1. Rewrite the **journal** script of Chapter 9 (exercise 5, page 416) by adding commands to verify that the user has write permission for a file named **journal-file** in the user’s home directory, if such a file exists. The script should take appropriate actions if **journal-file** exists and the user does not have write permission to the file. Verify that the modified script works.

A: **$ cat journal**

# journal: add journal entries to the file

# $HOME/journal-file

file=$HOME/journal-file

if [ $# -w 0 ]

then

date >> $file

echo -n "Enter name of person or group: "

read name

echo "$name" >> $file

echo >> $file

cat >> $file

echo "----------------------------------------------------" >>

$file

echo >> $file

else

echo "You do not have write permission."

exit 1

fi

2. The special parameter "**$@**" is referenced twice in the out script (page 989). Explain what would be different if the parameter "**$\***" were used in its place.

A: If you replace "**$@**" with "**$\***", cat or less would be given a single argument: a list of all files you specified on the command line enclosed within single quotation marks. This list works when you specify a single filename. When you specify more than one file, the shell reports **No such file or directory** because there is not a file whose name matches the string you specified on the command line.

3. Write a filter that takes a list of files as input and outputs the basename (page 1011) of each file in the list.

A: ls –la | basename /home/max/bin/safedit

4. Write a function that takes a single filename as an argument and adds execute permission to the file for the user.

$ **function permi () {**

> **chmod u+x $1**

> **}**

1. When might such a function be useful?

A: When you are writing many shell scripts, it can give chmod commands to all of them and speed up the process.

1. Revise the script so it takes one or more filenames as arguments and adds execute permission for the user for each file argument.

A: $ **function permi () {**

> **chmod u+x $\***

> **}**

1. What can you do to make the function available every time you log in?

A: Put the function in **~/.bash\_profile** or **~/.bashrc** to make it available each time you log in (using bash).

1. Suppose that, in addition to having the function available on subsequent login sessions, you want to make the function available in your current shell. How would you do so?

A: Use source to execute the file you put the function in. For example: $ **source ~/.bash\_profile**

5. When might it be necessary or advisable to write a shell script instead of a shell function? Give as many reasons as you can think of.

A: A shell function will do nothing unless it is explicitly called by other code, typically in a shell script. Hence, shell functions are used for breaking down the required functionality of a code into smaller and logical set of instructions that can be called to perform some specific task as and when needed. On the other hand, a shell script is a runnable, executable process, which can call other shell scripts and/or functions.

6. Write a shell script that displays the names of all directory files, but no other types of files, in the working directory.

A: $ **cat listdirs**

file "$@" |

grep directory |

sed 's/:.\*//'

7. Write a script to display the time every 15 seconds. Read the date man page and display the time, using the %r field descriptor. Clear the window (using the clear command) each time before you display the time.

A: $ cat showtime

#!/bin/bash

While [sleep 15]

do

clear

Time=’date +%r’

echo –n $Time

done

$ ./showtime

8. Enter the following script named **savefiles**, and give yourself execute permission to the file:

**$ cat savefiles**

#! /bin/bash

echo "Saving files in working directory to the file savethem."

exec > savethem

for i in  \*

          do

echo

"==================================================="

         echo "File: $i"

echo

"==================================================="

         cat "$i"

         done

a. Which error message do you receive when you execute this script? Rewrite the script so that the error does not occur, making sure the output still goes to savethem.

A: Error message:

cat: savethem: input file is output file

To eliminate the error message, add the following lines after the line with **do** on it:

if [ $i == "savethem" ] ; then

continue

fi

b. What might be a problem with running this script twice in the same directory? Discuss a solution to this problem.

A: Each time you run **savefiles**, it overwrites the **savethem** file with the current contents of the working directory. When you remove a file and run **savefiles** again, that file will no longer be in **savethem**. If you want to keep an archive of files in the working directory, you need to save the files to a new file each time you run **savefiles**. If you prefix the filename **savethem** with **$$**, you will have a unique filename each time you run **savefiles**.

9. Read the bash man or info page, try some experiments, and answer the following questions:

a. How do you export a function?

A: export -f functionname # exports a function in the current shell.

-f The names refer to shell functions;

b. What does the hash builtin do?

A: Bash uses a hash table to remember the full pathnames of executable files. A full search of the directories in PATH is performed only if the command is not found in the hash table. Built-in commands are contained within the shell itself. When the name of a built-in command is used as the first word of a simple command, the shell executes the command directly, without creating a new process. Built-in commands are necessary to implement functionality impossible or inconvenient to obtain with separate utilities. Hence, hash builtin is to quickly search files.

c. What happens if the argument to exec is not executable?

A: The reason is you have no system executable permission. Hence, the system will show an error message.

10. Using the find utility, perform the following tasks:

a. List all files in the working directory and all subdirectories that have been modified within the last day.

A: $ **find . -mtime -1**

b. List all files you have read access to on the system that are larger than 1 megabyte.

A: $ **find / -size +1024k**

c. Remove all files named **core** from the directory structure rooted at your home directory.

A: $ **find ~ -name core -exec rm {} \;**

d. List the inode numbers of all files in the working directory whose filenames end in **.c**.

A: $ **find . -name "\*.c" -ls**

e. List all files you have read access to on the root filesystem that have been modified in the last 30 days.

A: $ **find / -xdev -mtime -30**