Answers to Even-Numbered Exercises 12

from page 620

1. The following shell script adds entries to a file named **journal-file** in your home directory. The script can help you keep track of phone conversations and meetings:

```
$ cat journal
# journal: add journal entries to the file
# $HOME/journal-file

file=$HOME/journal-file
date >> $file
echo -n "Enter name of person or group: "
read name
echo "$name" >> $file
echo >> $file
cat >> $file
echo "------------------------------------------------------------" >> $file
echo >> $file
```

   a. What do you have to do to the script in order to be able to execute it?

   b. Why does it use the read builtin the first time it accepts input from the terminal and the cat utility the second time?

2. What are two ways you can execute a shell script when you do not have execute access permission to the file containing the script? Can you execute a shell script if you do not have read access permission?

You can give the name of the file containing the script as an argument to the shell (for example, **sh** ***scriptfile*** or **tcsh** ***scriptfile,*** where ***scriptfile*** is the name of the file containing the script).

Under bash you can give the following command:

```
$ . scriptfile
```

Under tcsh you can use

```
$ source scriptfile
```

Because the shell must read the commands from the file containing a shell script before it can execute the commands, you must have read permission for the file in order to execute the shell script.

3. What is the purpose of the **PATH** variable?

   a. Set up your **PATH** variable so that it causes the shell to search the following directories in order:

   - **/usr/local/bin**
   - **/usr/bin/X11**
   - **/usr/bin**
   - **/bin**
   - **/usr/openwin/bin**
   - Your own **bin** directory (usually **bin** or **.bin** in your home directory)
   - The working directory

   b. If a file named **whereis** is in **/usr/bin** and also in your **~/bin**, which one does **whereis** indicate will be executed? (Assume that you have execute permission for both of the files.)

   c. If your **PATH** variable is not set to search the working directory, how can you execute a program located there?

   d. What command can you use to add the directory **/usr/games** to the end of the list of directories in **PATH**?

4. Assume that you have made the following assignment:

   ```
   $ person=jenny
   ```

   Give the output of each of the following commands:

   a. **echo $person**

   jenny

b. **echo '$person'**

$person

c. echo "$person"

jenny

5. Explain the unexpected following result:

```
$ whereis date
date: /bin/date
$ echo $PATH
.:/usr/local/bin:/usr/bin:/bin
$ cat > date
echo "This is my own version of date."
$ date
Wed Mar 12 09:11:54 MST 2003
```

6. Assume that the **/home/jenny/grants/biblios** and **/home/jenny/biblios** directories exist. For both (a) and (b), give Jenny's working directory after she executes the sequence of commands given. Explain.

   a.

```
$ pwd
/home/jenny/grants
$ CDPATH=$(pwd)
$ cd
$ cd biblios
```

   After executing the preceding commands, Jenny's working directory is **/home/jenny/grants/biblios**.

   When **CDPATH** is set, `cd` searches only the directories specified by **CDPATH**, `cd` does not search the working directory unless the working directory is specified in **CDPATH** (by using a period).

   b.

```
$ pwd
/home/jenny/grants
CDPATH=$(pwd)
$ cd $HOME/biblios
```

   After executing the preceding commands, Jenny's working directory is **/home/jenny/biblios** because, when you give `cd` an absolute pathname as an argument, `cd` does not use **CDPATH**.

7. Name two ways you can identify the PID of your login shell.

8. Try giving the following command:

```
$ sleep 30 | cat /etc/inittab
```

Is there any output from sleep? Where does cat get its input from? What has to happen before you get a prompt back?

There is no output from sleep (try giving the command **sleep 30** by itself). The **/etc/inittab** file provides input for cat (when cat has an argument, it does not check its standard input). The sleep command has to run to completion before you get another prompt.

Advanced Exercises

9. Write a sequence of commands or a script that demonstrates that parameter expansion occurs before variable expansion and that variable expansion occurs before pathname expansion.

10. Write a shell script that outputs the name of the shell that is executing it.

    There are many ways to solve this problem. The following solutions are all basically the same. These scripts take advantage of the **PPID** shell variable which holds the **PID** of the shell that is the parent of the process using the variable and of the fact that echo changes multiple sequential SPACEs to a single SPACE. The cut utility interprets multiple sequential SPACEs as multiple delimiters so, without echo, the script does not work properly.

```
$ cat a
pid=$PPID
line=$(ps | grep $pid)
echo $line | cut --delimiter=" " --fields=4

$ cat a2
pid=$PPID
echo $(ps | grep $pid) | cut --delimiter=" " --fields=4

$ cat a3
echo $(ps | grep $PPID) | cut --delimiter=" " --fields=4
```

    The easy solution is to put the following line in the shell script:

```
echo $0
```

    The **$0** is the first command line token, which is usually the name of the script that is running. In some cases, such as when you call the script with a relative or absolute pathname, this may not be exactly what you want.

11. Type in the following shell scripts and run them:

```
$ cat report_dir
old_dir=$(pwd)
echo "Current working directory:  " $old_dir
go_home
echo "Current working directory:  " $(pwd)

$ cat go_home
cd
echo "New working directory:  " $(pwd)
echo "Last working directory: " $old_dir
```

What is wrong? Change the scripts so that each of the echo commands displays the proper value.

12. The following is a modified version of the **read2** script from page 574. Explain why it behaves differently. For what type of input does it produce the same output of the original **read2** script?

```
$ cat read2
echo -n "Enter a command: "
read command
"$command"
echo "Thanks"
```

In the preceding **read2** script, the string `$command` is surrounded by double quotation marks, causing it to pass a single argument to the shell. The script in the text does not surround the string with quotation marks and therefore passes the shell as many arguments as the user enters. The result is that, given single word commands (commands without arguments), both versions of the script produce the same results.

When you respond to the prompt issued by the preceding command with **echo hi there**, the script generates an error because the script passes the string `echo hi there` as the command to execute; the SPACEs are not special characters that separate words because they are quoted. You get the same error message when you give the following command:

```
$ echo\ hi\ there
```

13. Explain the behavior of the following shell script:

```
$ cat quote_demo
twoliner="This is line 1.
This is line 2."
echo "$twoliner"
echo $twoliner
```

a. How many arguments does each echo command see in this script? Explain.

b. Redefine the **IFS** shell variable so that the output of the second echo is the same as the first.