*. Your entry:

```
ftp> help glob
glob            toggle metacharacter expansion of local file names
ftp>
```

ftp COMMANDS (cont.)

- Change remote working directory
  `cd` *remote-directory*
  `ftp> cd mydir`

- Delete file on the remote machine
  `delete` *remote-file*
  `ftp> delete myfile`

- List directory contents in remote-directory and, optionally, place the output in a local-file
  `dir` [*remote-directory*] [*local-file*]
  `ftp> dir my.dir listing.file`

- Retrieve remote-file and store it on the local machine
  `get` *remote-file* [*local-file*]
  `ftp> get file working.file`

- Toggle filename globbing. When filename globbing is enabled, each local file or pathname is processed for csh(1) metacharacters
  `ftp> glob`

Using *ftp* Commands

List the directory contents in your remote-directory using the *dir* command. Your entry:

```
ftp> dir
200 PORT command okay.
150 Opening data connection for /bin/ls (100.0.0.10,1434) (0 bytes).
total 11
-rw-r--r-- 1 user   staff      413 Nov 25 08:30 .cshrc
-rw-r--r-- 1 user   staff       29 Nov 13 00:58 .exrc
-rw-r--r-- 1 user   staff      265 Dec  9 10:13 .login
-rw-r--r-- 1 user   staff        6 Nov 13 00:58 .logout
-rw-rw-r-- 1 user   staff       46 Nov 18 08:34 .mailrc
-rw-rw-r-- 1 user   staff        4 Dec 10 08:47 .msgsrc
-rw-r--r-- 1 user   staff       18 Nov 13 00:58 .project
-rw-rw-r-- 1 user   staff       29 Dec 10 09:10 file1
-rw-rw-r-- 1 user   staff       29 Dec 10 09:10 file2
-rw-rw-r-- 1 user   staff       29 Dec 10 09:15 temp1
-rw-rw-r-- 1 user   staff       29 Dec 10 09:11 temp2
226 Transfer complete.
2043 bytes received in 2.83 seconds (.705 Kbytes/s)
ftp>
```

Delete *temp1* on the remote machine; then verify the file is gone by excuting another *dir* command. Your entry:

```
ftp> delete temp1
200 DELE command okay.
ftp> dir
200 PORT command okay.
150 Opening data connection for /bin/ls (100.0.0.10,1434) (0 bytes).
total 10
-rw-r--r-- 1 user   staff      413 Nov 25 08:30 .cshrc

-rw-rw-r-- 1 user   staff       29 Dec 10 09:10 file1
-rw-rw-r-- 1 user   staff       29 Dec 10 09:10 file2
-rw-rw-r-- 1 user   staff       29 Dec 10 09:11 temp2
226 Transfer complete.
2043 bytes received in 2.83 seconds (.705 Kbytes/s)
ftp>
```

Toggle the filename globbing option twice (turning it off and then back on again). Your entry:

```
ftp> glob
Globbing off.
ftp> glob
Globbing on.
ftp>
```

ftp COMMANDS (cont.)

- Change local working directory

  `lcd` [*directory*]

  ftp> `lcd source.dir`

- List of the contents of a remote directory; places the results in a local file

  `ls` [*remote-directory*] [*local-file*]

  ftp> `ls`

  ftp> `ls orig.dir listing.file`

- Delete the specified remote files. If globbing is enabled, expand the filenames.

  `mdelete` *remote-files*

  ftp> `mdelete file1 file2`

- Obtain a directory listing of multiple files on the remote machine and place the results in a local-file

  `mdir` *remote-files local-file*

  ftp> `mdir source.dir listing.file`

- Retrieve the specified files from the remote machine and place them in the current local directory. If globbing is enabled, expand the filenames.

  `mget` *remote-files*

  ftp> `mget file1 file2`

Manipulating Directories Using *ftp* Commands

List the contents of a remote directory using the *ls* command. Your entry:

```
ftp> ls
200 PORT command okay.
150 Opening data connection for /bin/ls (100.0.0.10,1436) (0 bytes).
.cshrc
.exrc
.login
.logout
.mailrc
.msgsrc
.project
file1
file2
temp2
226 Transfer complete.
298 bytes received in 0.20 seconds (1.46 Kbytes/s)
ftp>
```

ftp COMMANDS (cont.)

- Make a directory on the remote machine
  `mkdir` *directory-name*
  `ftp> mkdir working.dir`

- Obtain a listing of multiple files on the remote machine and place the result in a local-file
  `mls` *remote-files local-file*
  `ftp> mls file1 file2 list.file`

- Transfer multiple local files from the current local directory to the current working directory on the remote machine
  `mput` *local-files*
  `ftp> mput file1 file2`

- Toggle interactive prompting. Interactive prompting occurs during multiple file transfers to allow the user to selectively retrieve or store files.
  `ftp> prompt`

Transferring Files Using *ftp* Commands

Make a new directory named *newdir* on the remote machine using the *mkdir* command. Verify the new directory exists by executing *ls*. Your entry:

```
ftp> mkdir newdir
200 MKDIR command okay.
ftp> ls
200 PORT command okay.
150 Opening data connection for /bin/ls (100.0.0.10,1473) (0 bytes).
.cshrc
.exrc
.login
.logout
.mailrc
.msgsrc
.project
file1
file2
newdir
temp2
226 Transfer complete.
317 bytes received in 0.52 seconds (.595 Kbytes/s)
ftp>
```

Toggle interactive prompting twice. Your entry:

```
ftp> prompt
Interactive mode off.
ftp> prompt
Interactive mode on.
ftp>
```

ftp COMMANDS (cont.)

- Store a local file on the remote machine
  put *local-file* [*remote-file*]
  ftp> put myfile mycopy

- Print the name of the current working directory on the remote machine
  ftp> pwd

- Rename a remote file
  rename [*from*] [*to*]
  ftp> rename temp.file program.f

- Delete a directory on the remote machine
  rmdir *directory-name*
  ftp> rmdir working.dir

- Toggle verbose mode; default is *on*
  ftp> verbose

- Terminate an *ftp* session
  ftp> bye

Using *ftp* Options

Print your current working directory on the remote machine. Your entry:

```
ftp> pwd
251 "/mnt/user" is current directory.
ftp>
```

Show the current status of your *ftp* session. Your entry:

```
ftp> status
Connected to convext.
Mode: stream; Type: ascii; Form: non-print; Structure: file
Verbose: on; Bell: off; Prompting: on; Globbing: on
Hash mark printing: off; Use of PORT cmds: on
ftp>
```

Terminate your *ftp* session with the *bye* command. Your entry:

```
ftp> bye
221 Goodbye.
%
```

UNIX TO UNIX COPY

uucp(1C)

- Asynchronous copy of files between UNIX machines
- Format of *uucp*

  uucp [*options*] *source-file destination-file*

- *uucp* options:
  - -d – make all necessary directories for the file copy
  - -c – copy from the source file rather than making an intermediate copy in the spool directory
  - -C – make a copy before sending a copy this is the default
  - -m – send mail to requester when the copy is complete; only works when sending or receiving single files
- Both the source and destination machines may be the user's current host machine
- Files received by *uucp* will be owned by *uucp*
- *uucp* preserves execute permissions across the transmission and gives *0666* read and write permissions

UNIX to UNIX Copy

You will be using the files generated in the *ftp* section. Please attempt to copy *file1* to *temp1* in your current directory by means of *uucp*. Your entry:

```
% uucp file1 temp1
permission denied
uucp failed. code 1
%
```

The command probably failed because *uucp* did not have write permission in your directory, or read access on the source file. Remember, the new file would be owned by *uucp*.

uucp FILENAMES

- Shell metacharacters *?*[]* appearing in a path name will be expanded on the appropriate machine

- Types of filenames:
  - Local system
    ```
    % uucp myfile /tmp/myfile
    ```
  - Remote system
    system_name\!*pathname*
    ```
    % uucp myfile\
     c1\!/usr/spool/uucppublic/myfile
    ```

- Pathname may be one of the following:
  - Full pathname
    ```
    % /mnt/user/filename
    ```
  - Path name preceded by *~user*
    ~user/filename
    ```
    % uucp file1 c1\!~user/new1
    ```
  - Partial pathname
    ```
    % uucp file 1 c1\!my.dir/filename
    ```
  - Directory used as a destination filename; the source file name is used as the destination filename
    ```
    % uucp file1 c1\!~uucp
    ```

uucp Filenames

Using *uucp*, place a copy of *file1* into the */tmp* directory. Verify that the copy occurred by listing the contents of the */tmp* directory. Your entry:

```
% uucp file1 /tmp/temp1.$user
% ls -l /tmp/temp1.$user
-rw-r----- 1 uucp       29 Dec 10 13:33 /tmp/temp1.user
%
```

Using *uucp*, place a copy of *file1* into the ~*uucp* directory. Name the new copy $*user.1* so that it will have your login name. Remember to escape the ~ character with a backslash. Verify that the copy occurred by listing the contents of the ~*uucp* directory. Your entry:

```
% uucp file1 ~uucp/$user.1
% ls -l ~uucp/$user*
-rw-r----- 1 uucp       29 Dec 10 13:40 /usr/spool/uucppublic/user.1
%
```

Copy *file2* to $*user.2*. Then using *uucp*, place a copy of *file2* into the ~*uucp* directory. When designating the destination filename, use the local machine name as the remote host name. Remember to escape the *!* character with a backslash. When designating the destination name, supply only a directory name. Verify that the copy occurred by listing the contents of the ~*uucp* directory. Your entry:

```
% cp file2 $user.2
% uucp $user.2 convext\!~uucp
% ls -l ~uucp/$user*
-rw-r----- 1 uucp       29 Dec 10 13:40 /usr/spool/uucppublic/user.1
-rw-r----- 1 uucp       29 Dec 10 13:50 /usr/spool/uucppublic/user.2
%
```

MONITORING *uucp*

uulook(1)

- Monitors the inter-system communications of *uucp*
- Displays the end of the *uucp* log found in */usr/spool/uucp/LOGFILE*
- Format of *uulook*

```
% uulook
```

uulog(1)

- Maintains a summary log of *uucp* transactions
- Format of *uulog*

uulog *options*

- *uulog* options (select at least one of the following)

-u *user* – print information about work done for the specified *user*

-s *sys* – print information about work involving system *sys*

```
% uulog -u user -s convex1
```

Monitoring *uucp*

Using *uulook*, monitor the *uucp* system. Your entry:

```
% uulook
uucp convext (12/10-14:15-13683) SUCCEEDED (call to convex )
uucp convext (12/10-14:15-13683) OK (startup)
uucp convext (12/10-14:15-13683) OK (conversation complete)
uucp convext (12/10-15:15-17274) SUCCEEDED (call to convex )
uucp convext (12/10-15:15-17274) OK (startup)
uucp convext (12/10-15:15-17274) OK (conversation complete)
user  (12/10-15:15-17298) DONE (WORK HERE)
user  (12/10-15:15-17319) DONE (WORK HERE)
user convext (12/10-15:16-17362) DONE (WORK HERE)
%
```

Using *uulog*, print information concerning the work which has been done on your behalf. Your entry:

```
% uulog -u $user
user cli-c1 (12/5-11:26-5393) QUE'D (C.cli-c1n0pE1)
user  (12/5-13:52-12627) DONE (WORK HERE)
user  (12/10-13:31-11614) DONE (WORK HERE)
user  (12/10-13:33-11739) DONE (WORK HERE)
user convext (12/10-13:40-12074) DONE (WORK HERE)
user  (12/10-15:15-17298) DONE (WORK HERE)
user  (12/10-15:15-17319) DONE (WORK HERE)
user convext (12/10-15:16-17362) DONE (WORK HERE)
%
```

MONITORING *uucp* (cont.)

uusnap(8C)

- Shows a snapshot of the *uucp*system
- Format of *uusnap*

  % uusnap
- *Uusnap* displays in tabular format a synopsis of the current *uucp* situation
- The format of the output is as follows:

  site N Cmds N Data N Xqts Message
- Where

site	name of the site with work
N Cmds	count of the commands to be executed
N Data	amount of data to be transmitted
N Xqts	count of the requested remote executions
Message	current status message for a site

Using the *uusnap* Command

Acquire a snapshot of the *uucp* system by means of *uusnap*. Your entry:

```
% uusnap
cli-c1   1 Cmd   1 Data   ---  DIAL FAILED  Retry time reached
nrao1    1 Cmd   1 Data   ---
systech   ---     ---     ---  DIAL FAILED  Retry time reached
%
```

INITIATE A *uucp* PHONE CALL

send(1)

- Initiates a *uucp* phone call to a neighboring site
- Format of *send*

  send *site*

- Starts up *uucico* which initiates the call
- The site must exist in the local *L.sys* information file
- Anyone may initiate an outgoing call
- Transfers files *spooled* in the */usr/spool/uucp* directory

send

Initiate a *uucp* phone call to your local machine. Your entry:

```
% send convext
%
```

REMOTE LOGIN

rlogin(1C)

- Connects a user's terminal on the current local host system to a remote host system

- Format of *rlogin*

```
rlogin rhost [-lusername]
rlogin system1
```

- To designate the login name desired on the remote system if it differs from the login name on the local machine

```
rlogin system1 -lsmith
```

- The user requests to disconnect from the remote system by any of the following commands:

  % *CTRL* D

  % logout

  % exit (if using *csh*)

  % ~.

- The *rlogin* session may be suspended by typing ~CTRL-z. The session may then be restarted like any other suspended job.

Remote Login

Login onto your remote machine remotely using *rlogin hostname*. (Since the CONVEX machine you are currently on may not be connected to any other machine, you can use the *hostname* of your local machine as the name of the remote machine). Then execute the following commands on the remote machine: *ls*, *uptime*, and *date*. Your entry:

```
% rlogin convext
Password: password
Last login: Wed Dec 10 08:44:44 on tty0d
You have old mail.
There are new messages.
% ls
file1          file2          temp2
% uptime
 3:57pm up 2 days,  8:01,  27 users,  load average: 1.75, 1.53, 1.70
% date
Wed Dec 10 15:57:34 CST 1986
%
```

Suspend your *rlogin* session by typing ~*CTRL-z*. Verify that you have a suspended job by executing *jobs*. Restart the suspended job with the *fg* (foreground) command. Your entry:

```
% ~^Z

Stopped
% jobs
[1] + Stopped          rlogin convext
% fg %1
rlogin convext
%
```

Terminate your *rlogin* session by typing ~.. Your entry:

```
% ~.
Closed connection.
%
```

REMOTE SYSTEM ACCESS

- To allow access to remote systems without password checking, each user can create a *.rhosts* file in the home directory

- When a user attempts an *rlogin*, his/her home directory is checked for the *.rhosts* file. If the remote system name is in the file, there is no password checking.

- The format of the file:

```
hostname
```

  If a user is logging on to *system2* from *system1* and the user has an *.rhosts* file (on system2) with the entry `system1`, the user will not be prompted for a password.

- An alternative to the *.rhosts* file is creating an */etc/hosts.equiv* file containing a list of all *trusted* systems. (This file can be created only by the system administrator.)

  The *hosts.equiv* file would allow access without password checking for all systems listed in this file. (Users would need accounts on the appropriate systems.)

Creating the *.rhosts* File

Assume that you have accounts on your local system (*systema*), and remote systems - *systemb, systemc,* and *systemd.* Create an *.rhosts* file(s) on the appropriate system(s) to allow access between your local system and *systemb* without password validation.

Your entry on *systema*:

```
% vi .rhosts
systemb
%
```

Your entry on *systemb*:

```
% vi .rhosts
systema
%
```

REMOTE FILE COPY

rcp(1C)

- Copies files between machines
- Either the *~/.rhosts* file or the */etc/hosts.equiv* file must exist for remote copy to be successful. Users must have an account on each system.
- Format of *rcp*

  `rcp` *local_file1 remote_system:file2*
  `rcp [-r]` *file1 ... remote_system:directory*

- Each *file* or *directory* argument is one of two forms:
  - Local filename which does not contain any colon characters

    ```
    rcp oldfile newfile
    rcp file1 file2 file3 new.dir
    ```

  - Remote file name of the form *rhost:path*

    ```
    rcp localfile convex0:newfile
    rcp f1 f2 f3 convex0:new.dir
    rcp convex0:file1 local.file
    ```

- Both the source and destination machines may be different than the local machine

  ```
  rcp c0:file1 c1:newfile
  ```

Remote File Copy

For the following exercises, you need two files: *file1* and *file2* on the local machine. If you do not have these two files, create them with your favorite UNIX text editor. These files may contain any text you wish.

To determine the name of your local machine, type the following: **hostname**. Your entry:

```
% hostname
convext
%
```

Copy *file1* to *remote_mach:temp1* with *rcp*. Do use a *remote host* (that exists in your environment) for this exercise.

```
% ls
% file1 file2
% rcp file1 mach1:temp1
%
```

Note: If your current machine is not connected to another machine, you may not be able to complete some of the exercises contained in this module.

REMOTE FILE COPY (cont.)

- If the *-r* option is used and any of the source files are directories, *rcp* copies each subtree rooted at that name. The destination must be a directory.

```
rcp -r my.dir convex0:new.dir
rcp -r c0:orig.dir /mnt/usr/mydir
```

- If *path* is not a full pathname, it is interpreted relative to your login directory on *rhost*.

Using Remote Copy

Copy *temp1* from the remote machine (mach1) to *temp2* on the local machine with *rcp*. Your entry:

```
% ls
% file1 file2
% rcp mach1:file1 temp2
% ls
% file1 file2 temp2
%
```

Exercises for Chapter 5

1. How may you determine the hostname of the machine you are currently working on?

2. What character or characters separate the *hostname* from the filename in the *rcp* command line?

3. What two commands may be typed when executing *ftp* in order to obtain help?

4. What does the *glob* command in *ftp* actually do?

5. What is the difference between *put* and *mput* in *ftp*?

6. Who owns all files copied by the *uucp* utility?

7. When using *csh*, what is the character which separates the hostname from the filename in a *uucp* command?

8. Name three utilities which can be used to monitor the activity of *uucp*.

9. When using *csh*, how do you suspend and terminate an *rlogin* session?

10. Using *ftp*:

    a. Create a directory on the remote machine named *newdir*.
    b. Transfer two files from the local machine to the remote directory, *newdir*.
    c. Transfer these two files back to the local machine but name them *new1* and *new2*.

11. Set up the file that will allow you to remote copy (*rcp*) from your local machine to the remote machine and from the remote machine to the local machine.

    a. First verify that you can do remote logins from each machine without being prompted for your password.
    b. Copy one file (your choice) from the remote machine to the local machine; then reverse the procedure, copy one file (your choice) from the local machine to the remote machine.
    c. Verify the transfer took place by using *rsh* to execute the *lf* command on the appropriate machine.

12. Transfer one file (your choice) using *uucp* to the remote machine. Your file should exist in */usr/spool/uucppublic*. Transfer the file to the remote directory */usr/spool/uucppublic*.

6

Using CXbatch

Objectives for Chapter 6

After completing Chapter 6, you will be able to:

1. Submit jobs to a specific queue.
2. Run a job at a stated time.
3. Set up the appropriate files required to use CXbatch on remote systems.
4. Delete a request from a specific queue.
5. Move jobs from one queue to another queue.
6. Redirect standard output.
7. Specify resources required for a request.

CXBATCH SYSTEM OVERVIEW

- The batch system allows processes to be queued for execution.
- There are advantages to using the batch system for long, time-consuming tasks:
  - A CPU will not be overloaded with several of these processes running at one time.
  - Tasks can be assigned a higher or lower priority automatically.
  - Resources can be defined on a queue-by-queue basis.
- Processes are placed on the batch queue voluntarily.

Overview

The purpose of CXbatch is to get the best job turn-around time possible and still provide optimum performance for interactive users. To help make efficient use of your system's resources, use CXbatch.

Depending upon the batch system configuration and systems available, using CXbatch allows you to submit jobs on either the local or remote system. Ideally, CXbatch will be configured to give equal CPU usage to all queues.

MODIFY USER FILES

- CXbatch reads the user's login shell when jobs are executed.

- If the submitter has interactive commands in the *.login, .cshrc, .profile,* or *.logout* files, these commands will be executed during a batch job.

- To prevent interactive commands from be run during batch jobs, submitters can add the following test to their files:

```
if (! $?ENVIRONMENT) then
    ...
    interactive commands
endif
```

  Another format:

```
if ($?ENVIRONMENT) exit
```

- For remote submission:
  - Each user needs an account on each machine executing batch requests.
  - Each user must have a *.rhosts* file in the home directory on each machine executing batch requests.

Modifying User Files

Create an entry at the beginning of your *.login*, *.cshrc* and *.logout* files to eliminate interactive commands from being executed when a CXbatch request is run. Your entry:

```
% vi .login
if ($?ENVIRONMENT) exit
% vi .cshrc
if ($?ENVIRONMENT) exit
% vi .logout
if ($?ENVIRONMENT) exit
%
```

BATCH REQUESTS

- Jobs are submitted to the batch queue system with the *qsub* command. The default configuration provides three queues: `short`, `long`, `verylong`.

- Users can submit batch requests with interactive commands or a script file.

- *qsub* has the format:

  qsub *options cmdfile*

  or

  qsub *options*
  commands
  ^D

  cmdfile is the name of the file containing the commands to run. Commands are read from standard input if *file* is not listed.

  Example command line entry:

```
% qsub
fc my.prog.f
a.out
^D
%
```

  The request will be submitted to the default queue, `long`.

Batch Requests

Submit a request to the default queue that will list the files in your current directory and display the date. Your entry:

```
% qsub
lf
date
Request 9.convext submitted to queue: long.
%
```

BATCH EXECUTION OUTPUT

- Any files created during batch execution are written in the current directory unless directed elsewhere.

- Unless redirected, output from the batch execution is sent to *STDIN.o* and error messages are sent to *STDIN.e*. These files are written to the current working directory.

  When the *STDIN.o* file is created, it will contain the message *Warning: no access to tty; thus no job control in this shell, logout* with the batch execution output. To eliminate this message, the submitter must redirect output to another file.

```
% qsub
fc my.prog.f
a.out > out.exe
^D
%
```

  The output from *a.out* will be placed in *out.exe*. All other output will be placed in *STDIN.o*; error messages will be placed in *STDIN.e*.

Batch Output

Submit a request to the default batch queue that will display the message: `testing batch output; testing complete`. Do not redirect the output. Your entry:

```
% qsub
echo "Testing batch output"
echo "Testing complete"
Request 10.convext submitted to queue: long.
%
```

View the contents of the files *STDIN.o* and the *STDIN.e* files. You will see the output of your command and also the *Warning* message. To view the files:

```
% cat STDIN.o10 STDIN.e10
Warning: no access to tty; thus no job control in this shell...
Testing batch output
Testing complete
%
```

The *STDIN.e* file should be empty; *STDIN.o* contains the output and warning message.

Now submit the same job but redirect the output to the file *mes*. View the *STDIN.o* and the *mes* file. Your entry:

```
% qsub
echo "Testing batch output" > mes
echo "Testing complete" >> mes
Request 11.convext submitted to queue: long.
% cat mes
Testing batch output
Testing complete
% cat STDIN.o11
Warning: no access to tty; thus no job control in this shell...
logout
%
```

CXBATCH SUBMITTER OPTIONS

- Users have several options that can be specified to control the environment in which the request is submitted and executed:
  - time to run the request
  - import or no import of current directory
  - intra-queue priority and *nice* value
  - resource requirements
  - mail notification
  - queue
  - redirect standard output

- Sample options:

a	Time to run request.
i	Import current directory
p	Set intra-queue priority
q	Specify queue to which request is submitted
me	Send mail when job has completed

- The options can be entered on the command line with the *qsub* command or included in a script file that is submitted as a batch request.

```
% qsub -q short
grep DATA1 /mnt/snoopy/*
^D
```

Submitting to Specific Batch Queues

Submit a request to the *short* batch queue that will display the message: `testing the short queue`. Do not redirect the output. Your entry:

```
% qsub -q short
echo "Testing the short queue"
Request 12.convext submitted to queue: short.
%
```

View the *STDIN.o* file to verify that the command executed.

SUBMITTING BATCH REQUESTS

- An example sending mail to the submitter upon batch execution completion, using the *slow* queue:

```
qsub -me -q slow
find / -name "*.c" -user smith -print > finderfile
^D
```

- Jobs can be submitted with a script:

```
#
# Example C shell batch request script
# Command line options can be placed in a script
# Use @$ immediately preceding the options
#
# Send the user mail upon completion
# Submit the job to the slow queue
#
# @$-me -q slow
#
# To run the script at 11:00 this evening:
#
# @$-a "11:00pm CDT"
find / -name "*.c" -user smith -print > finderfile
#
```

Submit the job:

```
qsub myjob
```

- A copy of the script is kept in the file */usr/spool/nqs/scripts*.

- The submitter may use *qjlist* to examine a job's script.

```
qjlist request_ID
```

Submitting a Command File

Write a C shell batch request script named *hunt90* that will execute at 2:15 p.m. This script should find all your files that have not been modified for more than 90 days. The list of files should be written to the file *90.day*. Submit the job to the verylong batch queue. After submitting the job, examine your job's script. Your entry:

```
% vi hunt90
#
#@$ -q verylong
#@$-a "2:15pm CDT"
find /mnt/smith -mtime +90 -print > /mnt/smith/90.day
#
% qsub hunt90
Request 12.convext submitted to queue: verylong.
% qjlist 12
#
#@$ -q verylong
#@$-a "2:15pm CDT"
find ~ -mtime +90 -print > ~/90.day
%
```

DISPLAYING QUEUE INFORMATION

- The *qstat* command examines the queue area and displays the status of submitted jobs.

- The *qstat* command format is:

  `qstat` *[options] [queuename]*

  When *qstat* is entered without options, a listing of all the available queues is displayed.

- Some of the available options are:

`x` *queue*	Lists the contents of *queue* and the resource limits.
`a`	Shows all requests; by default, only the requests for the user are shown.
`l`	Shows long form of information for a queued request.
`m`	Displays the date and time if the request was submitted with the *-a* option.

Listing Available CXbatch Queues

Determine the queues available on your system. Your entry:

```
% qstat
Queue for short jobs.
short@convext: type=BATCH: [ENABLED, INACTIVE]: pri=48
aliases: s, short_queue, S, SHORT
 0 exit:  0 run:  0 stage:  0 queued:  0 wait:  0 hold:  0 arrive:

Queue for long jobs.
long@convext: type=BATCH: [ENABLED, INACTIVE]: pri=32
aliases: l, long_queue, L, LONG
 0 exit:  0 run:  0 stage:  0 queued:  0 wait:  0 hold:  0 arrive:

Queue for very long jobs.
verylong@convext: type=BATCH: [ENABLED, INACTIVE]: pri=16
aliases: v, verylong_queue, V, VERYLONG
 0 exit:  0 run:  0 stage:  0 queued:  0 wait:  0 hold:  0 arrive:
%
```

USING THE *qstat* COMMAND

- To display the status of the *short* queue:

```
qstat short
short@convext; type=BATCH; [ENABLED, INACTIVE]; pri=30
 0 exit;  0 run;  0 stage;  0 queued;  0 wait;  0 hold;  0 arrive;
   REQUEST NAME   REQUEST ID  USER   PRI   STATE
PGRP <2 requests RUNNING>
 3:   STDIN              127.convext  snoopy   24  QUEUED
```

Only the submitter's jobs are shown.

- To display the resources available for a queue:

```
qstat -x short
short@convext; type=BATCH; [ENABLED, INACTIVE]; pri=30
 0 exit;  0 run;  0 stage;  0 queued;  0 wait;  0 hold;  0 arrive;
 Run_limit = 2;
 Accounting: On
 Activity ID offset: 0
 Cumulative system space time = 48.890000 seconds
 Cumulative user space time = 16.240000 seconds
 Unrestricted access
 Import directory: Yes
 Per-process core file size limit = UNLIMITED <DEFAULT>
 Per-process data size limit = UNLIMITED <DEFAULT>
 Per-process permanent file size limit = UNLIMITED <DEFAULT>
 Per-process execution nice value = 0 <DEFAULT>
 Per-process stack size limit = UNLIMITED <DEFAULT>
 Per-process CPU time limit = 36000.0 <DEFAULT>
 Per-process working set limit = 10 megabytes <DEFAULT>
   REQUEST NAME  REQUEST ID  USER  PRI  STATE  PGRP
 1:  STDIN       128.convext       snoopy  24 RUNNING 28518
```

Displaying Queue Resources

Determine the resources available for the long queue. Your entry:

```
% qstat -x
Queue for long jobs.
long@convext; type=BATCH; [ENABLED, INACTIVE]; pri=32
aliases: l, long_queue, L, LONG
 0 exit; 0 run; 0 stage; 0 queued; 0 wait; 0 hold; 0 arrive;
 Run_limit = 1;
 Accounting: Off
 Activity ID offset: 0
 Cumulative system space time = 19.070000 seconds
 Cumulative user space time = 9.510000 seconds
 Unrestricted access
 Import directory: Yes
 Per-process core file size limit = UNLIMITED
 Per-process data size limit = UNLIMITED
 Per-process permanent file size limit = UNLIMITED
 Per-process execution nice value = 0
 Per-process stack size limit = UNLIMITED
 Per-process CPU time limit = 600.0
 Per-process working set limit = UNLIMITED
%
```

USING THE *qstat* COMMAND (cont.)

- To display all requests in a queue:

```
qstat -a short
```

```
short@convext; type=BATCH; [ENABLED, RUNNING]; pri=30
0 exit;  2 run;  0 stage;  1 queued;  0 wait;  0 hold;  0 arrive;
 REQUEST NAME   REQUEST ID     USER    PRI   STATE     PGRP
1:   STDIN      125.convext    mentor  24  RUNNING    28153
2:   STDIN      126.convext    mentor  24  RUNNING    28160
3:   STDIN      127.convext    snoopy  24  QUEUED
```

Determining all Requests in a Specific Queue

Assume that you executed the command `qstat short`. The only jobs listed were you own. However, you know that other jobs are pending in addition to your own. Display all the requests for the *short* queue. Your entry:

```
% qstat -a short
short@convext; type=BATCH; [ENABLED, RUNNING]; pri=30
0 exit;  2 run;  0 stage;  1 queued;  0 wait;  0 hold;  0 arrive;

 REQUEST NAME  REQUEST ID  USER    PRI  STATE    PGRP
    1:    STDIN   13.convext   mentor  24  RUNNING  28153
    2:    STDIN   14.convext   mentor  24  RUNNING  28160
    3:    STDIN   15.convext   snoopy  24  QUEUED
```

REMOVING QUEUED JOBS

- **Jobs can be removed from the queue with the *qdel* command.**

  The *qdel* command format for a submitter to remove his/her own request:

  `qdel` *request_ID*
  `qdel 127`

- **Users can *remove*** only their own requests.

  To remove one of the jobs belonging to user *snoopy*:

  `# qdel -u mentor 126`

- **To kill executing jobs, use the *-k* option.**
  `qdel -k` *request_ID*

- **To kill the process leader, the submitter may use:**
  `qdel -1` *PGRP* **or** `kill -HUP` *-PGRP*

Removing Queued Requests

Submit a job to the long queue; then attempt to remove the running job. What happened? Can you kill the job? Your entry:

```
% qsub
sleep 50
echo "testing the qdel and kill commands"
^D
Request 16.convext submitted to queue: long.
% qdel 16
Request 16 is running.
% qdel -k 16
Request 16 is running, and has been signaled.
%
```

MOVING JOBS

- Submitters can move their jobs from one queue to another with the *qmgr move my_request* command.

- The job cannot be moved if the request exceeds queue limits, access restrictions, or other attributes of the new queue.

- The format for the *move* command:

  `/usr/convex/qmgr move my_request` *`reqst_id quename`*

  `/usr/convex/qmgr move my_request 127 long`

Moving Queued Jobs

Submit several jobs to the *short* queue. Select one of the jobs and move it to the long queue. Remember running jobs cannot be moved. Your entry:

```
% qsub mycmds
Request 17.convext submitted to queue: short.
% qsub anothercmds
Request 18.convext submitted to queue: short
% qsub onemorecmds
Request 19.convext submitted to queue: short.
% /usr/convex/qmgr move my_request 17 long
CXbatch manager[TCML_REQMOVSTA ]: Request is not in
an allowable state at local host.
% /usr/convex/qmgr move my_request 19 long
CXbatch manager[TCML_COMPLETE ]: Transaction complete at local host.
%
```

HOLDING BATCH REQUESTS

- Submitters can prevent their queued jobs from starting execution with the *qmgr hold request* command.

- The format to hold a job:

  /usr/convex/qmgr hold request *request_id*

  /usr/convex/qmgr hold request 127

- To release the job:

  /usr/convex/qmgr release request *request_id*

  /usr/convex/qmgr release request 127

Holding Requests

Submit several jobs to a queue of your choice. Then select one job to put on hold. View the statistics for the queue to verify that you have put the job on hold. Now release the job. Verify that you have released the job. Your entry:

```
% qsub mycmds
Request 20.convext submitted to queue: long.
% qsub anothercmds
Request 21.convext submitted to queue: long.
% qsub onemorecmds
Request 22.convext submitted to queue: long.
% /usr/convex/qmgr hold request 21
CXbatch manager[TCML_COMPLETE ]: Transaction complete at local host.
% qstat long
Queue for long jobs.
long@convext; type=BATCH; [ENABLED, RUNNING]; pri=32
aliases: l, long_queue, L, LONG
 0 exit;  1 run;  0 stage;  1 queued;  0 wait;  1 hold;  0 arrive;

  REQUEST NAME   REQUEST ID  USER  PRI   STATE   PGRP
 1:   STDIN     20.convext  mentor 31 RUNNING  15558
 2:   STDIN     22.convext  mentor 31  QUEUED
 3:   STDIN     21.convext  mentor 31 HOLDING
% /usr/convex/qmgr release request 21
CXbatch manager[TCML_COMPLETE ]: Transaction complete at local host.
% qstat long
Queue for long jobs.
long@convext; type=BATCH; [ENABLED, RUNNING]; pri=32
aliases: l, long_queue, L, LONG
 0 exit;  1 run;  0 stage;  1 queued;  0 wait;  1 hold;  0 arrive;

  REQUEST NAME   REQUEST ID  USER  PRI   STATE   PGRP
 1:   STDIN     21.convext  mentor 31 RUNNING
 2:   STDIN     22.convext  mentor 31  QUEUED
```

Exercises for Chapter 6

1. Write a command that submits the command file *progs* to the *verylong* queue.

2. Write a command that displays the resources available for the *programs* queue.

3. You notice that interactive commands from your *.cshrc* file and the word *logout* are appearing in your CXbatch output. Explain how you would correct this problem.

4. You want to submit a job to execute on a remote machine. What files must exist for you to be able to submit requests to a remote machine and get the output on your local machine?

5. If you do not specify a queue with the *qsub* command, what happens to your request?

6. Write a command that shows the contents of the command files you have placed in the batch queue; the identification number is 24.

7. What CXbatch queues are available on your local system?

8. How many jobs are currently queued for the queue *short*?

9. One of your colleagues has submitted a request to the short queue. Where will the output be placed?

10. Write a command that will submit a request to the *verylong* queue and send you mail upon completion.

11. Write a command that will kill an executing batch job, number 50.

7

Magnetic Tape Support

Objectives for Chapter 7

After completing Chapter 7, you will be able to:

1. Identify the UNIX system names for magnetic tape drives.
2. Determine the densities which are supported by the UNIX system.
3. Allocate a specific tape drive for your exclusive use.
4. Deallocate a tape drive.
5. Control the tape reel movement.
6. Read and write tape archive files using the *tar* utility.
7. Read and write ANSI labeled multifile tapes by means of the *ansitar* utility.

UNIX MAGNETIC TAPE SUPPORT

- Three tape densities are supported: 800bpi, 1600bpi, and 6250bpi
- When a file opened for writing is closed, two end-of-file marks are written
- If the tape is not rewound, it is positioned with the read/write head between the two tapemarks
- A standard UNIX tape consists of a series of 1024-byte records terminated by an end-of-file
- The UNIX system makes it possible for a tape to be treated like any other file
- When referencing tape as ordinary files, the *block* interface is used (*mt*)
- When dealing with foreign tapes or very long records, the *raw* interface is used (*rmt*)

UNIX Magnetic Tape Support

For the exercises in this module, you will need two files: *file1* and *file2*. Generate *file1* by copying the *csh* manual page; generate *file2* by copying the *dd* manual page. The manual pages may be found in the */usr/man/man1* directory. Your entry:

```
% cp /usr/man/man1/csh.1 file1
% cp /usr/man/man1/dd.1 file2
%
```

MAGNETIC TAPE UNITS

First Four Tape Units

Density	Rewind?	unit0	unit1	unit2	unit3
800 bpi	rewind	mt0	mt1	mt2	mt3
800 bpi	no rewind	mt4	mt5	mt6	mt7
1600 bpi	rewind	mt8	mt9	mt10	mt11
1600 bpi	no rewind	mt12	mt13	mt14	mt15
6250 bpi	rewind	mt16	mt17	mt18	mt19
6250 bpi	no rewind	mt20	mt21	mt22	mt23

Second Four Tape Units

Density	Rewind?	unit0	unit1	unit2	unit3
800 bpi	rewind	mt24	mt25	mt26	mt27
800 bpi	no rewind	mt28	mt29	mt30	mt31
1600 bpi	rewind	mt32	mt33	mt34	mt35
1600 bpi	no rewind	mt36	mt37	mt38	mt39
6250 bpi	rewind	mt40	mt41	mt42	mt43
6250 bpi	no rewind	mt44	mt45	mt46	mt47

For every *mt* unit, there is a corresponding *rmt* unit

UNIX Magnetic Tape Units

You can find the UNIX naming conventions for the magnetic tape units in the *mtio*(4) manual page. Read this manual page. Your entry:

```
% man mtio

MTIO(4)          UNIX Programmer's Manual          MTIO(4)

NAME
   mtio - UNIX magtape interface

. . .

CONVEX Release          20 May 1986

%
```

TAPE DRIVE ALLOCATION

tpalloc(1)

- Used to allocate specific tape drives
- Format of *tpalloc*:

  tpalloc [*options*] [-f *device*] [*name*]

- *tpalloc* options:

-b	Request a block mode tape unit % `tpalloc -b mytape`
-d *bpi*	Specify the density of the tape unit (default 1600 bpi) % `tpalloc -d 6250 mytape`
-f *device*	A specific tape unit (*mt*n or *rmt*n) % `tpalloc -f /dev/rmt16`
-i *ips*	Speed of the tape unit in inches per second % `tpalloc -i 125 mytape`

Tape Drive Allocation

You are probably working on a CONVEX machine which has only one tape drive. Therefore, you must allocate the drive, use it promptly, and then release the drive so that others may use the tape drive. If it is inconvenient for you to actually work with the tape drive at this time, you will still be able to perform several of the exercises in this section.

Allocate the tape drive, using the name *mytape* as the logical tape name. Verify that the tape is allocated by viewing a long listing of the entry *mytape*. Your entry:

```
% tpalloc mytape
/dev/rmt8 allocated; linked to mytape
% ls -l mytape
lrwxrwxr-x 1 root        9 Dec 12 11:24 mytape -> /dev/rmt8
%
```

Release the tape drive with the *tpdealloc* command. Your entry:

```
% tpdealloc mytape
ta0: not online
mytape deallocated
%
```

TAPE DRIVE ALLOCATION (cont.)

- *tpalloc* options (cont.):

  -r Requests that the tape unit not be rewound when the file is closed; the default is to rewind the tape unit when the file is closed

  ```
  % tpalloc -r mytape
  ```

- The *name* may be any name you wish to use; it appears as a filename in your current directory which is owned by *root*. This file is a link to the true device name.

  ```
  % tpalloc -d 1600 mytape
  /dev/rmt8 allocated; linked to mytape

  % ls -l mytape
  lrwxrwxr-x  1 root  9 Dec 12 11:24 mytape -> /dev/rmt8
  ```

tpalloc Options

Allocate the drive; request a 6250bpi drive, block mode, and no rewind upon closing the unit. Use the logical filename *mytape*. Your entry:

```
% tpalloc -b -d 6250 -r mytape
/dev/mt20 allocated; linked to mytape
%
```

TAPE DRIVE DEALLOCATION

tpdealloc(1)

- Used to deallocate tape drives
- Format of *tpdealloc*:

  tpdealloc [-k] [*device_name*]
  tpdealloc [-k] [*logical_name*]

- *tpdealloc* option:

  -k Keeps the status quo of the tape unit -- leaves the tape unit online and does not rewind the tape

  ```
  % tpdealloc -k mytape
  mytape deallocated
  ```

- Relinquishes the exclusive access of a specific tape unit
- By default, *tpdealloc* takes the tape unit offline as it deallocates the unit

Tape Drive Deallocation

Release the tape drive; use the logical tape name. Your entry:

```
% tpdealloc mytape
ta0: not online
mytape deallocated
%
```

MAGNETIC TAPE MANIPULATION

mt(1)

- Used to give commands to a magnetic tape drive
- Format of *mt*:

  mt [-f *tapename*] *command* [*count*]

- *mt* options:

bsf	Backspace *count* files % `mt -f /dev/rmt16 bsf 2`
fsf	Forward space *count* files % `mt -f /dev/rmt16 fsf 3`
offline, rewoffl	Rewind the tape and place the tape unit offline (*count* is ignored) % `mt -f /dev/rmt16 offline`
status	Print status information about the tape unit 0: operation successful 1: unrecognizable command 2: operation failed % `mt -f /dev/rmt 16 status`

Using the *mt* Command

Assume you have used the default drive and have read serveral files and want to rewind the tape. Use the *mt* command to rewind the tape and place the tape unit offline. Your entry:

```
% mt -f mytape rewoffl
%
```

Request the status of your tape drive; then release the drive. Your entry:

```
% mt -f mytape status
ta0: not online
mytape: I/O error
% tpdealloc mytape
ta0: not online
mytape deallocated
%
```

TAPE ARCHIVER

tar(1)

- Saves and restores multiple files on a single file
- Format of *tar*:

  tar [*option_string*] [*name ...*]

- *tar* options:

  c Create a new tape; the files are added at the beginning of the tape
  % `tar c file1 file2`

  r The named files are written on the end of the tape
  % `tar r file1 file2 file3`

  t The specified files are listed each time they occur on the tape. If no file argument is given, all of the files on the tape are listed.
  % `tar t`

  v The verbose option makes *tar* list each file it processes preceded by its function letter. When used with the *t* option, *tar* provides even more information.
  % `tar tv`

tar

If it is inconvenient for you to work with a tape drive at this time, you may not be able to perform some of the following exercises.

Allocate a tape drive under the name of *mytape*. Write *file1* and *file2* to the tape. Verify the files are on the tape with the *t* option. Your entry:

```
% tpalloc mytape
/dev/rmt8 allocated; linked to mytape
% tar cv file1 file2
a file1 126 blocks
a file2 7 blocks
% tar t
file1
file2
%
```

Determine what is on the tape with both the *t* and *v* options. Your entry:

```
% tar tv
rw-r--r--311/74  64162 Dec 15 15:51 1986 file1
rw-r--r--311/74   3447 Dec 15 15:52 1986 file2
%
```

TAPE ARCHIVER (cont.)

- *tar* options (cont.):

  x Extract the named files from the tape. If no file argument is designated, the entire contents of the tape is extracted. If the file argument is a directory found on the tape, the entire directory is extracted. The owner, modification time, and mode are restored whenever possible. If multiple copies of the same file are found on the tape, the last copy overwrites all earlier versions extracted from the tape.

```
% tar x source.dir
```

  f *Tar* uses the next argument as the name of the archive instead of */dev/rmt?*. If the file is named -, tar writes to standard output or reads from standard input. Thus *tar* can be used in a filter chain.

```
% tar cvf /dev/rmt16 newdir1
% cd olddir; tar cvf - . | \
(cd newdir; tar xvf -)
```

  C If a file name is preceded by *-C*, *tar* will change to that directory. This allows multiple directories to be archived using short relative path names.

```
% tar cv -C /usr include -C /etc
```

Extracting Files Using *tar*

Move *file1* and *file2* to **.sav*. Then extract the two files from the tape and verify they exist in the directory with the *ls -l* command. Your entry:

```
% mv file1 file1.sav
% mv file2 file2.sav
% tar xv file1 file2
x file1, 64162 bytes, 126 tape blocks
x file2, 3447 bytes, 7 tape block
% ls -l file*
-rw-r--r-- 1 user      64162 Dec 15 15:51 file1
-rw-r--r-- 1 user      64162 Dec 15 15:51 file1.sav
-rw-r--r-- 1 user       3447 Dec 15 15:52 file2
-rw-r--r-- 1 user       3447 Dec 15 15:52 file2.sav
%
```

Use the *f* option to designate the logical tape name. Add the two **.sav* files to the tape. Verify the two additional files are on the tape. Your entry:

```
% tar rvf mytape *.sav
a file1.sav 126 blocks
a file2.sav 7 blocks
% tar tv
rw-r--r--311/74  64162 Dec 15 15:51 1986 file1
rw-r--r--311/74   3447 Dec 15 15:52 1986 file2
rw-r--r--311/74  64162 Dec 15 15:51 1986 file1.sav
rw-r--r--311/74   3447 Dec 15 15:52 1986 file2.sav
%
```

Use *tar* to copy the *passwd* file from the *etc* directory to */tmp*. Verify that your command works correctly; then remove the */temp/passwd* file. Your entry:

```
% cd /etc; tar cvf - passwd | (cd /tmp; tar xvf -)
a passwd 17 blocks
x passwd, 8386 bytes, 17 tape blocks
% pwd
/etc
% cd /tmp
% ls -l passwd
-rw-r--r-- 1 username      8386 Feb  9 09:51 passwd
% rm passwd
%
```

ANSI LABELED TAPES

ansitar(1)

- Reads and writes ANSI multifile labeled tapes
- Useful for exhange of ASCII character files with a non-ConvexOS system
- Will read VMS *copy* tapes which have been initialized and mounted as VMS volumes with labels
- Format of *ansitar*:

  ansitar [*subcommand*] [*option_values*] [*name ...*]

- Uses the same options as *tar*; additional options are provided for working with labeled tapes

Using *ansitar*

Allocate a tape drive under the name of *mytape*. Your entry:

```
% tpalloc mytape
/dev/rmt8 allocated; linked to mytape
%
```

ansitar OPTIONS

c Create a new tape; the files are added at the beginning of the tape
```
% ansitar c file1 file2
```

r The named files are written on the end of the tape
```
% ansitar r file1 file2 file3
```

t The specified files are listed each time they occur on the tape. If no file argument is given, all of the files on the tape are listed. If both the *t* and *v* options are given, *ansitar* provides even more detailed information about the files.
```
% ansitar t
```

x Extract files from the tape. The * wild card character may be used to get a number of related files from the tape, but it must be quoted to prevent the shell from attempting to expand the wild card.
```
% ansitar x 'file*'
```

Creating a Tape Using *ansitar*

Create a new tape with *ansitar*; write *file1* and *file2* onto the beginning of the tape. Verify the files are on the tape with the *t* option. Your entry:

```
% ansitar c file1 file2
% ansitar t
Volume label:
        Label: VOL1 Serial: ANSTAR Access:
        Owner: user         Standard: 1
t file1         126 blocks 64162 bytes
         D-type: blocksize: 512
t file2  -        7 blocks 3448 bytes
        D-type: blocksize: 512
2 files
%
```

ansitar OPTIONS (cont.)

D Reading: not required for system level 3 tape with valid HDR2 label; for system level 1 and 2, use next argument as tape block size
Writing: write HDR2/EOF2 labels and variable-length records (D format, system level 3) with block size of 2048 bytes and maximum line size of 255
% `ansitar xD 2048 file1`

F Reading: not required if tape has valid HDR2 label; use *B* option for ANSI system level 1 tapes
Writing: same as option *B*, but write HDR2/EOF2 label for automatic unblocking (F format, system level 3)
% `ansitar cF 1000 'file*'`

v Use the *verbose* mode; generates detailed information concerning labels
% `ansitar cv file1 file2`

V Use the next argument (up to 6 alphanumeric characters) as the tape *vsn*; it is preferable to use only uppercase alphabetics in the argument
% `ansitar cV TAPE00 file1`

f Use the next argument as the tape unit to use; the default is */dev/rmt8*
% `ansitar cf /dev/rmt16 file1`

Using *vsn*

Write the files *file1* and *file2* at the beginning of the tape. Designate that the *vsn* is *MYTAPE* and the tape unit name is *mytape*. Verify that the files were correctly written to the tape by *ansitar tv*. Finally, deallocate your tape drive. Your entry:

```
% ansitar cvVf MYTAPE mytape file1 file2
a file1 126 blocks 64162 bytes
a file2 7 blocks 3448 bytes
% ansitar tv
Volume label:
        Label: VOL1 Serial: MYTAPE Access:
    -   Owner: user        Standard: 1
File Label:
        Label: HDR1 File: file1
        Set: MYTAPE Section: 0001 Sequence: 0001
        Generation: 0001 Generation Version: 00
        Created: 86348 Expires: 99364 Access:
        Blocks: 000000 System: CONVEX UNIX
File Label:
        Label: EOF1 File: file1
        Set: MYTAPE Section: 0001 Sequence: 0001
        Generation: 0001 Generation Version: 00
        Created: 86348 Expires: 99364 Access:
        Blocks: 000126 System: CONVEX UNIX
% tpdealloc mytape
mytape deallocated
%
```

Exercises for Chapter 7

1. Write a command that reserves a 125-inch/second tape drive unit?

2. What is the difference between block and raw mode for tape I/O?

3. What is the default tape drive unit selected by *tpalloc*?

4. Write a command that releases a tape drive unit so that other people may use it?

5. What UNIX utility permits you to control the motion of the tape reel?

6. Write a command that determines the status of a tape drive?

7. How can you use *tar* in a UNIX filter chain?

8. What UNIX utility can be used to read VAX VMS copy tapes?

9. Obtain a scratch tape from your instructor.

    a. Allocate the tape drive for 6250 density.
    b. Load the tape on the appropriate tape drive.
    c. Use ansitar/tar command to transfer a directory (and subdirectories) to the tape with the following options:

        a. Create a new tape.
        b. Rewind on close.
        c. Record at 6250 bpi.
        d. Display the file names as they are transferred to the tape.
        e. Make a new directory named *SHOWME*.
        f. Use the ansitar/tar command to extract your directory and/or files as specified by your instructor into the *SHOWME* directory.
        g. Have your instructor verify your work.

8
Problem Reporting

Objectives for Section 8

After completing Chapter 8, you will be able to:

1. Submit problem reports to CONVEX.
2. Utilize the editing options found in *contact*.
3. Review a problem report.
4. Generate a *.contact* file

SUBMITTING A PROBLEM REPORT

contact(1)

- Submits problem reports to the CONVEX Technical Assistance Center (TAC)
- Queries the user for all the necessary problem information
- Sends the report to CONVEX by electronic mail
- Sends the files to CONVEX by *uucp*
- Notifies the user that the analysts at CONVEX have received the problem report
- Provides the user with a problem report number which is used to reference the problem report
- Notifies the user when a solution has been found for the problem report

Submitting Problem Report

For the exercises in this module, you will need two files: *file1* and *file2*. These files should contain your login name and a statement which designates that this is a test of *contact*. Use your favorite UNIX editor. Your entry:

```
% cat file1
user
This is only a test file for contact.
This is not a report of a problem.
% cat file2
user
This is only a test file for contact.
This is not a report of a problem.
%
```

INFORMATION SUPPLIED

The user should supply the following information:

- User Information – user's name, title, corporate name, and phone number

```
Mr. User
Software Engineer
Corporation
(100) 999-1000
```

- Product – name and version of the product involved in the problem report

```
Fortran Compiler Version 4.0
```

- Summary – a one-line summary of the problem

```
mine.f will not compile at -O2
```

- Description – a detailed description of the problem

```
When I attempt to compile this
routine, the compiler gives the
following error message: Error on
line .35 of mine.f: recursive calls
are not permitted in FORTRAN
```

Initiating a *contact* Report

Initiate a *contact* report. Answer the prompts for the user information. Please define the product as N/A, the summary as **No problem exists**, and the description as **Only a test**. Your entry:

```
% contact
Welcome to contact version 0.11

Enter your name, title, phone number, and corporate name (^D to terminate)
> Your Name
> Your Title
> Your Corporation
> Your Phone Number
> .

Enter the name of the product involved
> N/A

Enter the version of the product involved
> N/A

Enter a short (1 line) summary of the problem
> No problem exists

Enter a detailed description of the problem (^D to terminate)
> Only a test
> .
```

INFORMATION SUPPLIED (cont.)

- Priority – a measure of the importance of the problem; please select from one of the following six options:

  1) Critical – work cannot proceed until the problem is resolved

  2) Serious – work can proceed around the problem, with difficulty

  3) Necessary – problem has to be fixed

  4) Annoying – problem is bothersome

  5) Enhancement – requested enhancement

  6) Informative – for informational purposes only

- Repeat-By – instructions on how to repeat the problem

```
fc -V -O2 -c -w -a mine.f
```

- Files – a list of any files necessary to duplicate the problem

```
mine.f
```

- Comments – any other comment which might be relevant to the problem report

```
Routine will not compile
```

contact Information Supplied

Please designate the *priority* as *Informative* (6). Enter the instructions necessary to repeat the problem as **No Instructions**. Reply **yes** to the question that asks if you are including any files and use *file1* and *file2* as the files to be included. Your entry:

```
Enter a problem priority, based on the following:
1) Critical   - work cannot proceed until the problem is resolved.
2) Serious    - work can proceed around the problem, with difficulty.
3) Necessary  - problem has to be fixed.
4) Annoying   - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
> 6

Enter instructions by which the problem may be reproduced (^D to terminate)
> No Instructions
> .

Enter any comments that are applicable (^D to terminate)
> .

(Optional) Do you have any suggestions or comments on the documentation
that you referenced when you were trying to resolve your problem (for
example, additions, corrections, organization, accessibility)?
(^D to terminate)
> .

Are there any files that should be included in this report (yes | no)?
> yes
Please enter the names of the files, one to a line (^D to terminate)
> file1
> file2
> .
```

DISPOSITION OF *contact* REPORT

After all the sections have been entered into a *contact* report, the user may request any of four operations:

- Review – the report may be reviewed with *more*
- Edit – the text editor is invoked on the problem report
- Submit – submit the problem report, and terminate the *contact* session
- Abort – terminate the *contact* session without submitting the report. The report is saved in the user's home directory under the name of *dead.report*. Invoking *contact* with the *-r* option causes *dead.report* to be used as input.
- CTRL-c – abort the *contact* session without saving the contents of the report

Disposition of *contact* Report

Specify a disposition of *abort* for the problem report. Your entry:

```
Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
> 4

Saving report in /mnt/user/dead.report... done
%
```

.contact FILE

- The initial user information required by *contact* may be placed into a file named *.contact*
- The *.contact* file should exist in the user's main directory
- The *.contact* file contains the user information as follows:

```
Mr. User
Software Engineer
Corporation
(100) 999-1000
```

- When the *.contact* file exists, *contact* reads the information from the *.contact* file
- Thus the first prompt the user sees is the following:

```
Enter the name and version
of the product involved
```

.contact File

Generate a *.contact* file using your editor of choice. Then execute the *contact* utility. Answer *dot* (.) to all the questions. Then request to review the report you have generated by selecting **1**. After reviewing the report, abort the *contact* report by selecting **4**. Your entry:

```
% cat .contact
User Name
Job Title
The Corporation
Your Phone Number
% contact
Welcome to contact version 0.11

Enter the name and version of the product involved
> .

Enter a short (1 line) summary of the problem
> .

Enter a detailed description of the problem (^D to terminate)
> .

Enter a problem priority, based on the following:
1) Critical    - work cannot proceed until the problem is resolved.
2) Serious     - work can proceed around the problem, with difficulty.
3) Necessary   - problem has to be fixed.
4) Annoying    - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
> 6

Enter the instructions by which the problem may be reproduced
> .

Are there any files that should be included in this report (yes | no)?
> no

Enter any comments that are applicable (^D to terminate)
> .

(Optional) Do you have any suggestions or comments on the documentation
that you referenced when you were trying to resolve your problem (for
example, additions, corrections, organization, accessibility)?
(^D to terminate)
.

Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
> 1
Hit return to continue:

...
> 4
Saving report in /mnt/user/dead.report. . .done
%
```

contact OPTIONS

The *contact* utility has two types of options:

- Options on the command line
  - a – use an alternate configuration file entry for *contact*
  - r – start *contact* up with a file named *dead.report* as input
- Options to be used in multi-line sections of *contact* (User-Information, Description, Repeat-By, and Comments)
  - ~e – invoke a text editor on the information in this section; *vi* is the default editor. You may define the editor by setting the *EDITOR* environment variable.
  - ~h – display a list of the available tilde escapes
  - ~p – print the information in this section on the terminal
  - ~r file – read the contents of *file* into the current section
  - ~~ – insert a single tilde as the first character in the line

contact Options

Restart the same *contact* report with the *-r* option on the command line. Edit your *contact* file with the editor selected with the *EDITOR* environment variable; the default is *vi*. Exit the editor. Your entry:

```
% contact -r
Welcome to contact version 0.11
"/tmp/tac_a028002" 18 lines, 193 characters Subject: CPU-74 (convext)
Cc: user
User-Information:
...
:wq
```

Abort the *contact* session and remove your *dead.report* file. Your entry:

```
Hit return to continue:

Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
> 4
Saving report in /mnt/user/dead.report... done
% rm dead.report
%
```

To experiment with the tilde commands, start a contact report. Write any comments that you feel are appropriate. When you reach the *comment* section, request a list of available *tilde* commands. Finally, abort the session with ^C. Your entry:

```
% contact
Welcome to contact version 0.11
...
...
Enter any comments that are applicable (^D to terminate)
> This is a comment
> ~h
Available tilde functions are:
~e          edit the text buffer.
~h          display this list.
~p          print the text buffer.
~r file     read the contents of file into the text buffer.
> .
```

Abort this contact report without saving the contents by typing ^C.

Exercises for Chapter 8

1. What utility do you use to report problems to CONVEX?

2. What is the purpose of the *-r* option on the *contact* command line?

3. How do you invoke an editor while in *contact*.

4. What are the possible dispositions for a *contact* report?

5. Submit a problem report to CONTACT. Please designate that this is just a practice report. The problem information: **No problem - just a UNIX training exercise.** Abort the contact report, sending the contents to *dead.report*. Mail the contents of this report to your instructor or print a copy of the contact report and give it to your instructor.

A

VI Commands

Command	Description
h	Back 1 character
(	Back 1 sentence
B/b	Back 1 word
s	Change current letter then insert
C	Change entire line (cc)
r	Change current letter Only
~	Change case one character
cb/cw	Change word backward/forward
Q	Change to ex editor
:vi	Change back to vi
ctx	Change through letter "x"
R	Change multiple characters
:'a,'bt.	Copy Block after current line
:'a,'bd	Delete block a - b
dd	Delete line
"xndd	Delete n lines into buffer x (a-z)
d(/ d)	Delete to beginning/end of sentence
D	Delete to end of line
db/dw	Delete word backward/forward
X/x	Delete character before/under cursor
:set all	Display all options
z-	Display current line at bottom
z.	Display current line at center
z	Display current line at top
:set	Display current options
^G	Display file information
(cr)	Down 1 line - first character
j	Down 1 line - same column
YP/Yp	Duplicate current line before/after
:e <file>	Edit different file
Fx/fx	Find previous/next "x" same line (; and , = repeat)
l	Forward 1 character
W/w	Forward 1 word
)	Forward 1 sentence
L	Go to bottom of screen

Command	Description
n|	Go to character n (same line)
G	Go to end of file
$	Go to end of line
0	Go to first character of line
M	Go to middle of screen
nG	Go to line n (1G = Line 1)
H	Go to top of screen
A/a	Insert after EOL/cursor
I/i	Insert before line/cursor
O/o	Insert before/after current line
DEL	Interrupt (Stop)
:set nu	Line number enable
:set nonu	Line number disable
mx	Mark Position (x= a-z)
:'a,'bm.	Move Block (a thru b)
^B/^U	Full/half page up
^F/^D	Full/half Page Down
:q	Quit editor no mods made
:q!	Quit editor abandon changes
:r <file>	Read file (insert)
^l	Refresh screen
.	Repeat last command
P/p	Restore deleted text Before/After
"np	Restore deleted buffer n (1-9)
"xP/xp	Restore from buffer x (a-z)
`x	Return to mark x (a-z)
``	Return to previous position
ZZ	Save File & Quit editor
^Y/^E	Scroll down/up one line
?^<strng>	Search backward for <string> beginning of line only
?<strng>$	Search backward for <string> end of line only
?<string>	Search file backward

Command	Description
/<string>	Search file forward
/^<strng>	Search forward for <string> beginning of line only
/<strng>$	Search forward for <string> end of line only
N/n	Search repeat opposite/same direction
------>	:1,$s/original/changed to/g Global search & replace
!}fmt	format
:n	goto next file<L one shiftwidth lef

Command	Description
>L	Shift remaining lines on screen one shiftwidth right
:set sw=n	Set shiftwidth n spaces
!}sort	Sort until next blank line
U	Undo all changes to one line
u	Undo last command
k	Up 1 line - same column
:'a,'bw!	Write block (a-b) to existing file
:'a,'bw	Write block (a-b) to new file
:w <file>	Write file
Yp/YP	Yank buffer
"xny	Yank into buffer x, n lines

Entering/Leaving *vi*

The following table illustrates the basic ways of entering and exiting *vi*:

Table A-1: Entering/Leaving *vi* Commands

Command	Explanation
vi *filename*	edit *filename*
vi +*n filename*	When the text appears on the screen, the cursor is positioned on the *n* line.
:wq	Writes buffer contents to permanent storage; returns you to the *shell*
:q!	Exits the editor and aborts all modifications to the buffer (file)
:w *filename*	Writes buffer contents to permanent storage to new *filename* without exiting to *shell*; will not write to an existing file
:w! *filename*	Writes buffer contents to an existing *filename*; overwrites contents of *filename*

Command Modes in *vi*

There are two modes in *vi*:

Table A-2: *vi* Command Modes

Mode	Explanation
command	Normal and initial startup state; can enter commands but no text; to return to command mode press the ESC key
text input	Entered by typing a command (**a**/A—append, **i**/I—insert, **o**/O—open line, **c**/C—change, **s**/S—overtyping, **R**—replaces) followed by the text; must use ESC to return to command mode

Specifying a Terminal for *vi*

Use the command **set term**=*terminal* in your *.exrc* file. You can also set the environmental variable *TERM* to your *terminal* type, i.e., **setenv term** *terminal*.

B

UNIX Command Summary

The following table lists commonly used UNIX commands by category of use.

Table B-1: Common UNIX Commands

Category	Commands	Description
Batch	`qstat` `qdel` *job* `qsub` *f1*	list jobs in queue remove *job* from batch queue submit executable file to batch queue

Communication	`biff`
	`mail`
	`mesg`
	`talk`
		determine type of mail notification
		send/receive electronic mail
		permit or deny messages from others
		start two-way terminal communication

Directory	`cd` *d1* `mkdir` *d1* `pwd` `rmdir` *d1*	change directory to *d1* make new directory named *d1* print working (current) directory remove empty directory *d1*
Editing	`awk` *commands* `emacs` *f1* `ex` *f1* `sed` *commands* `vi` *f1*	string processing language *emacs* full-screen text editor *ex* line editor batch line editor *vi* full-screen text editor
File	`cat` *f1 f2* `chmod` *opt f1* `cp` *f1 f2* `find` *options* `head` *f1* `less` *f1* `ln` [`-s`] *f1 f2* `ls` [`-alF`] `more` *f1* `mv` *f1 f2* `rm` [`-r`] *f1 f2* `tail` *f1* `tee`	list both files *f1* and *f2* to standard out change permission bits for file *f1* copy file *f1* to *f2* find and act on designated files show top portion of file *f1* show page with forward and backward scrolling link file *f2* to existing file *f1* list files in directory show page with forward scrolling move file *f1* to *f2* remove files *f1* and *f2* (-r recursively) show tail portion of file *f1* split output into two files

Table B-1: Common UNIX Commands (cont.)

Category	Commands	Description
Filter	`cmp` *f1 f2* `dd if=`*f1 of=f2* `diff` *f1 f2* `grep` *string f1* `sort` *f1* `wc` *f1*	display first difference in files *f1* and *f2* copy data from *f1* to *f2*, converting data show differences between files *f1* and *f2* search for string in text file sort file *f1* count the words/lines in file *f1*<

Help	`info`
	`learn`
	`man [-k]` *command*
		access information by topic or menu selection
		access tutorial
		display manual page on selected topic

Interactive	`fg`
	`bg`
	`jobs`
	`kill`
	`ps`
		bring background job into foreground
		put foreground job into background
		list jobs in background
		kill job in background mode
		list all processes running on your behalf

Language	`as` *fl.s* `cc` *f1.c* `fc` *f1.f*	assemble file containing assembly code compile/link C language files compile/link FORTRAN files

Miscellaneous	`alias` *al string*
	`cal` *mo yr*
	`date`
	`echo` *string*
	`finger` *user*
	`history`
	`stty` *options*
	`time` *command*
		alias *al* to command found in string
		display a calendar
		print the date
		echo *string* to standard out
		provide user information
		view history of commands you have entered
		set terminal options
		time the command

Printer	`lpr` *f1*
	`pr` *f1*
	`lpq`
	`lprm`
	`nroff -ms` *f1*
		send *f1* to line printer
		add headings, etc., to printed output
		display printer request queue
		remove request from printer queue
		formats text to be printed

Tape	`ansitar`
	`mt`
	`tar`
	`tpalloc`
	`tpdealloc`
	`tpmnt`
	`tpq`
	`tprm`
	`tpumnt`
		ANSI labeled tape utility
		magnetic tape utility
		tape archive utility
		allocate tape drive
		deallocate tape drive
		request tape mount
		display tape request queue
		remove request from tape queue
		request tape unmount

User	`login username`
	`logout`
	`passwd`
	`who`
	`whoami`
		login in to system
		logout of system
		change user password
		list users currently logged in
		display username

C

Sample Scripts

This appendix contains sample scripts illustrating the various UNIX commands and utilities discussed in the UNIX training courses.

C Shell Scripts

Example 1::

```
#!/bin/csh -f
##################################################
#
#  This script will take an allocated tape unit
#  offline.  The reel will then be unmounted.
#  Finally, the tape drive will be deallocated.
#  The user must supply the logical tape name
#  which was selected when the tape drive was
#  allocated.
#
#  The script is used as follows:
#                 return logical_tape_name
#                     or
#                 return
#
##################################################
if($#argv == 0) then
  echo -n 'enter the logical file name: '
  set lfn = $<
else
  set lfn = $1
endif

mt -f $lfn offl
tpumnt $lfn
tpdealloc $lfn
```

Example 2:

```
#!/bin/csh -f
##########################################################
#
#  This utilitity can be used to read
#  either ASCII or EBCDIC files from
#  a tape.  First, the user selects
#  either ASCII or EBCDIC format.
#  The user supplies a filename
#  template for the files to be
#  extracted from the tape.  If no
```

```
#  template is designated, tapein.*
#  in used.  The utility will accept
#  densities of 1600 and 6250.
#  The user specifies the blocksize
#  in number of bytes.  Finally, the
#  user specifies the number of files
#  to be retrieved from the tape.
#  This script is hardwired to
#  read 80-character records.
#
#  How to run this script:
#     read_tape.scr
#
##############################################################################
#
type:
  echo -n 'enter tape format ' \
    '(A for ASCII, E for EBCDIC)'
  set in = $<
  if (($in != "A") && ($in != "E")) goto type
  if ($in = "A") then
    set conv = 'sync,unblock'
  else
    set conv = 'unblock,sync,ascii'
  endif

name:
  echo -n \
    'enter the name template to be used: '
  set in = $<
  if ($in == "") then
    set nufile = 'tapein'
  else
    set nufile = $in
  endif
  echo 'filename = '$nufile

dens:
  echo -n \
    'enter the tape density (1600 or 6250): '
  set in = $<
  if ($in != 1600 && $in != 6250) then
    echo 'invalid tape density, try again'
    goto dens
  else
    set density = $in
  endif

blok:
  echo -n \
    'enter blocksize as number of bytes: '
  set in = $<
  if ($in == "") then
    echo 'invalid record count, try again'
    goto blok
  else
    set blok = $in
  endif
```

```
numb:
  echo -n 'enter the number ' \
    'of files to be extracted: '
  set in = $<
  if ($in == "") then
    echo -n 'invalid number of files, try again'
    goto numb
  else
    @ in = ($in + 1)
    set count = $in
  endif

tpalloc -r -d $density tape_$$
set i = 1
while ($i != $count)
  dd if=tape_$$ of=$nufile.$i ibs=$blok \
    cbs=80 conv=$conv
  @ i = ($i + 1)
end
mt rewind
mt -f tape_$$ offl
tpdealloc tape_$$
```

Example 3:

```
#!/bin/csh -f
##############################################################################
#
#   This script allows you to send an electronic
#   letter to one person or to everyone in a list
#   which you provide.  You generate the letter
#   with an editor and save it.  If you wish to
#   build a list of names, you should place one
#   name per line in a text file.  If you are
#   sending a letter to only one person, the
#   third argument must be a '1', designating the
#   second argument to be an an individual's name.
#   If the script receives only two arguments, the
#   second argument is taken to be a list of users
#   to which the letter is to be sent.  If the
#   script receives more than two arguments, and
#   the third argument is not a '1', the second
#   through last arguments are accepted as
#   usernames.
#
#   How to use this script:
#      mailit.scr note_name username 1
#      mailit.scr note_name user_list
#      mailit.scr note_name user1 user2 user3
#
##############################################################################
#
#
#  check validity of arguments supplied
#
start:
   set num = $#argv
   if ($num < 2) then
      echo "insufficient arguments supplied, " \
```

```
      "try again"
    goto start
  endif
#
#  how many users are being sent mail
#
  if ($#argv == 2) then

    foreach i (`cat $2`)
#       use file which contains the list of users
      mail -v $i < $1
    end

  else

    if ($3 == 1) then
#     send to only one user
      mail -v $2 < $1
    endif

    if ($3 != 1) then
#     send mail to people listed as arguments
      set i = 2
      while ($i <= $num)
        mail -v $argv[$i] < $i
        @ i = ($i + 1)
      end
    endif

  endif
#
#  work completed
#
  echo "letters sent"
  echo "mailit.scr terminated"
```

Example 4:

```
#!/bin/csh -f
##################################################
#
#  This script will monitor a job and
#  announce when that job terminates.
#  The user designates the PID
#  (process id) to be monitored.
#
#  How to run this script:
#     monitor pid interval
#     monitor
#
##################################################
#
#  acquire PID
#
  if !($1) then
    clear
loop1:
    echo -n 'ENTER PID TO BE ' \
      'MONITORED (NUMBERS ONLY) : '
```

```
    set j = ($<)
  else
    set j = $1
  endif
#
#  check PID
#
  if ($j < 7 || $j > 99999) then
    clear
    echo 'BEYOND SYSTEM LIMITS FOR PIDS'
    goto loop1
  endif
#
#  acquire interval
#
  if !($2) then
    clear
    echo -n 'ENTER NUMBER OF SECONDS ' \
      'BETWEEN CHECKS OF PID : '
    set check = ($<)
  else
    set check = $2
  endif

  if ($check < 15) then
    set check = 15
  endif

  set k = `ps $j | grep '^'$j | wc -l`
#
#  process is not running
#
  if ($k == 0) then
    clear
    echo 'NOT AN EXECUTING PROCESS'
    echo 'TRY AGAIN PLEASE'
    sleep 1
    goto loop1
  endif
#
#  process is currently running
#
loop2:
    set k = `ps $j | grep '^'$j | wc -l`

    if ($k == 0) then
        set cnt = 0
      goto END
    else
      sleep $check
      goto loop2
    endif
#
#  process has terminated
#
  END:
    @ cnt = ($cnt + 1)
    echo ''
    echo 'PROCESS ' $j ' HAS COMPLETED'
```

```
    sleep 8
    if ($cnt < 10) then
      goto END
    endif
```

Example 5:

```
#!/bin/csh -f
##################################################
#
#  This script will find the home directory
#  for a user when only the username
#  is known.  This script looks in the system
#  file /etc/passwd and finds the entry
#  which matches the user's name.  The script
#  then selects the sixth field which contains
#  the home directory.  This information is
#  then put into a file named user_home.
#  This script is helpful when the user is
#  coming into the CONVEX through RJE software
#  and the default value of the environment
#  variable '$HOME' is '/'.
#
#  How to run this script:
#     fhd.scr username
#
##################################################

#  check input parameter

#
if($1 == "") then
        echo "no username specified"
        exit(1)
endif

##################################################
#  select entry in /etc/passwd file
##################################################
set colon = ":"
grep '^'$1$colon /etc/passwd > name_$$

##################################################
#  determine if only one user was selected
##################################################
wc -l name_$$ > lcnt_$$
set ncnt = `awk '{print $1}' < lcnt_$$`
if ($ncnt == 0) then
        echo "user name is not known"
        rm -r name_$$
        rm -r lcnt_$$
        exit(2)
endif
if ($ncnt != 1) then
        echo "user name is not unique"
        rm -r name_$$
        rm -r lcnt_$$
        exit(3)
endif
```

```
##########################################################
#  pick out the home directory field
##########################################################
set hd = `awk -F":" '{print $6}' < name_$$`
echo $hd  > $1_home

##########################################################
#  wrap-up
##########################################################
rm -f name_$$
rm -f lcnt_$$
exit(0)
```

Example 6:

```
#! /bin/csh -f
#
#############################################################
#
#   This script will modify the priorities of very
#   large jobs found in the 'c' queue.  If a large
#   job is active in the 'c' queue, then the script
#   analyzes the load in the system.  If the total
#   of the active jobs is the 'a' and 'b' queues is
#   less than 2, the priority of the large job in
#   the 'c' queue is changed to +4.  If the total
#   of the jobs in the 'a' and 'b' queues is equal
#   to 2, the priority of the large job in the 'c'
#   queue is changed to +16.  The script checks
#   the load on the system every 5 minutes.  This
#   script is run by the "super-user" and is
#   restarted in the rc.local file during reboot.
#
#   How to run this script:
#      nite.batch   (run by root)
#
#   Note:
#      high priority  <  normal  <  low priority
#
#############################################################
#
top:
#  sleep for 5 minutes
   sleep 300

  ##########################################################
  #  determine if a 'c' batch job is active
  ##########################################################
    list -n c >> all
    fgrep 'active' all > actv
    set ccnt = `wc -l < actv`
    if $ccnt == 0 goto top

  ##########################################################
  #  determine if both 'a' and 'b' batch jobs are
  #  active
  ##########################################################
    list -n a > all
    list -n b >> all
```

```
  fgrep 'active' all > actv
  set abcnt = `wc -l < actv`

##################################################################
#  modify the priority of the job in the 'c'
#  as needed
##################################################################
if ($abcnt !=2) then
#  renice the active job in 'c' queue to pr#4
#  pseudo-user is called 'niteman'
  /etc/renice +4 -u niteman >& /dev/null
else
#  renice the active job in 'c' queue to pr#16
  /etc/renice +16 -u niteman >& /dev/null
endif
goto top
```

Example 7:

```
#!/bin/csh -f
##################################################################
#
#  This script will send the output to the best
#  available printer found in the printer data base.
#  The printer data base is found in a file named
#  ~/bin/.printers.  This script calls an awk
#  script to determine the best printer to use.
#  The name of the selected printer is then placed
#  into a file named ~/bin/.bestprinter.  This
#  script should be placed in ~/bin directory.
#
#  How to use this script:
#     lpr filename
#
##################################################################
#
if (-e ~/bin/.printers) then
        foreach arg ($argv[*])
                switch ($arg)
                        case -P*:
                                /usr/ucb/lpr $argv[*]
                                exit
                        default:
                endsw
        end
        set best = `best`
        echo "lpr: printing on $best"
        echo $best >! ~/bin/.bestprinter
        /usr/ucb/lpr -P$best $argv[*]
else
        /usr/ucb/lpr $argv[*]
endif
```

Example 8:

```
#!/bin/csh -f
##################################################################
#
#  This script will determine the best
#  printer to be used at this time.  It
#  will find an idle printer if one exists.
#  The script will search through a list
#  of printers found in the .printers
#  (see previous example)
#  file in the user's /bin directory.
#  The .printers file contains the names
#  of available printers as they are listed
#  in the printcap system file.  The
#  script will attempt to use these
#  printers in the order listed in the
#  .printers file.  These printer names
#  may be listed in separate lines, or on
#  one line with blanks separating the
#  names.
#
#  How to use this script:
#     best   (executed by the lpr script, previous example)
#
##################################################################
#
if (-e ~/bin/.printers) then
   set printers = (`cat ~/bin/.printers`)
   @ s2 = 2147483647
   set best = $printers[1]

   foreach p ($printers)
      set s1 = \
         `(/etc/lpc status $p; lpq -P$p) \
         | awk -f ~/bin/best.awk -`
      if ($s1 == "") then
         set best = $p
         break
      endif
      if ($s1 < $s2) then
         set s2 = $s1
         set best = $p
      endif
   end

   echo $best

else

   echo "best: no ~/bin/.printers"

endif
```

Example 9:

```
##################################################################
#
#  This awk script will parse the output of the
#  /etc/lpc utility.  The output of this awk script
#  will be used by the best C shell script.  (See the
#  preceding example)
#
#  How to run this awk script:
#     awk -f best.awk
#
##################################################################
{
   for (i = 0; i <= NF; i++) {
      if (($i === "unknown") || ($i === "disabled")) {
         s = 2147483647
         break
      }
      if ($i === "bytes")
         s += $(i-1)
      }
}
END     {print s}
```

Example 10:

```
#!/bin/csh -f
##################################################################
#
#  This script will print the load information on the
#  last line printer selected as the best one to use.
#  If a best printer has not been selected,
#  (see preceding printer script examples) this
#  script will print the load information on the
#  requested line printers.
#
#  How to use this script:
#     lpq      or      lpq -Pprinter
#
##################################################################
#
if (-e ~/bin/.bestprinter) then
   foreach arg ($argv[*])
      switch ($arg)
         case -P*:
            /usr/ucb/lpq $argv[*]
            exit
         default:
      endsw
   end
   set best = `cat ~/bin/.bestprinter`
   echo "lpq: last printer was $best"
   /usr/ucb/lpq -P$best $argv[*]
else
   /usr/ucb/lpq $argv[*]
endif
```

Example 11:

```
#!/bin/csh
##################################################################
#
#  This script will perform full (level 0) dumps for
#  each mounted file system.  The script uses the df
#  utility to determine which file systems to dump.
#  Each file system will be placed on a separate tape.
#
#  How to use this script:
#     full_dump
#
##################################################################

echo " "
echo "This script performs a full (level 0) dump"
echo "of all mounted file systems." ; echo " "
set df_out = 'df | awk 'NR > 1 {print $6}' '
set df_cnt = 'echo $df_out | wc -w'
@ df_cnt = ($df_cnt + 1)
echo "Each file system is dumped to a separate tape."
echo " "

set ptr = 1
while ($ptr != $df_cnt)
     echo -n "Press ENTER when the tape for " \
        $df_out[$ptr] " is ready. "
     set ans = ($<)
     dump 0uG   $df_out[$ptr]
     echo $df_out[$ptr] " has been dumped, " \
        "please dismount the tape."
     echo " "
     @ ptr = ($ptr + 1)
end

echo " " ; echo "The full dump is done."
```

awk Scripts

Example 1:

```
##################################################################
#  This script provides the user with a list of
#  jobs waiting to be run by at.  This script
#  calls the awk program named at.awk.  All jobs
#  waiting to be started by at are found in a
#  directory named /usr/spool/at.
#
#  To run this script:
#     at.scr
##################################################################
ls -l /usr/spool/at > /tmp/at.list
awk -f at.awk /tmp/at.list
rm -f /tmp/at.list
```

Example 2:

```
##########################################################################
#  This awk program analyzes the list of jobs to
#  be run by at.  These jobs are found in a
#  directory named '/usr/spool/at.
#
#  To run this script:
#     awk -f at.awk filename
#
#  Notes:
#  A typical list will look as follows:
#  total 2
#  -rw-r--r-- 1 hill 769 Jun  3 14:56 86.153.1600.96
#  -rw-rw-rw- 1 root   5 Jun  3 14:45 lasttimedone
#  drwxr-xr-x 2 root 512 Jun  3 10:45 past
##########################################################################
BEGIN {
        # print title
        printf "%s%s0,"user      time entered queue  ", \
        "time to be run      filename"
        printf "%s0, \
        "-------------------------------------------------------"
        icnt = 0
}
{
# acquire list of jobs to be run
        if ((NF > 3) && ($3 != "root"))         {
                ++icnt
                name[icnt] = $3
                que[icnt] = $5 " " $6 " " $7
                job[icnt] = $8
                split($8,array,".")
                time[icnt] = array[3] " hours " \
                array[2] "." array[1]
                printf "%-10s%-20s%-20s%-20s0, \
                name[icnt],que[icnt],time[icnt],job[icnt]
        }
}
END     {
        if (icnt == 0)
                print "no jobs are waiting in the at queue"
}
```

Example 3:

```
##########################################################################
#  This script will list the people current logged
#  onto the C 1.  This list will be sorted by the
#  user's login time.
#
#  To run this script:
#     who.awk
```

```
#
#  Notes:
#     The who utility provides the following
#     information:
#     user terminal month day time rlogin_cpu
#     $1__ $2_______ $3___ $4__ $5__ $6_______
#############################################################################
who | awk '{ print $5, $1 }' | sort
```

Example 4:

```
#############################################################################
#
#  This awk program will produce FORTRAN
#  listings with line numbers and basic
#  blocks marked with a level number.
#
#  To run this script:
#     awk -f list.awk filename
#
#############################################################################
#
BEGIN { blanks = "            " }          #  12 blanks

{
# search for do loop and if block reserved words
        for (i = 1; i <= 2; i++)          {
                idecr = 0
                if ($i == "do")           {
                        icont = icont + 1; break          }
                if ($i == "enddo")        {
                        idecr = 1
                        icont = icont - 1; break }
                if ($i == "if") {
                        icont = icont + 1; break          }
                if ($i == "endif")        {
                        idecr = 1
                        icont = icont - 1; break }
        }
}

{
# build the correct map of level numbers
        inum = icont
        if (idecr == 1) inum = icont + 1
        pad = ""
        for (i = 1; i <= inum; i++)
                pad = pad i "  "
        pad = pad blanks
        pad = substr(pad,1,12)
        line = pad $0
}
{
# prepend line numbers to source lines
        lnum = "     " NR; len = length(lnum)
        lnum = substr(lnum,len-4,5)
        print "<"lnum"> " line
}
```

```
# example output for this script
<    1>                    program sample
<    2>
<    3>                    real a(100), b(100), c(100)
<    4>
<    5> 1                  do i = 1,100
<    6> 1                  a(i) = b(i)
<    7> 1                  enddo
<    8>
<    9> 1                  if (iflag .eq. 0) then
<   10> 1  2               do i = 1,100
<   11> 1  2               b(i) = c(i)
<   12> 1  2               enddo
<   13> 1                  itemp = 100
<   14> 1                  else
<   15> 1  2               do i = 1,100
<   16> 1  2               b(i) = 1.0
<   17> 1  2               enddo
<   18> 1                  endif
<   19>
<   20> 1                100 if (iflag .eq. 1) then
<   21> 1                  itemp = -1
<   22> 1                  endif
<   23>
<   24>                    stop
<   25>                    end
```

D

Introduction to the Revision Control System

This appendix illustrates the basic steps for using RCS. It

1. Identifies the steps for creating documents, program modules, etc., with the Revision Control System (RCS).
2. Illustrates checking in a changed module.
3. Illustrates checking out a module for revision.
4. Show how to change the attributes of an RCS file.
5. Discusses printing log messages and other information about RCS files.

REVISION CONTROL SYSTEM

- Automatically manages multiple revisions of program modules, documentation, graphics, papers, and other text files
- Maintains each module as a tree of revisions
- Maintains a complete history of changes
- Logs all changes automatically
- Guarantees project continuity
- Manages and merges multiple lines of development
- Resolves access conflicts
- Performs automatic identification of modules
- Compatible with *make*
- Adds a minimum of storage or operational overhead

Purpose of the Revision Control System

The Revision Control System (RCS) centralizes and catalogs changes to any type of text: programs, documentation, memos, papers, graphics, etc. RCS stores and retrieves multiple revisions of programs and other text. You can maintain one or more releases while developing the next release. Changes do not destroy the original—previous revisions remain available.

RCS maintains each text module as a revision tree; it manages multiple lines of development. When two or more users wish to modify the same revision, RCS alerts the users and makes sure that one change does not conflict with the other.

For the exercises throughout this section, you will need the following text file. If you do not have the file *poem*, create it with your favorite UNIX text editor.

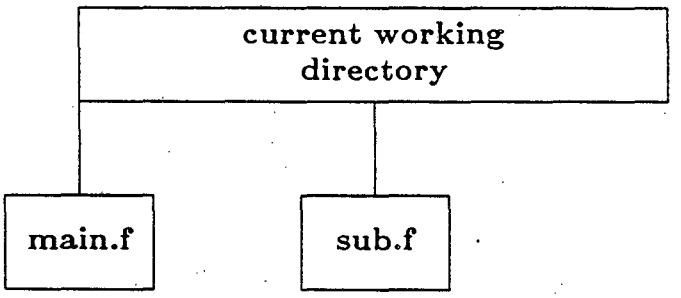
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
My Xanadu checkline.
Down to a sunless sea.*

From Kubla Khan by Samuel Taylor Coleridge.

rcs SETUP

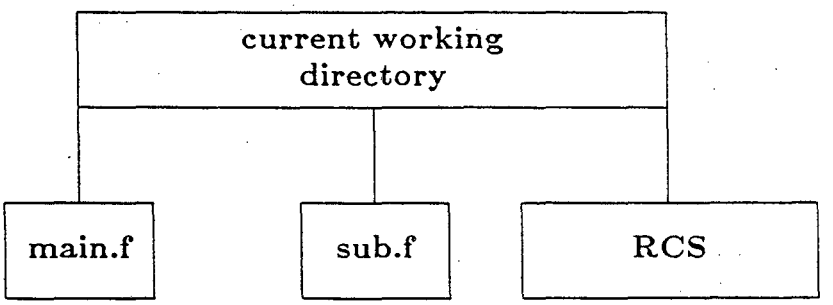
- Generate the original source code modules

```
vi main.f
vi sub.f
```



- Generate a subdirectory named *RCS* for the *rcs* modules

```
mkdir RCS
```



RCS Directory

Create the RCS directory. Your entry:

```
% mkdir RCS
%
```

rcs COMMAND

- Is used to setup or change *rcs* file attributes

- Format

  `rcs` [*options*] *file* ...

- Selected Options:

  -a[*logins*] – appends the *login* names appearing in the comma-separated list *logins* to the access list of the *rcs* list

  -c[*string*] – sets the comment leader to *string*

  -i – creates and initializes a new *rcs* file but does not deposit any revision

  -L – sets locking to *strict*; the owner of a file is not exempt from locking for check in.

  -n*name*[:*rev*] – associates the symbolic name *name* with the branch or revision *rev*

  -o*range* – deletes (outdates) the revisions given by *range*

  -s*state*[:*rev*] – sets the state attribute of the revision *rev* to *state*

  -t[*txtfile*] – writes descriptive text into the *rcs* file (deletes the existing text)

Opening an RCS File

Open the *RCS* file for *poem*. For the description use, **From Kubla Khan by Samuel Taylor Coleridge**. Type this command line:

rcs -i -c'C ' poem

When you have completed the description, exit by typing either **CTRL-d** or period on a new line. Your entry:

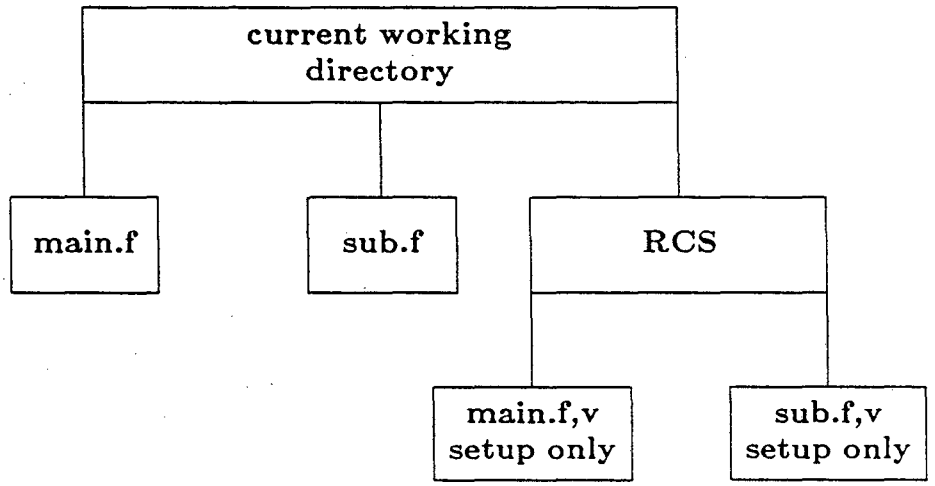
```
% rcs -i -c'C ' poem
RCS file: RCS/poem,v
enter description, terminated with ^D or '.':
NOTE: This is NOT the log message!
>> from Kubla Khan by Samuel Taylor Coleridge
>>
done
%
```

INITIALIZING A SETUP FILE

- Open an *rcs* file for *main.f* and *sub.f*; initialization with *rcs* is optional

Set access list for two program modules

```
rcs -i -asmith,jones main.f
rcs -i -asmith,nelson sub.f
```



Changing RCS File Attributes

Limit the users who can have access to the RCS file *poem*. Allow two of your colleagues to have access. Your entry:

```
% rcs -auser1,user2 poem
RCS file: RCS/poem,v
done
%
```

ci COMMAND

- Is used to check in the original or modified module

- Format

  `ci [`*options*`]` *file* ...

- Selected Options:

  -l[*rev*] – deposits a revision and then immediately checks the revision out and locks it

  -m*msg* – uses the string *msg* as the log message for all revisions checked in

  -n*name* – assigns the symbolic name *name* to the number of the checked-in revision

  -r[*rev*] – assigns the revision number *rev* as the checked-in revision

  -s*state* – sets the *state* of the checked-in revision to *state*; the default is *Exp*

  -t[*textfile*] – writes descriptive text into the *rcs* file (deletes the existing text). If *textfile* is omitted, *ci* prompts the user for text to be supplied.

  -u[*rev*] – deposits a revision and then immediately checks the revision out without locking it

Using the *ci* Command

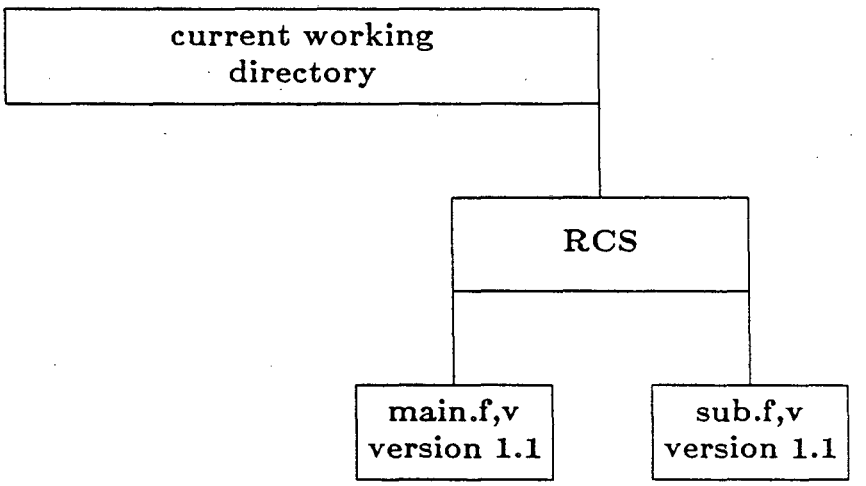
Assume that this first version of the source is not in experimental (alpha) state but is ready for beta testing. When the module is checked in, change the state to *beta*. Your entry:

```
% ci -nbeta poem
RCS/poem,v <-- poem
initial revision: 1.1
done
%
```

CHECKING IN THE ORIGINAL COPY

- Place the original copy of *main.f* and *sub.f* into the *rcs* file (version 1.1):

```
ci main.f
ci sub.f
```



Checking In the Original Source

Assume that you are now ready to keep track of all changes to your document or code. Place the original copy of *poem* (version 1.1) into the *RCS* file. Your entry:

```
% ci poem
RCS/poem,v <-- poem
initial revision: 1.1
done
%
```

co COMMAND

- Is used to check out a module for change
- Format

  co [*options*] *file* ...
- Selected Options:

  -l[*rev*] – locks the checked out revision for the caller

  -p*rev* – prints the retrieved revision on standard output instead of storing it in the working file

  -r*rev* – retrieves the latest revision whose number is less than or equal to *rev*

  -s*state* – retrieves the latest revision on the selected branch whose state is *state*

  -w*login* – retrieves the latest revision on the selected branch which was checked in by the user with a login name of *login*

co Options

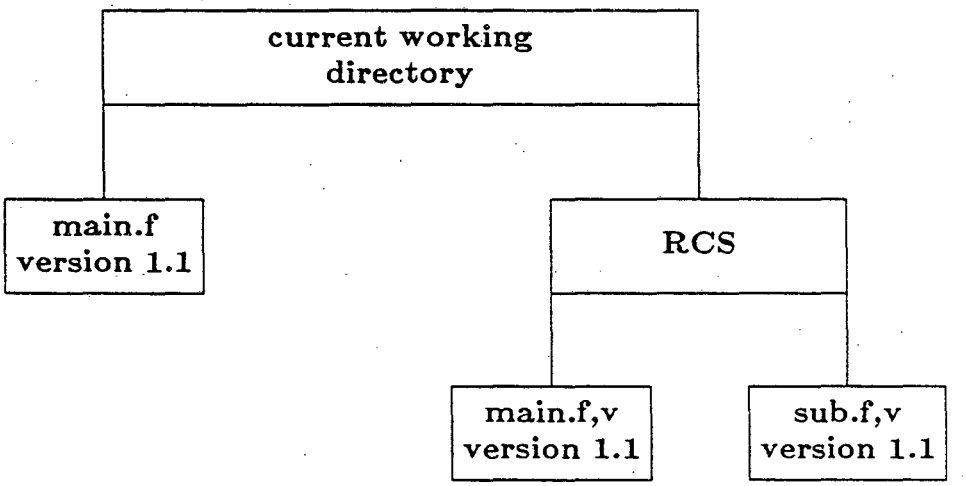
Check out the original version of *poem* and send the output to the line printer. Your entry:

```
% co -p1.1 poem | lpr
RCS/poem,v -> stdout
revision 1.1
%
```

CHECKING OUT THE SOURCE FILE

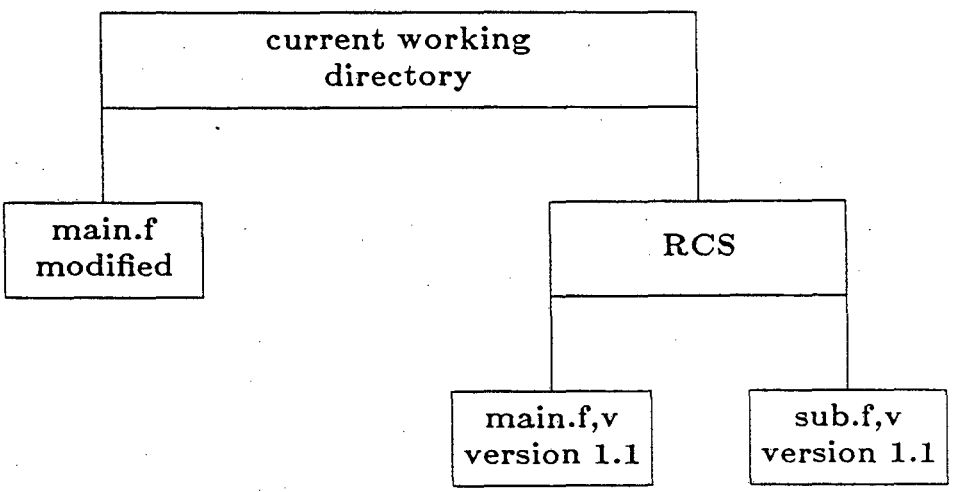
- Check out the original source for *main.f*

```
co main.f
```



- Edit the original source code:

```
vi main.f
```



Checking Out the Original Source

Check out the file *poem* and delete the line that reads: `My Xanadu checkline.`

Your entry:

```
% co poem
RCS/poem,v --> poem
revision 1.1
done
%
```

Now delete the line: `My Xanadu checkline.`

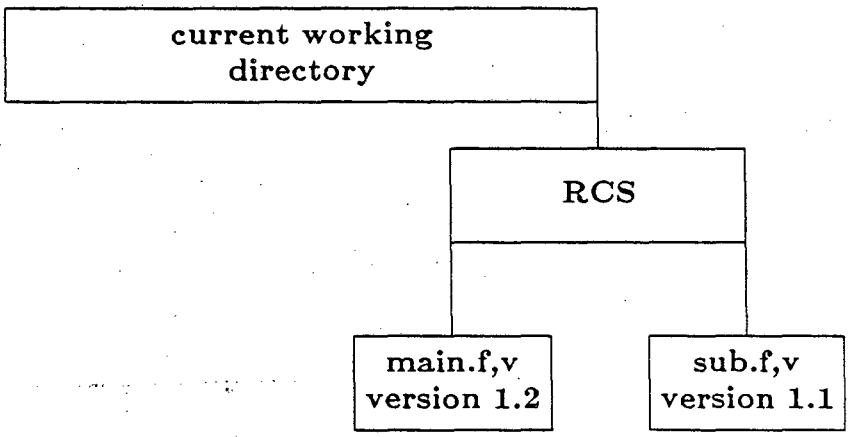
Your entry:

```
% vi poem
"poem" 6 lines, 169 characters
:d5
My Xanadu checkline.
:wq
"poem" 5 lines 146 characters
%
```

MODIFYING THE *rcs* FILE

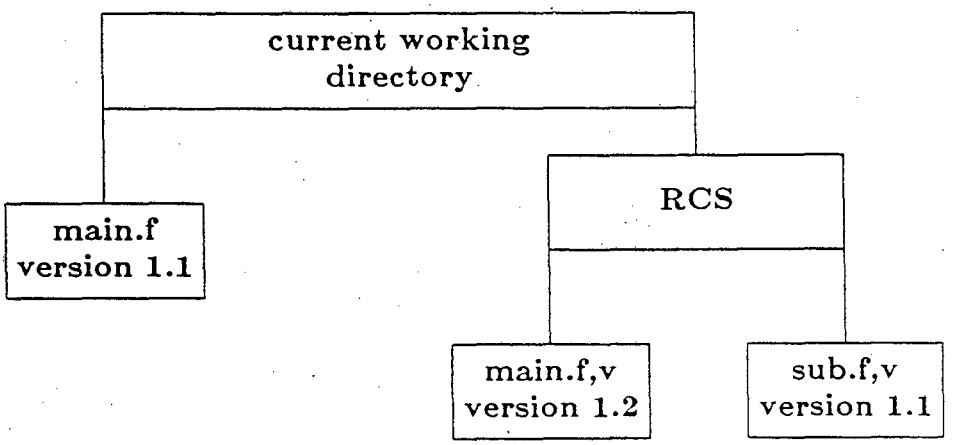
- Check in the modified source code, and document the changes (version 1.2):

```
ci main.f
```



- Check out the original source code:

```
co -r1.1 main.f
```



Checking in the Modified Source

Now check in the file *poem*; use `Deleted line 5` as your log message. Your entry:

```
% ci poem
poem,v <-- poem
new revision: 1.2; previous revision: 1.1
enter log message:
(terminate with ^D or single '.')
>> Deleted line 5.
>> ^D
done
%
```

Now check out the original version of *poem*; its version number is 1.1. Is the original version still intact? Verify that it is by displaying the contents of the file using the *more* command.

Your entry:

```
% co -r1.1 poem
RCS/poem,v --> poem
revision 1.1
done
% more poem
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
My Xanadu checkline.
Down to a sunless sea.
%
```

rlog COMMAND

- Prints log messages and other information about *rcs* files

- Format

  `rlog` [*options*] *file* ...

- When no options are present, the following information is printed:

  - full pathname and working filename of the *rcs* file

  - head (latest revision on the trunk) and total number of revisions

  - access list and locks

  - symbolic names, suffix, and descriptive text

  - number of revisions selected for printing

Printing an RCS Log

Print a log for the *poem*. Your entry looks similar to:

```
% rlog poem1
RCS file:     RCS/poem1,v;  Working file:   poem1
head:        1.5
vlocks:
access list:    user1 user2
symbolic names:  best: 1.3;
comment leader:  "C "
total revisions: 6;   selected revisions: 6
description:
This is a practice file for
learning about RCS.
----------------------------
revision 1.5
date: 86/11/25 16:38:04;  author: user;  state: Exp;  lines added/del: 5/2
*** empty log message ***
----------------------------
revision 1.4
date: 86/11/25 16:37:26;  author: user;  state: Exp;  lines added/del: 3/2
Added the keyword log
----------------------------
revision 1.3
date: 86/11/25 16:36:25;  author: user;  state: Exp;  lines added/del: 3/0
Added two keywords.
----------------------------
revision 1.2
date: 86/11/25 16:29:06;  author: user;  state: Exp;  lines added/del: 0/1
Deleted line 5.
----------------------------
revision 1.1
date: 86/11/25 16:28:11;  author: user;  state: Exp;
vvbranches:  1.1.1;
Initial revision
=============================================================================
%
```

rlog COMMAND (cont.)

- Options:

-d*dates* – prints information about revisions with a check in date/time in the ranges given by the semicolon-separated list of *dates*

-h – prints only *rcs* filename, working filename, head, access list, locks, symbolic name, and suffix.

-l[*lockers*] – prints information about locked revisions

-r*revisions* – prints information about revisions given in a comma-separated list of *revisions*

-s*states* – prints information about revisions whose state attributes match one of the states given in the comma-separated list *states*

-w[*logins*] – prints information about revisions checked in by users with login names appearing in the comma-separated list *logins*

Printing RCS Revision Information

Print log information for revision 1.3 for the file *poem*. Your entry looks similar to:

```
% rlog -r1.3 poem
RCS file:      RCS/poem,v;  Working file:   poem1
head:          1.5
locks:
access list:   user1 user2
symbolic names:  best: 1.3;
comment leader:  "C"
total revisions: 6;   selected revisions: 1
description:
This is a practice file for
learning about RCS.
----------------------------
revision 1.3
date: 86/11/25 16:36:25;  author: user;  state: Exp;  lines added/del: 3/0
Added two keywords.
============================================================================
%
```

Exercises for Chapter 6

1. Explain how you open an *rcs* file.

2. You want to edit a specific version and allow no one else to edit that version until it is released. How would you accomplish this task?

3. What command would you use to print a log of the RCS file, *main.f*?

4. Write a command that would give users *snoopy* and *jones* access to the RCS file *myprog*.

5. Create a file named *sam.rcs* with contents of your choice. After you have created the file, check it in to RCS. Next give users *snoopy* and *jones* access to this file. Modify your file by adding this line: **This is my RCS modification.** Finally, print the log information for *sam.rcs*.

Index

R

S

T

U

V

W

CONVEX UNIX Student Course Materials

Training Course Critique

You are invited to submit your comments concerning the clarity and quality of the training course. Constructive critical comments are most welcome and will help us continue in our efforts to provide quality training.

What part of the training course was most beneficial to you?

_____

_____

_____

_____

What suggestions do you have for improving the course content?

_____

_____

_____

_____

_____

_____

Do you have any comments regarding the instructor, course content, or class environment?

_____

_____

_____

Instructor_____Date_____

Company you represent_____