

# UNIX/LINUX FILE PROCESSING

**After reading this chapter and completing the exercises, you will be able to:**

- ◆ Explain UNIX and Linux file processing
- ◆ Use basic file manipulation commands to create, delete, copy, and move files and directories
- ◆ Employ commands to combine, cut, paste, rearrange, and sort information in files
- ◆ Create a script file
- ◆ Use the *join* command to link files using a common field
- ◆ Use the *awk* command to create a professional-looking report

The power of UNIX/Linux is based on its storage and handling of files. You've already learned about file systems, security, and UNIX/Linux editors. Now it's time to put your knowledge to work by manipulating files and their contents. To give you some background, this chapter starts with a short discussion of UNIX/Linux file types and file structures. Next, you learn more about using redirection operators, including how to use them to store error messages. You go on to learn file manipulation tools that you will use over and over again, either as a UNIX/Linux administrator or as an everyday user. These essential tools enable you to create, delete, copy, and move files. Other tools enable you to extract information from files to combine fields and to sort a file's contents. Finally, you learn how to assemble the information you extract from files, such as for creating reports. You also create your first script to automate a series of commands, link files with a common field, and get a first taste of the versatile *awk* command to format output.

## UNIX AND LINUX FILE PROCESSING

UNIX/Linux file processing is based on the idea that files should be treated as nothing more than character sequences. This concept of a file as a series of characters offers a lot of flexibility. Because you can directly access each character, you can perform a range of editing tasks, such as correcting spelling errors and organizing information to meet your needs.

### Reviewing UNIX/Linux File Types

Operating systems support several types of files. UNIX and Linux, like other operating systems, have text files, binary files, directories, and special files. As discussed in Chapter 3, “Mastering Editors,” text files contain printable ASCII characters. Some users also call these regular, ordinary, or ASCII files. Text files often contain information you create and manipulate, such as a document or program source code. Binary files, also discussed in Chapter 3, contain nonprintable characters, including machine language code created from compiling a program. In UNIX/Linux, text files and binary files are considered to be **regular files**, and you will sometimes see this terminology when working with files at the command line.

Chapter 2, “Exploring the UNIX/Linux File Systems and File Security,” explained that directories are system files for maintaining the structure of the file system. In Chapter 2, you also learned about device special files. Character special files are used by input/output devices for communicating one character at a time, providing what is called raw data. The first character in the file access permissions is “c,” which represents the file type, a character special file. Block special files are also related to devices, such as disks, and send information using blocks of data. The first character in these files is “b.” For comparison, as you learned in Chapter 2, the first character for a directory is “d,” and for a normal file—not a device special file—the first character is a dash “-.”



**NOTE**

Character special and block special files might also be called character device and block device files or character-special device and block-special device files.

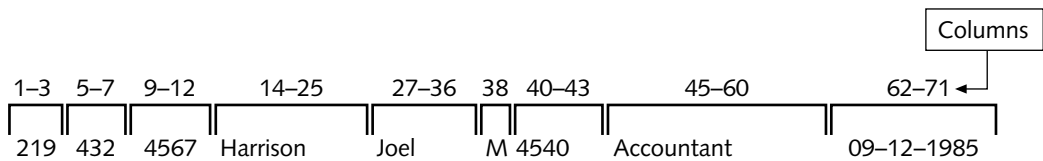
### Understanding File Structures

Files can be structured in several ways. For example, UNIX/Linux store data, such as letters, product records, or vendor reports, in **flat ASCII files**. The internal structure of a file depends on the kind of data it stores. A user structures a letter, for instance, using words, paragraphs, and sentences. A programmer can structure a file containing employee records using characters and words grouped together, with each individual employee record on a separate line in a file. Information about an employee in each separate line or record can be divided by separator characters or delimiters, such as colons. This type of record is called a **variable-length record**, because the length of each field bounded by colons can vary. The following is a simple example of an employee telephone record that might be stored in a flat

ASCII file and used by a human resources program. The first three fields, separated by colons, are the employee's home telephone number, consisting of the area code, prefix, and number. A human resources professional or boss might display some or all of this information in a program or report to be able to call the employee at home.

219:432:4567:Harrison:Joel:M:4540:Accountant:09-12-1985

Another way to create records is to have them start and stop in particular columns. For example, the area code in the previous example might start in column 1 and end in column 3. The prefix in the telephone number might go from column 5 to column 7, the last four digits in the telephone number would be in columns 9 through 12, and so on. Figure 4-1 illustrates this type of record, which is called a **fixed-length record**.



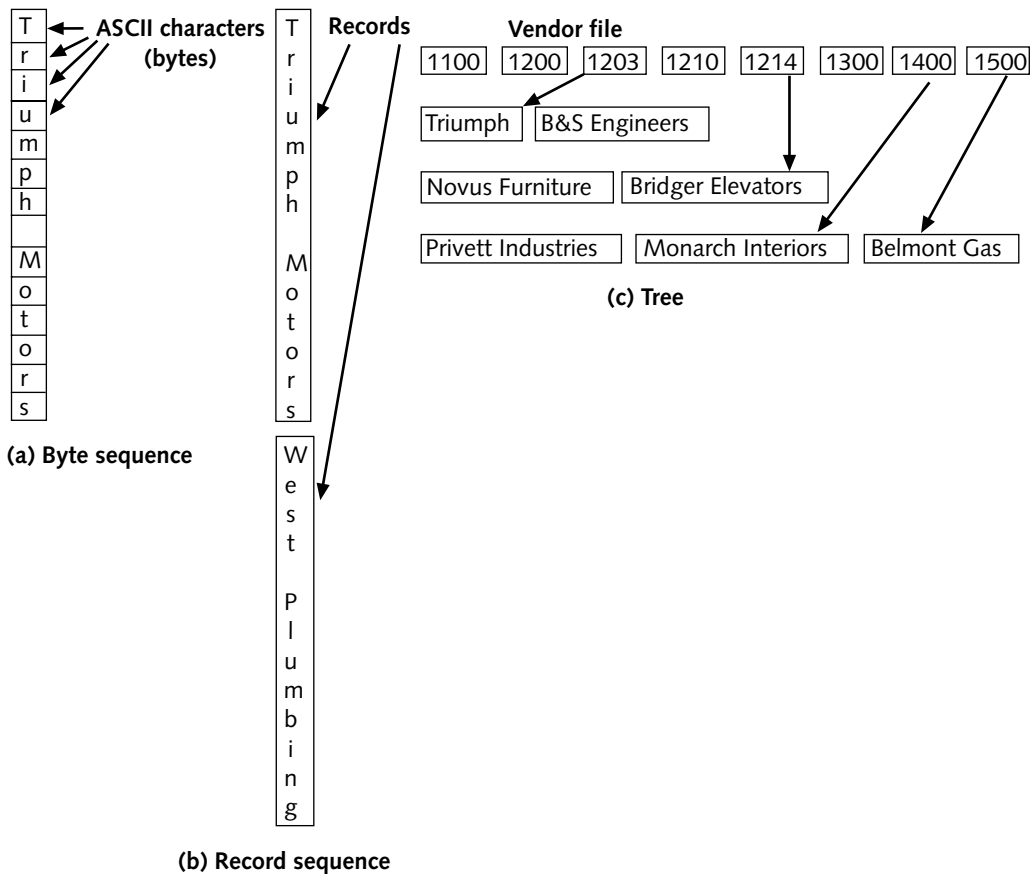
**Figure 4-1** Fixed-length record

Three simple kinds of text files are unstructured ASCII characters, records, and trees. Figure 4-2 illustrates these three kinds of text files.

Figure 4-2(a) shows a file that is an unstructured sequence of bytes and is a typical example of a simple text file. This file structure gives you the most flexibility in data entry, because you can store any kind of data in any order, such as the vendor name Triumph Motors and other information related to Triumph Motors, which might be a unique vendor number, the vendor's address, and so on. However, you can only retrieve the data in the same order, which might limit its overall usefulness. For example, suppose you list the vendors (product suppliers) used by a hotel in an unstructured ASCII text file. In this format, if you want to view only vendor names or vendor numbers, you really don't have that option. You most likely will have to print the entire file contents, including address and other information for all vendors.

Figure 4-2(b) shows data as a sequence of fixed-length records, each having some internal structure. In a UNIX/Linux file, a record is a line of data, corresponding to a row. For example, in a file of names, the first line or row might contain information about a single individual, such as last name, first name, middle initial, address, and phone number. The second row would contain the same kind of data about a different person, and so on. In this structure, UNIX/Linux read the data as fixed-length records. Although you must enter data as records, you can also manipulate and retrieve the data as records. For example, you can select only certain personnel or vendor records to retrieve from the file.

The third kind of file, illustrated in Figure 4-2(c), is structured as a tree of records that can be organized as fixed-length or variable-length records. In Figure 4-2(c), each record contains a key field, such as a record number, in a specific position in the record. The key



**Figure 4-2** Three kinds of text files

field sorts the tree, so you can quickly search for a record with a particular key. For example, you can quickly find the record for Triumph Motors by searching for record #1203.

You will practice creating and manipulating different kinds of files and records in this and later chapters.

## PROCESSING FILES

When performing commands, UNIX/Linux process data by receiving input from the standard input device—your keyboard, for example—and then sending it to the standard output: the monitor or console. System administrators and programmers refer to standard input as **stdin**. They refer to standard output as **stdout**. The third standard device, or file, is called standard error, or **stderr**. When UNIX/Linux detect errors in processing system tasks and user programs, they direct the errors to **stderr**, which, by default, is the screen.



stdin, stdout, and stderr are defined in IEEE Std 1003.1, “Standard for Information Technology—Portable Operating System Interface (POSIX),” and the ISO 9899:1999 C language standard (for C programming). The Institute of Electrical and Electronics Engineers (IEEE) and the International Organization for Standardization (ISO) set computer-based and other standards.

In Chapter 1, “The Essence of UNIX and Linux,” you learned about the `>` and `>>` output redirection operators. You can use these and other redirection operators to save the output of a command or program in a file or use a file as an input to a process. The redirection operators are a tool to help you process files.

4

## Using Input and Error Redirection

You can use redirection operators (`>`, `>>`, `2>`, `<`, and `<<`) to retrieve input from something other than the standard input device and to send output to something other than the standard output device.

You already used the output redirection operators in Chapter 1, when you created a new file by redirecting the output of several commands to files. Redirect output when you want to store the output of a command or program in a file. For example, recall that you can use the `ls` command to list the files in a directory, such as `/home`. The `ls` command sends output to stdout, which, by default, is the screen. To redirect the list to a file called `homedir.list`, use the redirection symbol by entering `ls > homedir.list`.

You can also redirect the input to a program or command with the `<` operator. For instance, a program that accepts keyboard input can be redirected to read information from a file instead. In Hands-On Project 4-1, you create a file from which the `vi` editor reads its commands, instead of reading them from the keyboard.

You can also use the `2>` operator to redirect commands or program error messages from the screen to a file. For example, if you try to list a file or directory that does not exist, you see the following error message: `No such file or directory`. Assume that `Fellowese` is not a file or directory. If you enter `ls Fellowese 2> errors`, this places the `No such file or directory` error message in the `errors` file. Try this redirection technique in Hands-On Project 4-2.

---

## MANIPULATING FILES

When you manipulate files, you work with the files themselves as well as their contents. This section explains how to complete the following tasks:

- Create files
- Delete files
- Remove directories
- Copy files

- Move files
- Find files
- Combine files
- Combine files through pasting
- Extract fields in files through cutting
- Sort files

## Creating Files

You can create a new file by using the output redirection operator (`>`). You learned how to do this to redirect the `cat` command's output in Chapters 1–3. You can also use the redirection operator without a command to create an empty file by entering `>` and the name of the file. For example, the following command:

```
> accountsfle
```

creates an empty file called `accountsfile`. Hands-On Project 4-3 enables you to create a file using the `>` redirection symbol.

You can also use the `touch` command to create empty files. For example, the following command creates the file `accountsfile2`, if the file does not already exist.

```
touch accountsfile2
```

---

**Syntax** `touch` `[-options]` *filename(s)*

---

### Dissection

- Intended to change the time stamp on a file, but can also be used to create an empty file
  - Useful options include:
    - a* updates the access time only
    - m* updates the last time the file was modified
    - c* prevents the `touch` command from creating the file, if it does not already exist
- 



**TIP**

To view time stamp information in full, use the `--full-time` option with the `ls` command, such as `ls --full-time myfile`.

The primary purpose of the `touch` command is to change a file's time stamp and date stamp. UNIX/Linux maintain the following date and time information for every file:

- *Change date and time*—The date and time the file's inode was last changed

- *Access date and time*—The date and time the file was last accessed
- *Modification date and time*—The date and time the file was last modified



Recall from Chapter 2 that an inode is a system for storing key information about files. Inode information includes the inode number, the owner of the file, the file group, the file size, the change date of the inode, the file creation date, the date the file was last modified and last read, the number of links to this inode, and the information regarding the location of the blocks in the file system in which the file is stored.

Although the *touch* command cannot alter a file's inode changed date and time, it can alter the file's access and modification dates and times. By default, it uses the current date and time for the new values. Hands-On Project 4-4 gives you experience using the *touch* command.

## Deleting Files

When you no longer need a file, you can delete it using the *rm* (remove) command. If you use *rm* without options, UNIX/Linux delete the specified file without warning. Use the *-i* (interactive) option to have UNIX/Linux warn you before deleting the file. You can delete several files with similar names by using the asterisk wildcard. (See Chapter 2.) For example, if you have 10 files that all begin with the letters “test,” enter *rm test\** to delete all of them at one time. Hands-On Project 4-5 enables you to use the *rm* command.

---

**Syntax** *rm* [-options] *filename* or *directoryname*

---

### Dissection

- Used to delete files or directories
  - Useful options include:
    - i displays a warning prompt before deleting the file (or directory)
    - r when deleting a directory, recursively deletes its files and subdirectories (to delete a directory that is empty or that contains entries, use the *-r* option with *rm*)
- 

## Removing Directories

When you no longer need a directory, you can use the commands *rm* or *rmdir* to remove it. For example, if the directory is already empty, use *rm -r* or *rmdir*. If the directory contains files or subdirectories, use *rm -r* to delete them all. The *rm* command with the *-r* option removes a directory and everything it contains. It even removes subdirectories of subdirectories. This operation is known as recursive removal. Note that if you use *rm* alone, in many versions of UNIX/Linux, including Fedora, Red Hat Enterprise Linux, and SUSE, it does not delete a directory.

Hands-On Project 4-6 enables you to use *rmdir* to delete an empty directory and *rm -r* to delete a directory that is not empty.

---

**Syntax** *rmdir* [-options] *directoryname*

---

### Dissection

- Used to delete directories
  - A directory must be empty to delete it with the *rmdir* command.
- 



Use *rm -r* with great care by first making certain you have examined all of the directory's contents and intend to delete them along with the directory. If you are just deleting an empty directory, it is safer to use the *rmdir* command in case you make a typo when you enter the name of the directory. Also, when you use *rm* with the *-r* option, consider using the *-i* option as well to prompt you before you delete. Additional precautions employed by some users are to (1) use *pwd* to make certain you are in the proper working directory before you delete another directory and (2) use the full path to the directory you plan to delete, because, if you mistype a name, the deletion is likely to fail rather than delete the wrong directory.

## Copying Files

In Chapter 2, you were introduced to the *cp* command for copying files, which we explore further here. Its general form is as follows:

---

**Syntax** *cp* [-options] *source destination*

---

### Dissection

- Used to copy files or directories
  - Useful options include:
    - i provides a warning before *cp* writes over an existing file with the same name
    - s creates a symbolic link or name at the destination rather than a physical file (a symbolic name is a pointer to the original file, which you learn about in Chapter 6)
    - u prevents *cp* from copying over an existing file if the existing file is newer than the source file
- 

The *cp* command copies the file or files specified by the source path to the location specified by the destination path. You can copy files into another directory, with the copies keeping the same names as the originals. You can also copy files into another directory, with the copies taking new names, or copy files into the same directory as the originals, with the copies taking new names.



For example, assume Tom is in his home directory (`/home/tom`). In this directory, he has the file `reminder`. Under his home directory, he has another directory, `duplicates` (`/home/tom/duplicates`). He copies the `reminder` file to the `duplicates` directory with the following command:

```
cp reminder duplicates
```

After he executes the command, a file named `reminder` is in the `duplicates` directory. It is a duplicate of the `reminder` file in the `/home/tom` directory. Tom also has the file `class_of_88` in his home directory. He copies it to a file named `classmates` in the `duplicates` directory with the following command:

```
cp class_of_88 duplicates/classmates
```

After he executes the command, the file `classmates` is stored in the `duplicates` directory. Although it has a different name, it is a copy of the `class_of_88` file. Tom also has a file named `memo_to_boss` in his home directory. He wants to make a copy of it and keep the copy in his home directory. He enters the following command:

```
cp memo_to_boss memo.safe
```

After he executes this command, the file `memo.safe` is stored in Tom's home directory. It is a copy of his `memo_to_boss` file.

You can specify multiple source files as arguments to the `cp` command. For example, Tom wants to copy the files `project1`, `project2`, and `project3` to his `duplicates` directory. He enters the following command:

```
cp project1 project2 project3 duplicates
```



**TIP**

The final entry in a multiple copy (`cp`) or move (`mv`) is a directory, as in the preceding example. You learn about the move command in the next section.

After he executes the command, copies of the three files are stored in the `duplicates` directory.

You can also use wildcard characters with the `cp` command. For example, Tom has a directory named `designs` under his home directory (`/home/tom/designs`). He wants to copy all files in the `designs` directory to the `duplicates` directory. He enters the following command:

```
cp designs/* duplicates
```

After he executes this command, the `duplicates` directory contains a copy of every file in the `designs` directory. As this example illustrates, the `cp` command is useful not only for copying but also for preventing lost data by making backup copies of files. You use the `cp` command in Hands-On Project 4-7 to make copies in a backup directory.

## Moving Files

Moving files is similar to copying them, except you remove them from one directory and store them in another. However, as insurance, a file is copied before it is moved. To move a file, use the *mv* (move) command along with the source file name and destination name. You can also use the *mv* command to rename a file by moving one file into another file with a different name.



### TIP

Moving and renaming a file are essentially the same operation.

When you are moving files, using the *-i* option with the *mv* command can be a good idea so that you don't unexpectedly overwrite a destination file with the same name.

---

**Syntax** *mv* [-options] *source destination*

---

### Dissection

- Used to move and to rename files
  - Useful options include:
    - i displays a warning prompt before overwriting a file with the same name
    - u overwrites a destination file with the same name, if the source file is newer than the one in the destination
- 

Hands-On Project 4-8 enables you to use the *mv* command.

## Finding Files

Sometimes, you might not remember the specific location of a file you want to access. The *find* command searches for files that have a specified name. Use the *find* command to locate files that have the same name or to find a file in any directory.

---

**Syntax** `find [pathname ] [-name filename ]`

---

### Dissection

- Used to locate files in a directory and in subdirectories
- Useful options include:

*pathname* is the path name of the directory you want to search. The *find* command searches recursively; that is, it starts in the named directory and searches down through all files and subdirectories under the directory specified by *pathname*.

*-name* indicates that you are searching for files with a specific *filename*. You can use wildcard characters in the file name. For example, you can use *phone\** to search for all file names that begin with “phone.”

*-iname* works like *-name*, but ignores case. For example, if you search for *phone\** as the search name, you’ll find all files that begin with “phone,” “Phone,” “PHONE,” or any combination of upper and lowercase letters.

*-mmin n* displays files that have been changed within the last *n* minutes.

*-mtime n* displays files that have been changed within the last *n* days.

*-size n* displays files of size *n*, where the default measure for *n* is in 512-byte blocks (you can also use *nc*, *nk*, *nM*, or *nG* for bytes kilobytes, megabytes, or gigabytes, such as *find -size 2M* to find files that are 2 megabytes). For other search conditions you can use with *find*, refer to Appendix B, “Syntax Guide to UNIX/Linux Commands.”

---

When you are using the *find* command, you can only search areas for which you have adequate permissions. As the search progresses, the *find* command might enter protected directories; you receive a “Permission denied” message each time you attempt to enter a directory for which you do not have adequate permissions. Also, when you use *find*, it is useful to note that some UNIX versions require the *-print* option after the file name to display the names of files.

Try Hands-On Project 4-9 to use the *find* command.

## Combining Files

In addition to viewing and creating files, you can use the *cat* command to combine files. You combine files by using a redirection operator, but in a somewhat different format than you use to create a file. As you already know, if you enter *cat > janes\_research*, you can then type information into the file and end the session by typing Ctrl+d, creating the file *janes\_research*. Assume that Jane has created such a file containing research results about bighorn

sheep. Now assume that there is also the file `marks_research`, which contains Mark's research on the same topic. You can use the `cat` command to combine the contents of both files into the `total_research` file by entering the following:

```
cat janes_research marks_research > total_research
```

Hands-On Project 4-10 enables you to use this technique for combining files.

## Combining Files with the `paste` Command

The `paste` command combines files side by side, whereas the `cat` command combines files end to end. When you use `paste` to combine two files into a third file, the first line of the output contains the first line of the first file followed by the first line of the second file. For example, consider a simple file, called `vegetables`, containing the following four lines:

```
Carrots  
Spinach  
Lettuce  
Beans
```

Also, the `bread` file contains the following four lines:

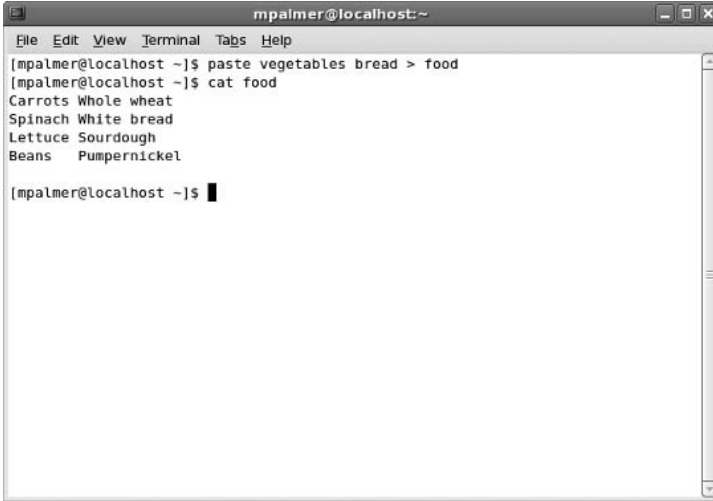
```
Whole wheat  
White bread  
Sourdough  
Pumpernickel
```

If you execute the command `paste vegetables bread > food`, the `vegetables` and `bread` files are combined, line by line, into the file `food`. The `food` file's contents are shown in Figure 4-3.



**TIP**

The `paste` command normally sends its output to `stdout` (the screen). To capture it in a file, use the redirection operator.

A terminal window titled 'mpalmer@localhost:~' with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal shows the following commands and output:

```
[mpalmer@localhost ~]$ paste vegetables bread > food
[mpalmer@localhost ~]$ cat food
Carrots Whole wheat
Spinach White bread
Lettuce Sourdough
Beans Pumpernickel
[mpalmer@localhost ~]$
```

**Figure 4-3** Using the *paste* command to merge files

---

**Syntax** `paste` [-options] *source files* [*> destination file*]

---

### *Dissection*

- Combines the contents of one or more files to output to the screen or to another file
  - By default, the pasted results appear in columns separated by tabs
  - Useful options include:
    - d enables you to specify a different separator (other than a tab) between columns
    - s causes files to be pasted one after the other instead of in parallel
- 

As you can see, the *paste* command is most useful when you combine files that contain columns of information. When *paste* combines items into a single line, it separates them with a tab. For example, look at the first line of the food file:

```
Carrots Whole wheat
```

When *paste* combined “Carrots” and “Whole wheat,” it inserted a tab between them. You can use the *-d* option to specify another character as a delimiter. For example, to insert a comma between the output fields instead of a tab, you would enter:

```
paste -d',' vegetables bread > food
```

After this command executes, the food file’s contents are:

```
Carrots,Whole wheat
Spinach,White bread
```

Lettuce, Sourdough  
Beans, Pumpernickel

Try Hands-On Project 4-11 to begin learning to use the *paste* command.

## Extracting Fields Using the cut Command

You have learned that files can consist of records, fields, and characters. In some instances, you might want to retrieve some, but not all, fields in a file. Use the *cut* command to remove specific columns or fields from a file. For example, in your organization, you might have a vendors file of businesses from which you purchase supplies. The file contains a record for each vendor, and each record contains the vendor's name, street address, city, state, zip code, and telephone number; for example, Office Supplies: 2405 S.E. 17th Street: Boulder: Colorado: 80302:303-442-8800. You can use the *cut* command to quickly list only the names of vendors, such as Office Supplies, in this file. The syntax of the *cut* command is as follows:

---

**Syntax** *cut* [-f *list*] [-d *char*] [*file1 file2 . . .*] or *cut* [-c *list*] [*file1 file2 . . .*]

---

### Dissection

- Removes specific columns or fields from a file
  - Useful options include:
    - f specifies that you are referring to fields  
*list* is a comma-separated list or a hyphen-separated range of integers that specifies the field. For example, -f 1 indicates field 1, -f 1,14 indicates fields 1 and 14, and -f 1-14 indicates fields 1 through 14.
    - d indicates that a specific character separates the fields  
*char* is the character used as the field separator (delimiter), for example, a comma. The default field delimiter is the tab character.
    - file1, file2* are the files from which you want to cut columns or fields
    - c references character positions. For example, -c 1 specifies the first character and -c 1,14 specifies characters 1 and 14.
- 

Recall the example vegetables and bread files discussed in the preceding section. Assume that you also have the file meats. When you use the command *paste vegetables bread meats > food*, the contents of the food file are now as follows:

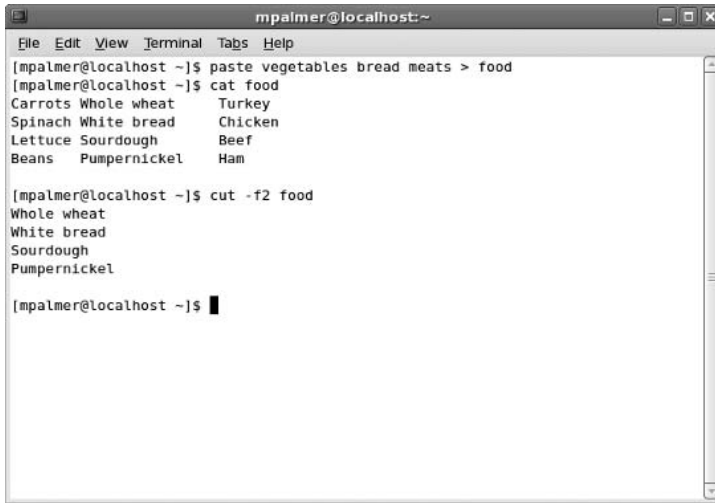
Carrots	Whole wheat	Turkey
Spinach	White bread	Chicken
Lettuce	Sourdough	Beef
Beans	Pumpernickel	Ham

Next, assume that you want to extract the second column of information (the bread list) from the file and display it on the screen. You enter the following command:

```
cut -f2 food
```

The option `-f2` tells the `cut` command to extract the second field from each line. Tab delimiters separate the fields, so `cut` knows where to find the fields. The result of the `cut -f2 food` command is output to the screen, as shown in Figure 4-4.

4



```

mpalmer@localhost:~
File Edit View Terminal Tabs Help
[mpalmer@localhost ~]$ paste vegetables bread meats > food
[mpalmer@localhost ~]$ cat food
Carrots Whole wheat Turkey
Spinach White bread Chicken
Lettuce Sourdough Beef
Beans Pumpernickel Ham

[mpalmer@localhost ~]$ cut -f2 food
Whole wheat
White bread
Sourdough
Pumpernickel

[mpalmer@localhost ~]$

```

**Figure 4-4** Using the `cut` command

Another option is to extract the first and third columns from the file using the following command:

```
cut -f1,3 food
```

The result of the command is:

```

Carrots Turkey
Spinach Chicken
Lettuce Beef
Beans Ham

```

Hands-On Project 4-12 enables you to practice using the `cut` command.

## Sorting Files

Use the `sort` command to sort a file's contents alphabetically or numerically. UNIX/Linux display the sorted file on the screen by default, but you can specify that you want to store the sorted data in a particular file by using a redirection operator.

---

**Syntax** `sort` [-options] [*filename* ]

---

### Dissection

- Sorts the contents of files by individual lines
  - Useful options include:
    - k n* sorts on the key field specified by *n*
    - t* indicates that a specified character separates the fields
    - m* merges input files that have been previously sorted (does not perform a sort)
    - o* redirects output to the specified file
    - d* sorts in alphanumeric or dictionary order
    - g* sorts by numeric (general) order
    - r* sorts in reverse order
- 

The `sort` command offers many options, which Appendix B also describes. The following is an example of its use:

```
sort file1 > file2
```

In this example, the contents of `file1` are sorted and the results are stored in `file2`. (If the output is not redirected, `sort` displays its results on the screen.) If you are sorting a file of records and specify no options, for instance, the values in the first field of each record are sorted alphanumerically. A more complex example is as follows:

```
sort -k 3 food > sortedfood
```

This command specifies a **sorting key**. A sorting key is a field position within each line. The `sort` command sorts the lines based on the sorting key. The `-k` is the key field within the file. For instance, `-k 3` in the preceding example sorts on the third field in the `food` file, which is the listing of meats, and writes the results of the sort to the file `sortedfood` (see Figure 4-5). Notice in Figure 4-5 that the third field in the first record is Beef (and all of the records are sorted by the third field).

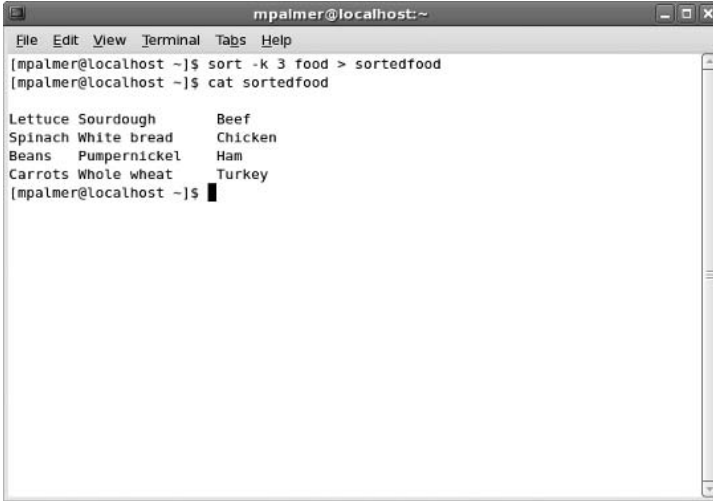
Hands-On Project 4-13 enables you to use the `sort` command. Also, try Hands-On Project 4-14 to use the `cat`, `cut`, `paste`, and `sort` commands in a project that puts together in one place what you have learned so far.

---

## CREATING SCRIPT FILES

As you have seen, command-line entries can become long, depending on the number of options you need to use. You can use the shell's command-line history retrieval feature to recall and reexecute past commands. This feature can work well for you if you need to repeat commands shortly after executing them, but it is a problem if you need to perform



A terminal window titled 'mpalmer@localhost:~' with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal shows the command '[mpalmer@localhost ~]\$ sort -k 3 food > sortedfood' followed by '[mpalmer@localhost ~]\$ cat sortedfood'. The output is a two-column list of food items: Lettuce Sourdough, Spinach White bread, Beans Pumpernickel, Carrots Whole wheat, Beef, Chicken, Ham, and Turkey.

```
mpalmer@localhost:~  
File Edit View Terminal Tabs Help  
[mpalmer@localhost ~]$ sort -k 3 food > sortedfood  
[mpalmer@localhost ~]$ cat sortedfood  
  
Lettuce Sourdough      Beef  
Spinach White bread    Chicken  
Beans Pumpernickel     Ham  
Carrots Whole wheat    Turkey  
[mpalmer@localhost ~]$
```

**Figure 4-5** Results of sorting on the third field in the food file

the same task a few days later. Also, there might be other people, such as an assistant or supervisor, who need to execute your stored commands and cannot access them from their own user accounts. MS-DOS and Windows users resolve this problem by creating batch files, which are files of commands that are executed when the batch file is run. UNIX/Linux users do the same: They create **shell script** files to contain command-line entries. Like MS-DOS/Windows batch files, script files contain commands that can be run sequentially as a set. For example, if you often create a specific report using a combination of the *cut*, *paste*, and *sort* commands, you can create a script file containing these commands. Instead of having to remember the exact commands and sequence each time you want to create the report, you instead execute the script file. Creating script files in this way can save you a significant amount of time and aggravation.

After you determine the exact commands and command sequence, use the *vi* or *Emacs* editor to create the script file. (See Figure 4-6.) Next, make the script file executable by using the *chmod* command with the *x* argument, as you learned in Chapter 2. Finally, use the *./* command to run a script, such as typing *./myscript* and pressing Enter to run the script file *myscript*.

Script files can range from the simple to the complex. In Hands-On Project 4-15, you get a basic introduction to using these files. Chapters 6 and 7, “Introduction to Shell Script Programming” and “Advanced Shell Programming,” give you much more experience with script files (or scripts for short).



Figure 4-6 Sample script file

## USING THE JOIN COMMAND ON TWO FILES

Sometimes, it is useful to know how to link the information in two files. You can use the *join* command to associate lines in two files on the basis of a common field in both files. If you want the results sorted, you can either sort the files on a common field before you join the information or sort on a specific field after you join the information from the files. For example, suppose you have a file that contains the employee's last name in one field, the employee's company ID in another field, and the employee's salary in the final field, as follows:

```

Brown:82:53,000
Anders:110:32,000
Caplan:174:41,000
Crow:95:36,000

```

Also, you have another file that contains each employee's last name, first name, middle initial, department, telephone number, and other information, but that file does not contain salary information, as follows:

```

Brown:LaVerne:F:Accounting Department:444-7508: . . .
Anders:Carol:M:Sales Department:444-2130: . . .
Caplan:Jason:R:Payroll Department:444-5609: . . .
Crow:Lorretta:L:Shipping Department:444-8901: . . .

```

You want to create a new third file to use for budgeting salaries that contains only the employee's last name, first name, department, and salary. To do this, you could use the *join* command to create a file with the following contents:

```

Brown:LaVerne:Accounting Department:53,000
Anders:Carol:Sales Department:32,000
Caplan:Jason:Payroll Department:41,000
Crow:Lorretta:Shipping Department:36,000

```

In this simple example, the common field for the two original files is the employee's last name. Note that in this context, the common field provides a **key** for accessing and joining the information to create a report or to create another file with the joined information. (Also refer back to Figure 4-2(c) for an example of a key-based file structure.)



The *join* command is also associated with linking information in complex databases. The use of these databases, such as **relational databases**, is beyond the scope of this book. However, learning the *join* command to manipulate data in flat files, as used in this book, can be useful. It is used here as another file manipulation tool to complement your knowledge of the *paste*, *cut*, and *sort* commands.

---

**Syntax** *join* [-options] *file1 file2*

---

### Dissection

- Used to associate information in two different files on the basis of a common field or key in those files
  - *file1, file2* are two input files that must be sorted on the join field—the field you want to use to join the files. The join field is also called a key. You must sort the files before you can join them. When you issue the *join* command, UNIX/Linux compare the two fields. Each output line contains the common field followed by a line from *file1* and then a line from *file2*. You can modify output using the options described next. If records with duplicate keys are in the same file, UNIX/Linux join on all of them. You can create output records for unpairable lines, for example, to append data from one file to another without losing records.
  - Useful options include:
    - 1 *fieldnum* specifies the common field in *file1* on which to join
    - 2 *fieldnum* specifies the common field in *file 2* on which to join
    - o specifies a list of fields to output. The list contains blank-separated field specifiers in the form *m.n*, where *m* is the file number and *n* is the position of the field in the file. Thus, -o 1.2 means “output the second field in the first file.”
    - t specifies the field separator character. By default this is a blank, tab, or new line character. Multiple blanks and tabs count as one field separator.
    - a *filenum* produces a line for each unpairable line in the file *filenum*. (In this case, *filenum* is a 1 for *file1* or a 2 for *file2*.)
    - e *str* replaces the empty fields for the unpairable line in the string specified by *str*. The string is usually a code or message to indicate the condition, for example, -e “No Vendor Record.”
- 

Hands-On Project 4-16 gives you an opportunity to use the *join* command.

---

## A BRIEF INTRODUCTION TO THE AWK PROGRAM

Awk, a pattern-scanning and processing language, helps to produce reports that look professional. Although you can use the *cat* and *more* commands to display the output file that you create with your *join* command, the *awk* command (which starts the Awk program when you enter it on the command line) lets you do the same thing more quickly and easily.

The name Awk is formed from the initials of its inventors, (Alfred) Aho, (Peter) Weinberger, and (Brian) Kernighan. They have provided a rich and powerful programming environment in UNIX/Linux that is well worth the effort to learn, because it can perform actions on files that range from the simple to the complex—and can be difficult to duplicate using a combination of other commands.



In Fedora, Red Hat Enterprise Linux, SUSE and some other versions of UNIX and Linux, you actually use *gawk*, which includes enhancements to *awk* and was developed for the GNU Project by Paul Rubin and Jay Fenlason. When you type *awk* at the command line, you really execute *gawk*—or you can just type *gawk*.

---

**Syntax** *awk* [- Fsep ] ['*pattern* {*action*} ..'] *filenames*

---

### Dissection

- *awk* checks to see if the input records in the specified files satisfy the *pattern* and, if they do, *awk* executes the *action* associated with it. If no pattern is specified, the action affects every input record.
  - *-F*: means the field separator is a colon
- 

The *awk* command is used to look for patterns in files. After it identifies a pattern, it performs an action that you specify. One reason to learn *awk* is to have a tool at your fingertips that lets you manipulate data files very efficiently. For example, you can often do the same thing in *awk* that would take many separate commands using a combination of *paste*, *cut*, *sort*, and *join*. Another reason for learning *awk* is that you might have a project you simply can't complete using a combination of *paste*, *cut*, *sort*, and *join*, but you can complete it using *awk*.

Some of the tasks you can do with *awk* include:

- Manipulate fields and records in a data file.
- Use variables. (You learn more about variables in Chapter 6.)

- Use arithmetic, string, and logical operators. (You learn more about these types of operators in Chapters 6 and 7.)
- Execute commands from a shell script.
- Use classic programming logic, such as loops. (You learn more looping logic in Chapter 6.)
- Process and organize data into well-formatted reports.

Consider a basic example in which you want to print text to the screen. The following is a simple *awk* command-line sequence that illustrates the syntax:

```
awk 'BEGIN { print "This is an awk print line." }'
```

When you type this at the command line, the following appears on the screen:

```
This is an awk print line.
```

The *awk* command-line sequence to produce this output does the following things:

1. *awk* starts the Awk program to process the command-line actions.
2. The pattern is signaled by BEGIN.
3. The pattern and the action are enclosed in single quotation marks.
4. The action in the curly brackets { } is processed by the Awk program.
5. The Awk *print* command is executed to print the string inside the double quotation marks (input from the keyboard or stdin) so that it appears on the screen (stdout).

Using a more advanced example, you can use *awk* to process input from a data file and display a report as output. Consider the following sample *awk* command-line sequence:

```
awk -F: '{printf "%s\t %s\n", $1, $2}' datafile
```

In this example, the following happens:

1. *awk -F:* starts the Awk program and tells Awk that the field separator between records in the input file (datafile) is a colon.
2. The pattern and action are enclosed within the single quotation marks.
3. *printf* is a command used in the Awk program to print and format the output. (You learn more about *printf* in Chapter 5, “Advanced File Processing”.) In this case, the output goes to the screen (stdout).
4. *\$1* and *\$2* signify that the fields to print and format are the first (*\$1*) and second (*\$2*) fields in the specified input file, which is datafile.
5. *datafile* is the name of the input file that contains records divided into fields.

Try Hands-On Projects 4-17 and 4-18 for a further introduction to *awk*.



*awk* is presented here to give you a first, experiential taste of this powerful tool. There is a lot to learn about using *awk*, and you learn more in later chapters. For now, consider this brief introduction of *awk* as a natural follow-on to your introduction to the *join* command—like a musician sight-reading new music as a rudimentary step to learning more about it. For more information about *awk*, type *man awk* to read the online documentation.

## CHAPTER SUMMARY

- UNIX/Linux support regular files, directories, character special files, and block special files. Regular files contain user information. Directories are system files for maintaining the file system's structure. Character special files are related to serial input/output devices, such as printers. Block special files are related to devices, such as disks.
- Files can be structured in several ways. UNIX/Linux store data, such as letters, product records, or vendor reports, in flat ASCII files. File structures depend on the kind of data being stored. Three kinds of regular files are unstructured ASCII characters, records, and trees.
- Often, flat ASCII data files contain records and fields. They typically use one of two formats: variable-length records and fixed-length records. Variable-length records usually have fields that are separated by a delimiter, such as a colon. Fixed-length records have fields that are in specific locations, such as a column range, within a record.
- When performing commands, UNIX/Linux process data—they receive input from the standard input device and then send output to the standard output device. UNIX/Linux refer to the standard devices for input and output as *stdin* and *stdout*, respectively. By default, *stdin* is the keyboard and *stdout* is the monitor. Another standard device, *stderr*, refers to the error file that defaults to the monitor. Output from a command can be redirected from *stdout* to a disk file. Input to a command can be redirected from *stdin* to a disk file. The error output of a command can be redirected from *stderr* to a disk file.
- The *touch* command updates a file's time stamp and date stamp and creates empty files.
- The *rmdir* command removes an empty directory. Also, the *rm* command can be used to delete a file, and the *rm* command with the *-r* option can be used to delete a directory that contains files and subdirectories.
- The *cut* command extracts specific columns or fields from a file. Select the fields you want to cut by specifying their positions and separator character, or you can cut by character positions, depending on the data's organization.
- To combine two or more files, use the *paste* command. Where *cat* appends data to the end of the file, the *paste* command combines files side by side. You can also use *paste* to combine fields from two or more files.
- Use the *sort* command to sort a file's contents alphabetically or numerically. UNIX/Linux display the sorted file on the screen by default, but you can also specify that you want to store the sorted data in a particular file.

- To automate command processing, include commands in a script file that you can later execute as a program. Use the `vi` editor to create the script file, and use the `chmod` command to make it executable.
- Use the `join` command to extract information from two files sharing a common field. You can use this common field to join the two files. You must sort the two files on the join field—the one you want to use to join the files. The join field is also called a key. You must sort the files before you can join them.
- Awk is a pattern-scanning and processing language useful for creating a formatted report with a professional look. You can enter the Awk language instructions in a program file using the `vi` editor and call it using the `awk` command.

## COMMAND SUMMARY: REVIEW OF CHAPTER 4 COMMANDS

Command	Purpose	Options Covered in This Chapter
<b>awk</b>	Starts the <code>awk</code> program to format output	-F identifies the field separator. -f indicates code is coming from a disk file, not the keyboard.
<b>cat</b>	Views the contents of a file, creates a file, merges the contents of files	
<b>cp</b>	Copies one or more files	-i provides a warning before <code>cp</code> writes over an existing file with the same name. -s creates a symbolic link or name at the destination rather than a physical file. -u prevents <code>cp</code> from copying over an existing file, if the existing file is newer than the source file.
<b>cut</b>	Extracts specified columns or fields from a file	-c refers to character positions. -d indicates that a specified character separates the fields. -f refers to fields.
<b>find</b>	Finds files	-iname specifies the name of the files you want to locate, but the search is not case sensitive. -name specifies the name of the files you want to locate, but the search is case sensitive. -mmin <i>n</i> displays files that have been changed within the last <i>n</i> minutes. -mtime <i>n</i> displays files that have been changed within the last <i>n</i> days. -size <i>n</i> displays files of size <i>n</i> .

Command	Purpose	Options Covered in This Chapter
<b>join</b>	Combines files having a common field	<ul style="list-style-type: none"> <li>-a <i>n</i> produces a line for each unpairable line in file <i>n</i>.</li> <li>-e <i>str</i> replaces the empty fields for an unpairable file with the specified string.</li> <li>-1 and -2 with the field number are used to specify common fields when joining.</li> <li>-o outputs a specified list of fields.</li> <li>-t indicates that a specified character separates the fields.</li> </ul>
<b>mv</b>	Moves one or more files	<ul style="list-style-type: none"> <li>-i displays a warning prompt before overwriting a file with the same name.</li> <li>-u overwrites a destination file with the same name, if the source file is newer than the one in the destination.</li> </ul>
<b>paste</b>	Combines fields from two or more files	<ul style="list-style-type: none"> <li>-d enables you to specify a different separator (other than a tab) between columns.</li> <li>-s causes files to be pasted one after the other instead of in parallel.</li> </ul>
<b>rm</b>	Removes one or more files	<ul style="list-style-type: none"> <li>-i specifies that UNIX/Linux should request confirmation of file deletion before removing the files.</li> <li>-r specifies that directories should be recursively removed.</li> </ul>
<b>rmdir</b>	Removes an empty directory	
<b>sort</b>	Sorts the file's contents	<ul style="list-style-type: none"> <li>-k <i>n</i> sorts on the key field specified by <i>n</i>.</li> <li>-t indicates that a specified character separates the fields.</li> <li>-m means to merge files before sorting.</li> <li>-o redirects output to the specified file.</li> <li>-d sorts in alphanumeric or dictionary order.</li> <li>-g sorts by numeric (general) order.</li> <li>-r sorts in reverse order.</li> </ul>
<b>touch</b>	Updates an existing file's time stamp and date stamp or creates empty new files	<ul style="list-style-type: none"> <li>-a specifies that only the access date and time are to be updated.</li> <li>-m specifies that only the modification date and time are to be updated.</li> <li>-c specifies that no files are to be created.</li> </ul>



## KEY TERMS

**fixed-length record** — A record structure in a file in which each record has a specified length, as does each field in a record.

**flat ASCII file** — A file that you can create, manipulate, and use to store data, such as letters, product reports, or vendor records. Its organization as an unstructured sequence of bytes is typical of a text file and lends flexibility in data entry, because it can store any kind of data in any order. Any operating system can read this file type. However, because you can retrieve data only in the order you entered it, this file type's usefulness is limited. Also called an ordinary file or regular file.

**key** — A common field in every file record shared by each of one or more files. The common field, or key, enables you to link or join information among the files, such as for creating a report.

**regular file** — A UNIX/Linux reference to ASCII/text files and binary files. Also called an ordinary file.

**relational database** — A database that contains files that UNIX/Linux treat as tables, records that are treated as rows, and fields that are treated as columns and that can be joined to create new records. For example, using the *join* command, you can extract information from two files in a relational database that share a common field.

**shell script** — A text file that contains sequences of UNIX/Linux commands that do not need to be converted into machine language by a compiler.

**sorting key** — A field position within each line of a file that is used to sort the lines. For instance, in the command *sort -k 2 myfile*, *myfile* is sorted by the second field in that file. The *sort* command sorts the lines based on the sorting key.

**stderr** — An acronym used by programmers for standard error. When UNIX/Linux detect errors in programs and program tasks, the error messages and analyses are directed to *stderr*, which is often the screen (part of the IEEE Std 1003.1 specification).

**stdin** — An acronym used by programmers for standard input and used in programming to read input (part of the IEEE Std 1003.1 specification).

**stdout** — An acronym used by programmers for standard output and used in programming to write output (part of the IEEE Std 1003.1 specification).

**variable-length record** — A record structure in a data file in which the records can have variable lengths and are typically separated by a delimiter, such as a colon.

---

## REVIEW QUESTIONS

1. You are starting a new year and need to create 10 empty files for your accounting system. Which of the following commands or operators enable you to quickly create these files? (Choose all that apply.)
  - a. *mkfile*
  - b. *>*
  - c. *newfile*
  - d. *;*
2. Your project team uses a group of the same files and tracks whether they are still in use by looking at the last modified date. You need to show that a series of files are still in use by changing the last modified date to today. What command do you use?
  - a. *date -m*
  - b. *chdate*
  - c. *touch*
  - d. *update*
3. Which of the following are ways in which you can structure a record containing data? (Choose all that apply.)
  - a. *prosadium*
  - b. *scanned*
  - c. *fixed-length*
  - d. *variable-length*
4. You need to delete 30 files that all start with the letters “customer,” such as *customer\_accounts*, *customer\_number*, and so on. Which of the following commands enables you to quickly delete these files?
  - a. *removeall customer*
  - b. *omit customer*
  - c. *-customer*
  - d. *rm customer\**
5. You are in your home directory and need to copy the file *MemoRequest* to a folder under your home directory called *Memos*. Which of the following commands do you use?
  - a. *cp MemoRequest Memos*
  - b. *duplicate MemoRequest Memos*
  - c. *mv MemoRequest /*
  - d. *link / Memos MemoRequest*

6. You have a personnel file that contains the names, addresses, and telephone numbers of all employees. The first field in the personnel file is the employee number and the second field is the employee last name. The third field is the first name and middle initial. Because there are no employees with the same last name, you want to check the file by last name to make certain all employees are included. Which of the following commands should you use?
  - a. *paste -l2 personnel*
  - b. *cut -f2 personnel*
  - c. *capture personnel 2*
  - d. *join 2.0 personnel*
7. You are doing some “house cleaning” and want to delete several empty directories. Which of the following commands can you use? (Choose all that apply.)
  - a. *rmdir*
  - b. *dirdel*
  - c. *deldir*
  - d. *clear -d*
8. You are trying to use the command *sort -t: +5 datastore*, but you get an error message each time you try it. How can you save the error message in a file called *error*, so you can e-mail the file to your computer support person?
  - a. Enter *error sort -t: +5 datastore*.
  - b. Enter *stderr sort -t: +5 datastore*.
  - c. Enter *sort -t: +5 datastore << error*.
  - d. Enter *sort -t: +5 datastore 2> error*.
9. Which of the following commands would you use to make a backup copy of the file *AR2008*?
  - a. *bak AR2008*
  - b. *AR2008 <<*
  - c. *cp AR2008 AR2008.bak*
  - d. *comp AR2008 < bak*
10. You have a lot of subdirectories under your home directory and know that you saved the file *supplemental* in one of them, but you are not sure which one. After you use *cd* to change to your home directory, which of the following commands enables you to search all of your subdirectories for the file?
  - a. *search / supplemental*
  - b. *mv -s supplemental*
  - c. *dirsearch supplemental*
  - d. *find -name supplemental*

11. You have created a script file called `sum_report` in your home directory and have made it executable. What command do you use to run the script?
  - a. `./sum_report`
  - b. `rm sum_report`
  - c. `go sum_report`
  - d. `#sum_report`
12. Standard output is referred to as which of the following?
  - a. `out_put`
  - b. `stdout`
  - c. `termout`
  - d. `outsouce`
13. Which of the following conditions must be met for you to combine two files using the `join` command? (Choose all that apply.)
  - a. The two files must be exactly the same length.
  - b. Both files must have a field that contains a numbered identifier.
  - c. The two files must have a common field, such as last name.
  - d. Both files must use a colon as the field separator.
14. Which of the following commands can you use to sort the file `vendor_name` and display the results on the screen?
  - a. `sort > vendor_name`
  - b. `sort vendor_name <`
  - c. `sort vendor_name stdin`
  - d. `sort vendor_name`
15. When you use the `paste` command, columns of information are separated by a tab. However, your boss wants the columns separated by a colon. What option enables you to specify the colon as the separator?
  - a. `-s:`
  - b. `-d:`
  - c. `--sep :`
  - d. `--div:`

16. You are examining your addresses file, which contains the first and last names of people you know as well as their street address, city, state, zip code, and telephone number. You want to print a list of last names, which is field 1, and telephone numbers, which is field 7. Which of the following commands enables you to print this list?
- a. `cat 1:7 addresses`
  - b. `sort -t: 1-7 addresses`
  - c. `cut -f1,7 addresses`
  - d. `paste r:1:7 addresses`
17. You keep a yearly record of the birds you've seen in your town. The name of the file is `birds`. The file contains the following fields: name (field 1), markings (field 2), year(s) viewed (field 3), and location (field 4). You want to review the contents of the file, sorted by location. Which of the following commands do you use?
- a. `sort -k 4 birds`
  - b. `sort birds >4`
  - c. `paste birds -s:45`
  - d. `cut birds .4`
18. Which of the following can be accomplished with the `mv` command? (Choose all that apply.)
- a. move a file
  - b. modify the contents of a file
  - c. rename a file
  - d. change a file's owner
19. Your boss asks you to create a professional-looking report from the contents of two files. Which of the following tools enables you to produce a polished report?
- a. `cat`
  - b. `awk`
  - c. `touch`
  - d. `finetouch`
20. What command enables you to sort the contents of a file in reverse order?
- a. `cut -r`
  - b. `paste -b`
  - c. `sort -t`
  - d. `sort -r`

21. You want to combine two files, `data07` and `data08`, into a file called `data_all`. Which of the following commands do you use?
  - a. `cut data07 + data08 2>> data_all`
  - b. `paste data07/data08 to data_all`
  - c. `sort data 07 data08 <<2 data_all`
  - d. `cat data07 data08 > data_all`
22. How can you use the `touch` command to create four new files called `sum`, `datanew`, `results`, and `calcs` (using one command line)?
23. Create a command that sorts on the second field in the file `addresses` and then writes the sorted results to the new file, `sorted_addresses`.
24. Create a command that enables you to copy all of the files in the `spreadsheets` directory to the `accounts` directory (when both directories are first-level directories under your home directory and you are currently in your home directory).
25. You play guitar and keep two files on your computer. One file, called `strings`, lists the different brands of strings you keep on hand. Another file, called `music`, lists the music books and scores you own. When you enter the command `paste strings music`, what happens?

---

## HANDS-ON PROJECTS



**TIP**

Remember that you can type `clear` and press Enter at the end of any project to start with a fresh screen display.



### Project 4-1

You can handle input and output in UNIX/Linux in many ways. In this project, you create a file that feeds commands into the `vi` editor. (For this and all projects in the chapter, log in using your own account, rather than logging in as root.)

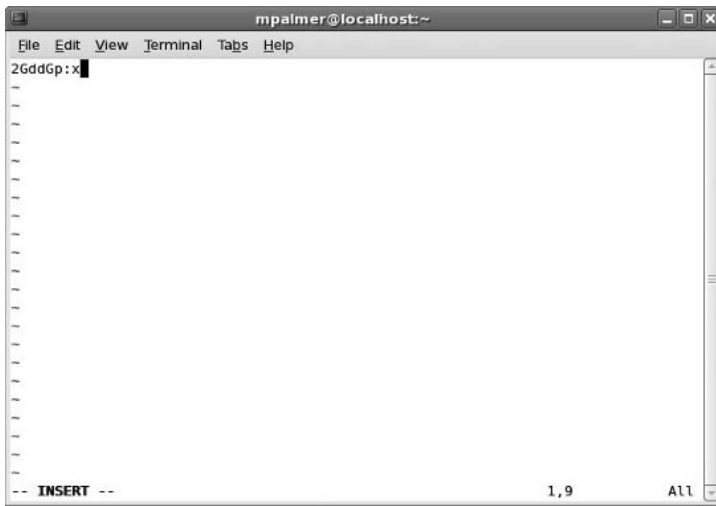
#### To create a file from which the `vi` editor reads commands:

1. Use the `vi` editor to create and then save the file `testfile` containing the following text (ensure that you don't enter a blank line after the last line):

```
This is line 1.
This is line 2.
This is line 3.
This is line 4.
```

2. Type **cat testfile** and press **Enter** to check your work.

- Next, using the vi editor, create and save another text file named **commands** with one line containing the following vi commands: **2GddGp:x**. (See Figure 4-7.)



**Figure 4-7** Using vi to create the commands file

- Type **cat commands** and press **Enter** to verify the contents of the commands file.
- Type **vi testfile < commands** and press **Enter**. (Because the input is from a file and not the keyboard, you might see a warning that the input is not from a terminal. Simply ignore the warning.) This loads testfile into the vi editor and redirects vi's input to the text in the commands file. The text in the commands file is treated as commands typed on the keyboard.
- Type **cat testfile** and press **Enter**. You see the contents of testfile after the vi commands execute. The contents are:

```
This is line 1.
This is line 3.
This is line 4.
This is line 2.
```



## Project 4-2

In this project, you use the **2>** redirection operator to write an error message to a file.

### To redirect an error message:

- Force the **ls** command to display an error message by giving it an invalid argument. Assuming you have no file or directory in your home directory named **oops**, type **ls oops** and press **Enter**. You see the following error message:

```
ls: oops: No such file or directory
```

2. Redirect the error output of the `ls` command. Type **`ls oops 2> errfile`** and press **Enter**. There is no output on the screen.
3. Type **`cat errfile`** and press **Enter**. You see `errfile`'s contents:  

```
ls: oops: No such file or directory
```
4. See Figure 4-8.

```

mpalmer@localhost:~
File Edit View Terminal Tabs Help
[mpalmer@localhost ~]$ ls oops
ls: oops: No such file or directory
[mpalmer@localhost ~]$ ls oops 2> errfile
[mpalmer@localhost ~]$ cat errfile
ls: oops: No such file or directory
[mpalmer@localhost ~]$

```

Figure 4-8 Creating the `errfile` and viewing its contents



## Project 4-3

In this project, you practice using the `>` redirector to create an empty file.

**To create an empty file:**

1. Type **`> newfile1`** and press **Enter**. This creates an empty file called `newfile1`.
2. To list the new file, type **`ls -l newfile1`** and press **Enter**.
3. You see only the information listed next, where `jean` is your user name.  

```
-rw-r--r-- 1 jean jean    0 Nov 1 16:57 newfile1
```
4. To create another new file, type **`> newfile2`** and press **Enter**.



## Project 4-4

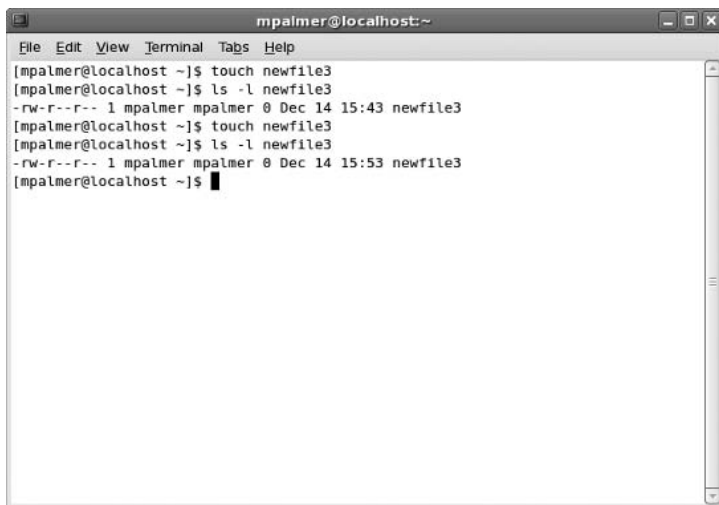
This project shows you how to use the `touch` command to create a file and to change its time stamp.

**To create a file and alter its date/time stamp with the `touch` command:**

1. Type **`touch newfile3`** and press **Enter**. This command creates the file `newfile3`.



2. Type **ls -l newfile3** and press **Enter**. You see a long listing for the newfile3 file. Note its modification date and time.
3. Wait at least one minute.
4. Type **touch newfile3** and press **Enter**. This updates the file's access and modification date stamp and time stamp with the system date and time.
5. Type **ls -l newfile3** and press **Enter**. Look at the file's modification time. It should be different now. (Figure 4-9 shows the time stamp changed after 10 minutes.)



```

mpalmer@localhost:~
File Edit View Terminal Tabs Help
[mpalmer@localhost ~]$ touch newfile3
[mpalmer@localhost ~]$ ls -l newfile3
-rw-r--r-- 1 mpalmer mpalmer 0 Dec 14 15:43 newfile3
[mpalmer@localhost ~]$ touch newfile3
[mpalmer@localhost ~]$ ls -l newfile3
-rw-r--r-- 1 mpalmer mpalmer 0 Dec 14 15:53 newfile3
[mpalmer@localhost ~]$

```

Figure 4-9 Using *touch* to create a file and change its time stamp



## Project 4-5

In this project, you delete two of the files you created previously.

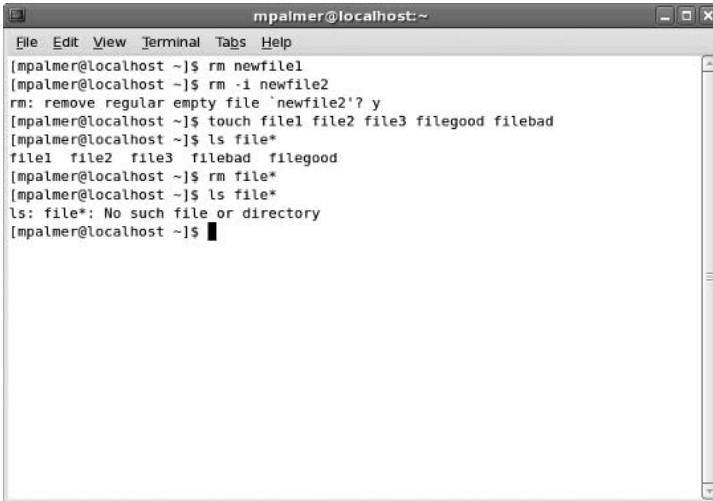
### To delete a file from the current directory:

1. Type **rm newfile1** and press **Enter**. This permanently deletes newfile1 from the current directory.
2. Type **rm -i newfile2** and press **Enter**. You see the message, *rm: remove regular empty file 'newfile2'?*
3. Type **y** for yes and press **Enter**.

### To delete a group of files using wildcards:

1. You can specify multiple file names as arguments to the *touch* command. Type **touch file1 file2 file3 filegood filebad** and press **Enter**. This command creates these files: file1, file2, file3, filegood, and filebad.
2. Type **ls file\*** and press **Enter**. You see the listing for the files you created in Step 1.

3. Type **rm file\*** and press **Enter**.
4. Type **ls file\*** and press **Enter**. The files have been erased. (See Figure 4-10.)

A terminal window titled 'mpalmer@localhost:~' with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal shows the following commands and output:

```
[mpalmer@localhost ~]$ rm newfile1
[mpalmer@localhost ~]$ rm -i newfile2
rm: remove regular empty file 'newfile2'? y
[mpalmer@localhost ~]$ touch file1 file2 file3 filegood filebad
[mpalmer@localhost ~]$ ls file*
file1 file2 file3 filebad filegood
[mpalmer@localhost ~]$ rm file*
[mpalmer@localhost ~]$ ls file*
ls: file*: No such file or directory
[mpalmer@localhost ~]$
```

Figure 4-10 Deleting files with the *rm* command



## Project 4-6

In this project, you create a directory and then use the *rmdir* command to remove it. Then, you use the *rm* command to delete a directory that contains subdirectories.

**To create a directory and then remove it with the *rmdir* command:**

1. Type **mkdir newdir** and press **Enter**. This creates a new directory named newdir.
2. Use a relative path with the *touch* command to create a new file in the newdir directory. Type **touch newdir/newfile** and press **Enter**. This creates the file newfile in the newdir directory.
3. Type **ls newdir** and press **Enter** to see a listing of the newfile file.
4. To attempt to remove the directory, type **rmdir newdir** and press **Enter**. You see an error message similar to:  

```
rmdir: newdir: Directory not empty
```
5. Use a relative path with the *rm* command to delete newfile. Type **rm newdir/newfile** and press **Enter**.
6. The directory is now empty. Type **rmdir newdir** and press **Enter**.
7. Type **ls** and press **Enter**. The newdir directory is no longer there.

**To recursively remove a directory with several subdirectories:**

1. Create a directory with several subdirectories. Type **mkdir company** and press **Enter**. Type **mkdir company/sales** and press **Enter**. Type **mkdir company/marketing** and press **Enter**. Type **mkdir company/accounting** and press **Enter**.
2. Create three empty files in the company directory. Type **touch company/file1 company/file2 company/file3** and press **Enter**.



The commands you type next in Step 3 are very similar. You can reduce your typing by using the up arrow key to recall the first command and then modify it.

3. Copy the files to the other directories by doing the following:
  - Type **cp company/file1 company/file2 company/file3 company/sales** and press **Enter**.
  - Type **cp company/file1 company/file2 company/file3 company/marketing** and press **Enter**.
  - Type **cp company/file1 company/file2 company/file3 company/accounting** and press **Enter**.
4. Use the **ls** command to verify that the files were copied into all three directories.
5. Remove the company directory and everything it contains. Type **rm -r company** and press **Enter**.
6. Type **ls** and press **Enter**. The company directory is removed.

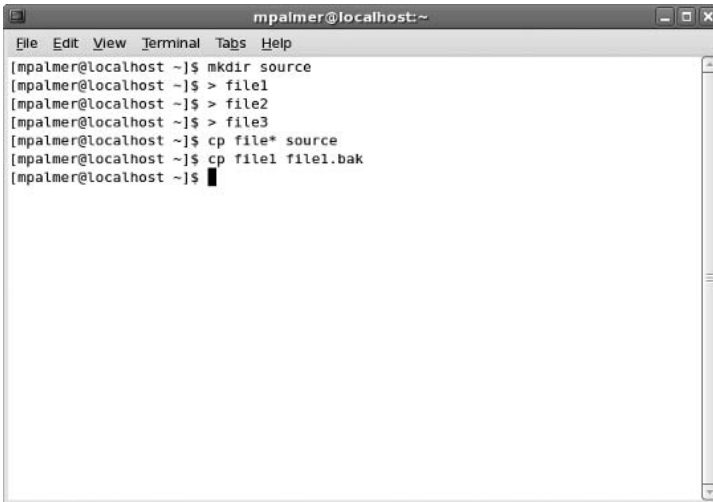
**Project 4-7**

The **cp** command is especially useful for preventing data loss; you can use it to make backup copies of your files. In this project, you create three new files, and then copy them to a different directory. Then, you duplicate one file and give it a different name.

**To create three files and copy them to a directory:**

1. If you do not already have a subdirectory called **source**, be certain you're in your home directory and then create the directory. Type **mkdir source** and then press **Enter**.
2. To create three files in your home directory, type **> file1** and press **Enter**, type **> file2** and press **Enter**, and then type **> file3** and press **Enter**.
3. Now, you can copy the three files to the source directory. Type **cp file1 file2 file3 source** and press **Enter**. (Or to save time, you can type **cp file\* source**.)
4. Next, copy one of the files and give it a different name, so you can distinguish it as a backup file. Type **cp file1 file1.bak** and press **Enter**. (See Figure 4-11.)

Now your working directory contains two files with identical contents but different names.



```

mpalmer@localhost:~
File Edit View Terminal Tabs Help
[mpalmer@localhost ~]$ mkdir source
[mpalmer@localhost ~]$ > file1
[mpalmer@localhost ~]$ > file2
[mpalmer@localhost ~]$ > file3
[mpalmer@localhost ~]$ cp file* source
[mpalmer@localhost ~]$ cp file1 file1.bak
[mpalmer@localhost ~]$

```

**Figure 4-11** Copying files to a new subdirectory and creating a backup file



## Project 4-8

In this project, you use the *mv* command to practice moving files.

### To move a file from one directory to another:

1. To create the new file **thisfile** in your home directory, type **> thisfile** and then press **Enter**.
2. Type **mv thisfile source** and press **Enter** to move the new file to the source directory.
3. Type **ls** and press **Enter**. **thisfile** is not listed. Type **ls source** and press **Enter**. You see **thisfile** listed.
4. To move more than one file, type the file names before the directory name. For example, type **mv file1 file1.bak source** and press **Enter**.
5. To create the new file **my\_file**, type **> my\_file** and press **Enter**.
6. To rename **my\_file** to **your\_file**, type **mv my\_file your\_file** and press **Enter**.
7. Type **ls** and press **Enter**. You see **your\_file** listed, but **my\_file** is not listed.



## Project 4-9

In this project, you use the *find* command to find every file named *file1* in the */home* directory and all its subdirectories.

### To find a file:

1. Type **find /home -name file1** and press **Enter**.
2. In what directories do you find the file? If there are other users on the system, can you view their home directories to see if *file1* exists there?



**TIP**

Although Linux does not require it, some UNIX versions require the *-print* option after the file name to display the names of files the *find* command locates.

4



## Project 4-10

In the next projects, you practice creating simple data files to gain initial experience in manipulating those files using UNIX/Linux commands. For this project, you begin by using the *cat* command with a redirection operator to practice creating and combining the *product1* and *product2* files and to further explore the versatility of the redirection operator. The basic files that you work with, in this instance, are sample product description files, with each record (line) containing the name of the product (in the first field) and a number to further help identify that product (in the second field). Figure 4-12 illustrates a conceptual example of the contents of the two files you create.

File name: **product1**

Lobby Furniture	1201
Ballroom Specialties	1221
Poolside Carts	1320
Formal Dining Specials	1340
Reservation Logs	1410

File name: **product2**

Plumbing Supplies	1423
Office Equipment	1361
Carpeting Services	1395
Auto Maintenance	1544
Pianos and Violins	1416

**Figure 4-12** Two sample product description files

**To use the *cat* command to combine files:**

1. Type **cat > product1** and press **Enter**.
2. Type the following text, pressing **Enter** at the end of each line:

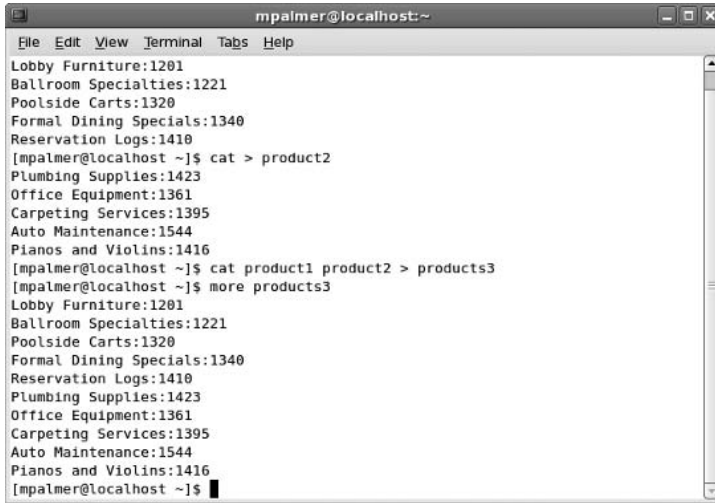
```
Lobby Furniture:1201
Ballroom Specialties:1221
Poolside Carts:1320
Formal Dining Specials:1340
Reservation Logs:1410
```

3. Press **Ctrl+d**.
4. Now, you can redirect the output of *cat* to create the product2 file in your home directory. This file also contains two colon-separated fields.
5. After the command prompt, type **cat > product2** and press **Enter**.
6. Type the following text, pressing **Enter** at the end of each line:

```
Plumbing Supplies:1423
Office Equipment:1361
Carpeting Services:1395
Auto Maintenance:1544
Pianos and Violins:1416
```

7. Press **Ctrl+d**.
8. Now, you can combine the two files in a master products file. After the \$ command prompt, type **cat product1 product2 > products3** and press **Enter**.
9. To list the contents of products, type **more products3** and press **Enter**. You see the following list (see Figure 4-13 for the entire command sequence):

```
Lobby Furniture:1201
Ballroom Specialties:1221
Poolside Carts:1320
Formal Dining Specials:1340
Reservation Logs:1410
Plumbing Supplies:1423
Office Equipment:1361
Carpeting Services:1395
Auto Maintenance:1544
Pianos and Violins:1416
```



```
mpalmer@localhost:~  
File Edit View Terminal Tabs Help  
Lobby Furniture:1201  
Ballroom Specialties:1221  
Poolside Carts:1320  
Formal Dining Specials:1340  
Reservation Logs:1410  
[mpalmer@localhost ~]$ cat > product2  
Plumbing Supplies:1423  
Office Equipment:1361  
Carpeting Services:1395  
Auto Maintenance:1544  
Pianos and Violins:1416  
[mpalmer@localhost ~]$ cat product1 product2 > products3  
[mpalmer@localhost ~]$ more products3  
Lobby Furniture:1201  
Ballroom Specialties:1221  
Poolside Carts:1320  
Formal Dining Specials:1340  
Reservation Logs:1410  
Plumbing Supplies:1423  
Office Equipment:1361  
Carpeting Services:1395  
Auto Maintenance:1544  
Pianos and Violins:1416  
[mpalmer@localhost ~]$
```

Figure 4-13 Creating and combining files with the *cat* command



## Project 4-11

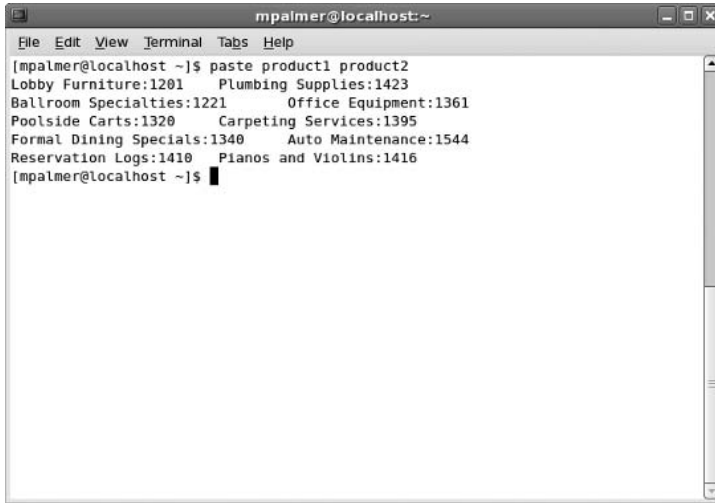
In Hands-On Project 4-10, you learned how to combine files using the *cat* command. In this project, you combine files using a different method, the *paste* command. Here, you combine the two product files you created in Project 4-10 to display the records in these files in two separate columns.

### To use the *paste* command to combine files:

1. Type **clear** and press **Enter** to clear the screen, if you are continuing directly from Project 4-10.
2. Type **paste product1 product2** and press **Enter**.

This command-line sequence means “combine the file called *product1* with the file called *product2* and show the results on the screen using a separate column for the contents of each file” (see Figure 4-14). The columns appear uneven, because the records in the files are of different lengths and are separated into columns by tab characters (which you don’t see on the screen).

3. How can you write the output of the *paste* command to a file instead of to the screen in this example?



```

mpalmer@localhost:~$ paste product1 product2
Lobby Furniture:1201      Plumbing Supplies:1423
Ballroom Specialties:1221 Office Equipment:1361
Poolside Carts:1320      Carpeting Services:1395
Formal Dining Specials:1340 Auto Maintenance:1544
Reservation Logs:1410     Pianos and Violins:1416
mpalmer@localhost ~]$

```

Figure 4-14 Using the *paste* command to combine files



## Project 4-12

The *cut* command offers versatility in manipulating and presenting the contents of basic data files. In this project, you begin by creating two files: *corp\_phones1* and *corp\_phones2*. The *corp\_phones1* file includes five records of variable size, and a colon separates each field in the record. Figure 4-2(c), shown earlier in the chapter, illustrates this type of file structure. The *corp\_phones2* file also includes five records of fixed length, illustrated in Figure 4-2(b). Figure 4-15 illustrates the contents of the two files. You can use the *cut* command with either file to extract a list of names.

### To create the *corp\_phones1* and *corp\_phones2* files:

1. Use the vi or Emacs editor to create the file **corp\_phones1**.
2. Type the following lines of text, exactly as they appear. Press **Enter** at the end of each line:
 

```

219:432:4567:Harrison:Joel:M:4540:Accountant:09-12-1985
219:432:4587:Mitchell:Barbara:C:4541:Admin Asst:12-14-1995
219:432:4589:Olson:Timothy:H:4544:Supervisor:06-30-1983
219:432:4591:Moore:Sarah:H:4500:Dept Manager:08-01-1978
219:432:4527:Polk:John:S:4520:Accountant:09-22-1998

```
3. Save the file, and create a new file named **corp\_phones2**.
4. Type the following lines of text, exactly as they appear. Consult Figure 4-15 for the precise position of each character. Press **Enter** at the end of each line. In this file, you are creating fixed-length records, which means that each field must start in a specific column and exactly line up under one another. For example, the telephone area code



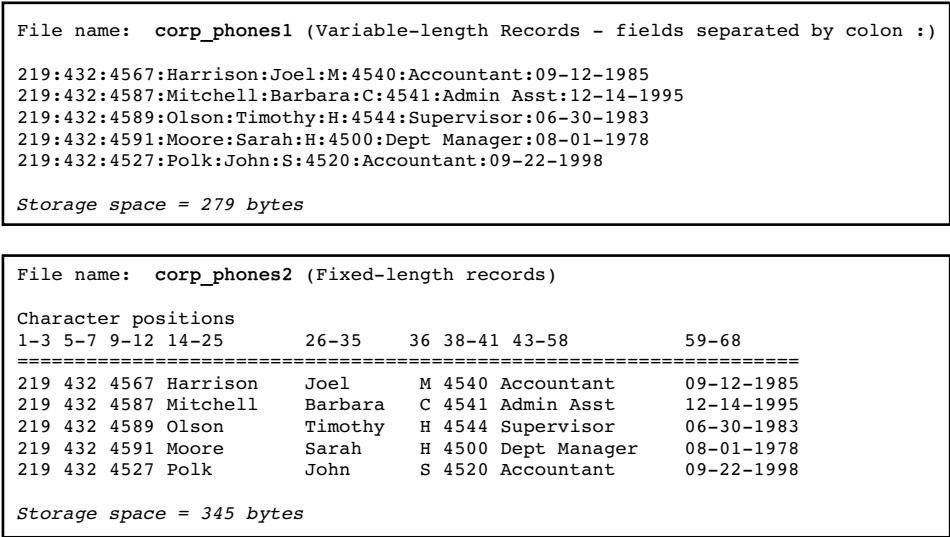


Figure 4-15 Two versions of a sample telephone file structure for a company

219 is in columns 1 through 3. The person’s last name starts in column 14 and can go through column 25.

219	432	4567	Harrison	Joel	M	4540	Accountant	09-12-1985
219	432	4587	Mitchell	Barbara	C	4541	Admin Asst	12-14-1995
219	432	4589	Olson	Timothy	H	4544	Supervisor	06-30-1983
219	432	4591	Moore	Sarah	H	4500	Dept Manager	08-01-1978
219	432	4527	Polk	John	S	4520	Accountant	09-22-1998

5. Save the file, and exit the editor.

Next, you extract the first and last names from the corp\_phones1 file. This file includes variable-length records and fields separated by colon characters. You can select the fields you want to cut by specifying their positions and separator character (which is a colon in this case).

To use the cut command to extract fields from variable-length records:

1. Type `cut -f4-6 -d: corp_phones1` and press **Enter**.

This command means “cut the fields (-f) in positions four through six (4-6) that the colon character (-d:) delimits in the corp\_phones1 file.”

You see the list of names:

```
Harrison:Joel:M  
Mitchell:Barbara:C  
Olson:Timothy:H  
Moore:Sarah:H  
Polk:John:S
```

Now, you extract the first and last names from the `corp_phones2` file. This file includes fixed-length records, instead of records containing colons to separate the fields, so you can cut by specifying character positions.

**To use the `cut` command to extract fields from fixed-length records:**

1. Type `cut -c14-25,26-35,36 corp_phones2` and press **Enter**.

This command means “cut the characters (`-c`) in positions 14 through 25, 26 through 35, and position 36 (`14-25,26-35,36`) in the `corp_phones2` file.”

You see the list of names:

Harrison	Joel	M
Mitchell	Barbara	C
Olson	Timothy	H
Moore	Sarah	H
Polk	John	S

Also, see Figure 4-16 for an example of how your screen will look after using the `cut` command on the `corp_phones1` and then the `corp_phones2` files.



Be certain not to include a space in the code sequence after the dash (`-`) options in the `cut` command. For example, the correct syntax is `cut (space) -c14-25,26-35,36 (space) corp_phones2`.

```

mpalmer@localhost:~$ cut -f4-6 -d: corp_phones1
Harrison:Joel:M
Mitchell:Barbara:C
Olson:Timothy:H
Moore:Sarah:H
Polk:John:S

[mpalmer@localhost ~]$ cut -c14-25,26-35,36 corp_phones2
Harrison      Joel      M
Mitchell     Barbara  C
Olson        Timothy  H
Moore        Sarah    H
Polk         John     S

[mpalmer@localhost ~]$
  
```

**Figure 4-16** Comparing the `cut` command results of a variable- versus a fixed-length file

Using the *cut* command with variable-length or fixed-length records produces similar results. Cutting from fixed-length records creates a more legible display but requires more storage space. For example, *corp\_phones2* requires about 345 bytes, and *corp\_phones1* requires about 279.



## Project 4-13

Sorting the *corp\_phones1* and *corp\_phones2* files you created in Hands-On Project 4-12 is relatively easy because you can refer to field numbers. In the first two steps of this project, you sort the *corp\_phones1* file by last name and first name, respectively. In the third and fourth steps, you do the same thing with *corp\_phones2*. Notice that the output of these four steps goes to stdout (the screen). The final step uses the *-o* option, instead of output redirection, to write the sorted output to a new disk file, *sorted\_phones*.



**TIP**

In earlier versions of UNIX/Linux, you had to specify character positions of fields to sort a fixed-length file, such as the *corp\_phones2* file in our project examples. This was done by using the *+F.C* option, where *F* is the number of the field and *.C* is the character position. The *+F.C* option is still available in some systems, but it is easier to use the *-k* option.

### To sort the *corp\_phones1* file:

1. After the *\$* prompt, type **`sort -t: -k 4 corp_phones1`** and press **Enter**.

In this example, the *-t* option indicates the separator character between fields, which is a colon (:). The *-k* option specifies sorting on the fourth field, or the last name field in this instance. You see the following on your screen:

```
219:432:4567:Harrison:Joel:M:4540:Accountant:09-12-1985
219:432:4587:Mitchell:Barbara:C:4541:Admin Asst:12-14-1995
219:432:4591:Moore:Sarah:H:4500:Dept Manager:08-01-1978
219:432:4589:Olson:Timothy:H:4544:Supervisor:06-30-1983
219:432:4567:Polk:John:S:4520:Accountant:09-22-1998
```

2. Type **`sort -t: -k 5 corp_phones1`** and press **Enter**.

This sorts the variable-length records (*-t:* indicates that the fields are delimited by a colon) starting at the first name field (*-k 5*). You see the following on your screen:

```
219:432:4587:Mitchell:Barbara:C:4541:Admin Asst:12-14-1995
219:432:4567:Harrison:Joel:M:4540:Accountant:09-12-1985
219:432:4567:Polk:John:S:4520:Accountant:09-22-1998
219:432:4591:Moore:Sarah:H:4500:Dept Manager:08-01-1978
219:432:4589:Olson:Timothy:H:4544:Supervisor:06-30-1983
```

3. Type **sort -k 4 corp\_phones2** and press **Enter**.

This sorts the fixed-length file by last name, starting at the fourth field. In this example, no separator is specified, because fixed-length files don't use a separator. You see the following on your screen:

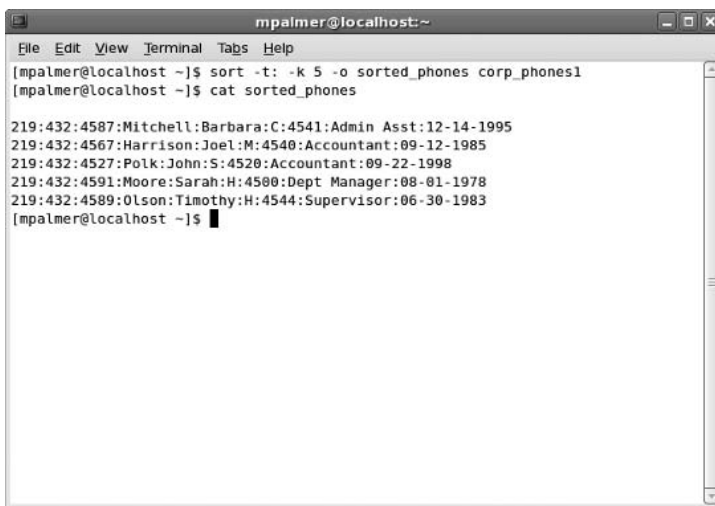
```
219 432 4567 Harrison   Joel      M 4540 Accountant   09-12-1985
219 432 4587 Mitchell  Barbara  C 4541 Admin Asst   12-14-1995
219 432 4591 Moore     Sarah    H 4500 Dept Manager 08-01-1978
219 432 4589 Olson     Timothy  H 4544 Supervisor   06-30-1983
219 432 4527 Polk      John     S 4520 Accountant   09-22-1998
```

4. Type **sort -k 5 corp\_phones2** and press **Enter**.

This sorts the file by first name, starting at the fifth field. You see the following on your screen:

```
219 432 4587 Mitchell  Barbara  C 4541 Admin Asst   12-14-1995
219 432 4567 Harrison  Joel     M 4540 Accountant   09-12-1985
219 432 4527 Polk      John     S 4520 Accountant   09-22-1998
219 432 4591 Moore     Sarah    H 4500 Dept Manager 08-01-1978
219 432 4589 Olson     Timothy  H 4544 Supervisor   06-30-1983
```

5. Type **clear** and press **Enter** to clear the screen for easier viewing.
6. To sort by first name and create the output file `sorted_phones`, type **sort -t: -k 5 -o sorted\_phones corp\_phones1** and press **Enter**. This sorts the `corp_phones1` file by first name and creates an output file, `sorted_phones`. Type **cat sorted\_phones** and press **Enter** to verify that you successfully created the `sorted_phones` file. (See Figure 4-17.)



```
mpalmer@localhost:~$ sort -t: -k 5 -o sorted_phones corp_phones1
mpalmer@localhost ~$ cat sorted_phones
219:432:4587:Mitchell:Barbara:C:4541:Admin Asst:12-14-1995
219:432:4567:Harrison:Joel:M:4540:Accountant:09-12-1985
219:432:4527:Polk:John:S:4520:Accountant:09-22-1998
219:432:4591:Moore:Sarah:H:4500:Dept Manager:08-01-1978
219:432:4589:Olson:Timothy:H:4544:Supervisor:06-30-1983
mpalmer@localhost ~$
```

**Figure 4-17** Sorting the `corp_phones1` file by first name and storing the result in `sorted_phones`



## Project 4-14

In this project, you use the many file-processing tools you've learned, to help reinforce your knowledge to this point. First, use the *cat* command to create the vendors file. The records in the vendors file consist of two colon-separated fields: the vendor number and vendor name.

### To create the vendors file:

1. Type **cat > vendors** and press **Enter**.
2. Type the following text, pressing **Enter** at the end of each line:

```
1201:Cromwell Interiors
1221:Design Extras Inc.
1320:Piedmont Plastics Inc.
1340:Morgan Catering Service Ltd.
1350:Pullman Elevators
1360:Johnson Office Products
```

3. Press **Ctrl+d**.

In the next steps, use the *cat* command to create the products file. The records in the products file consist of three colon-separated fields: the product number, the product description, and the vendor number.

### To create the products file:

1. Type **cat > products** and press **Enter**.
2. Type the following text, pressing **Enter** at the end of each line, including the last line (use all zeros in the first and last fields, including in S0107, for example):

```
S0107:Lobby Furniture:1201
S0109:Ballroom Specialties:1221
S0110:Poolside Carts:1320
S0130:Formal Dining Specials:1340
S0201:Reservation Logs:1410
```

3. Type **Ctrl+d** to end the *cat* command.
4. Figure 4-18 shows conceptual examples of the vendors and products files you have created.

Now use the *cut*, *paste*, and *sort* commands to create a single-example vendor report. You start by using the *cut* command to extract product descriptions and vendor numbers from the products file and storing them in separate files, p1 and p2. Then extract vendor numbers and names from the vendors file, and store them in v1 and v2. Use the *paste* command to combine the two vendor files (v1 and v2) in a third file, v3. Then combine the two product files (p1 and p2) in a file called p3. Sort and merge the v3 and p3 files, and send their output to the vrep file, the vendor report.

### To use the *cut*, *paste*, and *sort* commands to create a report:

1. Type **cut -f2 -d: products > p1** and press **Enter**.

```
File name: vendors

Vendor  Vendor Name
Number
=====
1201:Cromwell Interiors
1221:Design Extras Inc.
1320:Piedmont Plastics Inc.
1340:Morgan Catering Service Ltd.
1350:Pullman Elevators
1360:Johnson Office Products
```

```
File name: products

Prod      Product      Vendor
Number  Description  Number
=====
S0107:Lobby Furniture:1201
S0109:Ballroom Specialties:1221
S0110:Poolside Carts:1320
S0130:Formal Dining Specials:1340
S0201:Reservation Logs:1410
```

**Figure 4-18** Vendors and products files

This means “extract the data from the second field delimited by a colon in the products file, and store it in the p1 file.” It stores these product descriptions in the p1 file (remember you can use the *cat* command to verify the contents):

```
Lobby Furniture
Ballroom Specialties
Poolside Carts
Formal Dining Specials
Reservation Logs
```

2. Type **cut -f3 -d: products > p2** and press **Enter**.

This means “extract the data from the third field delimited by a colon in the products file, and store it in the p2 file.” It stores these vendor numbers in the p2 file:

```
1201
1221
1320
1340
1410
```

3. Type **cut -f1 -d: vendors > v1** and press **Enter**.

This means “extract the data from the first field delimited by a colon in the vendors file, and store it in the v1 file.” It stores these vendor numbers in the v1 file:

```
1201
1221
1320
1340
1350
1360
```

4. Type **cut -f2 -d: vendors > v2** and press **Enter**.

This means “extract the data from the second field delimited by a colon in the vendors file, and store it in the v2 file.” It stores these product descriptions in the v2 file:

```
Cromwell Interiors
Design Extras Inc.
Piedmont Plastics Inc.
Morgan Catering Service Ltd.
Pullman Elevators
Johnson Office Products
```

5. Type **paste v1 v2 > v3** and press **Enter**.

This means “combine the data in v1 and v2, and direct it to the file v3. It stores these vendor numbers and product descriptions in the v3 file:

```
1201 Cromwell Interiors
1221 Design Extras Inc.
1320 Piedmont Plastics Inc.
1340 Morgan Catering Service Ltd.
1350 Pullman Elevators
1360 Johnson Office Products
```

6. Type **paste p2 p1 > p3** and press **Enter**.

This means “combine the data in p2 and p1, and direct it to a file called p3.” It stores these vendor numbers and product descriptions in the p3 file:

```
1201 Lobby Furniture
1221 Ballroom Specialties
1320 Poolside Carts
1340 Formal Dining Specials
1410 Reservation Logs
```

7. Type **sort -o vrep -m v3 p3** and press **Enter**.

This means “merge the data in v3 and p3, and direct the output to a file called vrep.” It stores these vendor numbers and product descriptions in the vrep file:

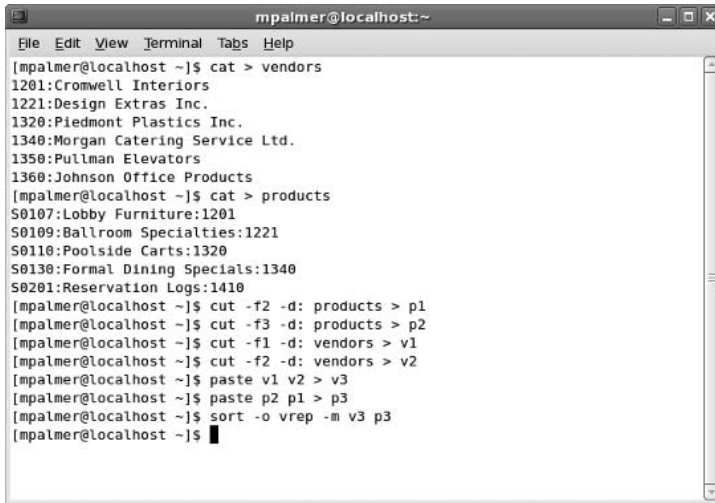
```
1201 Cromwell Interiors
1201 Lobby Furniture
1221 Ballroom Specialties
1221 Design Extras Inc.
1320 Piedmont Plastics Inc.
1320 Poolside Carts
1340 Formal Dining Specials
```

```

1340 Morgan Catering Service Ltd.
1350 Pullman Elevators
1360 Johnson Office Products
1410 Reservation Logs

```

At this point, your screen should look similar to Figure 4-19. In one project, you have accomplished quite a lot. You've used the *cat* command to create files and used the *cut*, *paste*, and *sort* commands to extract information from the files, combine the information, and then sort and merge the information into a new file.



```

mpalmer@localhost:~
File Edit View Terminal Tabs Help
[mpalmer@localhost ~]$ cat > vendors
1201:Cromwell Interiors
1221:Design Extras Inc.
1320:Piedmont Plastics Inc.
1340:Morgan Catering Service Ltd.
1350:Pullman Elevators
1360:Johnson Office Products
[mpalmer@localhost ~]$ cat > products
S0107:Lobby Furniture:1201
S0109:Ballroom Specialties:1221
S0110:Poolside Carts:1320
S0130:Formal Dining Specials:1340
S0201:Reservation Logs:1410
[mpalmer@localhost ~]$ cut -f2 -d: products > p1
[mpalmer@localhost ~]$ cut -f3 -d: products > p2
[mpalmer@localhost ~]$ cut -f1 -d: vendors > v1
[mpalmer@localhost ~]$ cut -f2 -d: vendors > v2
[mpalmer@localhost ~]$ paste v1 v2 > v3
[mpalmer@localhost ~]$ paste p2 p1 > p3
[mpalmer@localhost ~]$ sort -o vrep -m v3 p3
[mpalmer@localhost ~]$

```

Figure 4-19 Using *cat*, *cut*, *paste*, and *sort* together to create and process files



## Project 4-15

You might encounter many situations in which you must process files and create reports in the same way. You might do this on a weekly basis, for example, as the contents of files change and you want to create new reports or informational files to have on hand. Creating a script gives you a way to remember and reuse a sequence of commands that you can run over and over again. In this project, you create a simple script to process the products and vendors files and write your results to the *vrep* file.

### To use the *vi* editor to create a script:

1. Use the *vi* editor to create your script file. Type **vi ven\_report** and press **Enter**.
2. The *vi* editor starts and creates a new file, *ven\_report*.

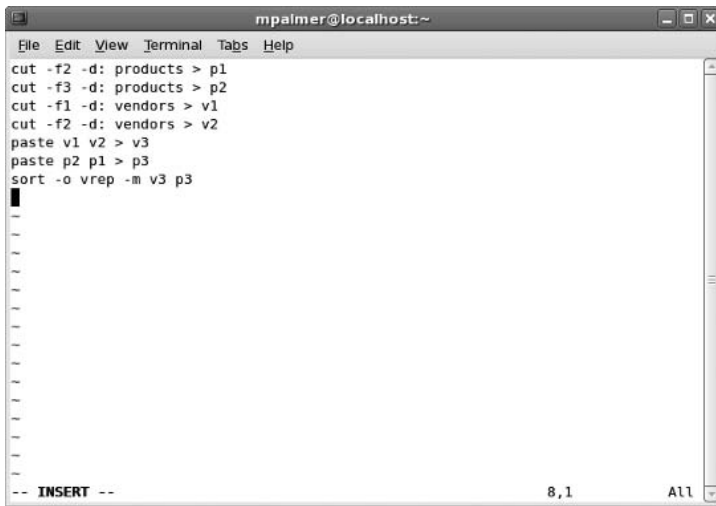


3. Enter insert mode (press **i**), and then type the following, pressing **Enter** at the end of every line:

```
cut -f2 -d: products > p1
cut -f3 -d: products > p2
cut -f1 -d: vendors > v1
cut -f2 -d: vendors > v2
paste v1 v2 > v3
paste p2 p1 > p3
sort -o vrep -m v3 p3
```

These are the same commands you used in Hands-On Project 4-14 to create the vrep vendor report. Figure 4-20 illustrates how the vi editor screen should look after you have entered the commands.

4. Press **Esc**.
5. Type **:wq** or **:x** and press **Enter** to exit the vi editor.



**Figure 4-20** Creating the ven\_report script file using the vi editor

Now, you can make the script executable with the *chmod* command. The *chmod* command sets file permissions. In the example that follows, the *chmod* command and its *ugo+x* option make the ven\_report file executable by users (owners), group, and others.

#### To make the script executable:

1. Type **chmod ugo+x ven\_report** and press **Enter**.  
(See Chapter 2 for more information on the *chmod* command.)
2. Type **rm vrep** and press **Enter** to delete the vrep file you created in Hands-On Project 4-14. Next, to ensure the script works, type **./ven\_report** and press **Enter**.  
(The **./** command enables you to run a script.)

3. Type **cat vrep** and press **Enter** to verify the vrep file contents look identical to those shown in Step 7 of Hands-On Project 4-14.

**NOTE**

In addition to making a shell script executable, it is a good idea to specify the shell for which the script is designed to run. For example, if you have designed a script for the Bash shell, you can place `#!/bin/bash` as the first line in the script. You learn how to do this in Chapter 7.



## Project 4-16

The products and vendors files that you have created offer an opportunity to begin exploring what you can do with the *join* command. This command is potentially more complex than many you have learned so far. In this project, you get a start in using the *join* command by creating a sample vendor report from the products and vendors files.

### To use the *join* command to create a report:

1. Type **join -a1 -e “No Products” -1 1 -2 3 -o '1.2 2.2' -t: vendors products > vreport ; cat vreport** and press **Enter**. (Remember that if you make a typing mistake, you can use the up arrow to recall a command, press the left arrow key to correct the mistake, and then run the command again.)

In this command, the *-1* and *-2* options indicate the first or second specified file, such as vendors or products. The numbers following *-1* and *-2* specify field numbers used for the join or match. Here, you use the first field of the vendors file to join the third field of the products file.

The *-a* option tells the command to print a line for each unpairable line in the file number. In this case, a line prints for each vendor record that does not match a product record.

The *-e* option lets you display a message for the unmatched (*-a1*) record, such as “No Products.”

The *-o* option sets the fields that will be output when a match is made.

The *1.2* indicates that field two of the vendors file is to be output along with 2.2, field two of the products file.

The *-t* option specifies the field separator, the colon. This *join* command redirects its output to a new file, vreport. The *cat* command displays the output on the screen.

See Figure 4-21 to view the output of the report.



```

mpalmer@localhost:~$ join -a1 -e "No Products" -1 1 -2 3 -o 1.2 2.2 -t: vendor
s products > vreport ; cat vreport
Cromwell Interiors:Lobby Furniture
Design Extras Inc.:Ballroom Specialties
Piedmont Plastics Inc.:Poolside Carts
Morgan Catering Service Ltd.:Formal Dining Specials
Pullman Elevators:No Products
Johnson Office Products:No Products
mpalmer@localhost ~$

```

Figure 4-21 vreport output from the *join* command



## Project 4-17

This project gives you a brief introduction to using the *awk* command for creating a more polished vendor report than in Hands-On Project 4-16 and gives you a glimpse of the next step in creating reports.

### To generate and format the vendor report:

1. Type **`awk -F: '{printf "%-28s\t %s\n", $1, $2}' vreport`** and then press **Enter**.

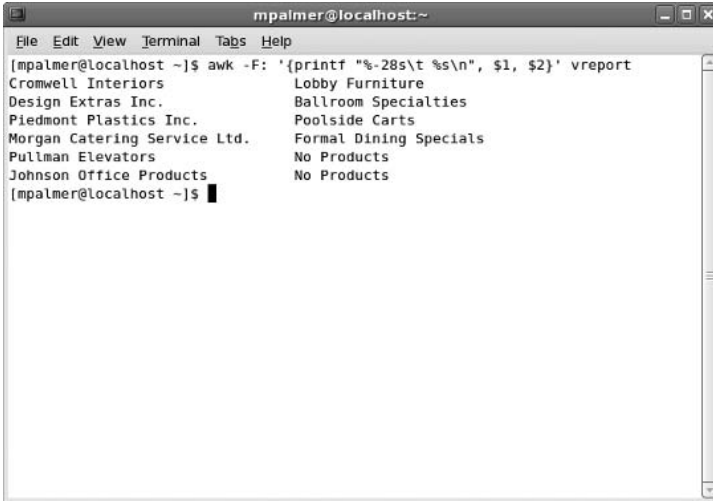
You see the vendor report, including vendor names and product descriptions, as illustrated in Figure 4-22.

The parts of the *awk* command you typed in Step 1 are:

- *awk -F*: calls the Awk program and identifies the field separator as a colon.
- `'{printf "%-28s\t %s\n", $1, $2}'` represents the action to take on each line that is read in. Single quotation marks enclose the action.
- *printf* is a print formatting function from the C programming language. It lets you specify an edit pattern for the output. The code inside the double quotation marks defines this pattern. The code immediately following the % tells how to align the field to be printed. The - sign specifies left alignment. The number that follows, 28, indicates how many characters you want to display. The trailing *s* means that the field consists of nonnumeric characters, also called a string. The *\t* inserts a tab character into the edit pattern. The %*s* specifies that another string field should be printed. You do not need to specify the string length in this case, because it is the last field printed (the product name). The *\n* specifies to skip a line after printing each output record. The \$1 and \$2, separated with a comma, indicate that the first and second fields in the input file should be placed in the edit pattern where the two *s* characters appear. The first field is the vendor name, and the second is the

product description. (You learn much more about *printf* in Chapter 10, “Developing UNIX/Linux Applications in C and C++”; it is presented here to provide you a brief introduction on which to build as you progress through the book.)

□ *vreport* is the name of the input file.



```

[mpalmer@localhost ~]$ awk -F: '{printf "%-28s\t %s\n", $1, $2}' vreport
Cromwell Interiors           Lobby Furniture
Design Extras Inc.           Ballroom Specialties
Piedmont Plastics Inc.       Poolside Carts
Morgan Catering Service Ltd. Formal Dining Specials
Pullman Elevators            No Products
Johnson Office Products     No Products
[mpalmer@localhost ~]$

```

Figure 4-22 Vendor report created via the *awk* command



## Project 4-18

To refine and automate the vendor report, you can create a shell script that uses the *awk* command. This new script, however, includes only the *awk* command, not a series of separate commands. You then call the Awk program using *awk* with the *-f* option. This option tells Awk that the code is coming from a disk file, not from the keyboard. You present the action statements inside the Awk program file, in a different way, which resembles programming code. The program file includes additional lines needed to print a heading and the current date for the report.

The next steps show what happens when you enter the Awk program in a file like this. You use the *FS* variable to tell the program what the field separator is—in this example, a colon. *FS* is one of many variables that *awk* uses to advise the program about the file being processed. Other codes you see here set up an initial activity that executes once when the program loads. *BEGIN* followed by the opening curly brace (*{*) indicates this opening activity. The closing curly brace (*}*) marks the end of actions performed when the program first loads. These actions print the headings, date, and dash lines that separate the heading from the body of the report.

### To create the *awk* script:

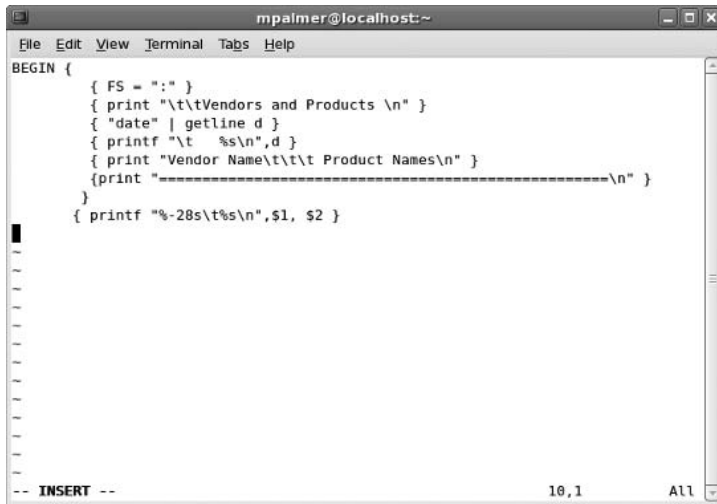
1. Type **vi awrp** and press **Enter** to start the vi editor and create the file awrp. Press **i** to start insert mode.

2. Type the following code. (Note: In the seventh line of code, enter 52 equal signs, keeping them on the same line as shown in Figure 4-23.)

```
BEGIN {
    { FS = ":" }
    { print "\t\tVendors and Products\n" }
    { "date" | getline d }
    { printf "\t  %s\n",d }
    { print "Vendor Name\t\t\t Product Names\n" }
    { print "===== \n" }
  }
  { printf "%-28s\t%s\n",$1, $2 }
```

In the code you have typed, the *getline* option is used. *Getline* is designed to read input. In this case, it reads the date and places it into the *d* variable, which then is printed via the *printf* command.

Your vi edit session should look like the one in Figure 4-23.

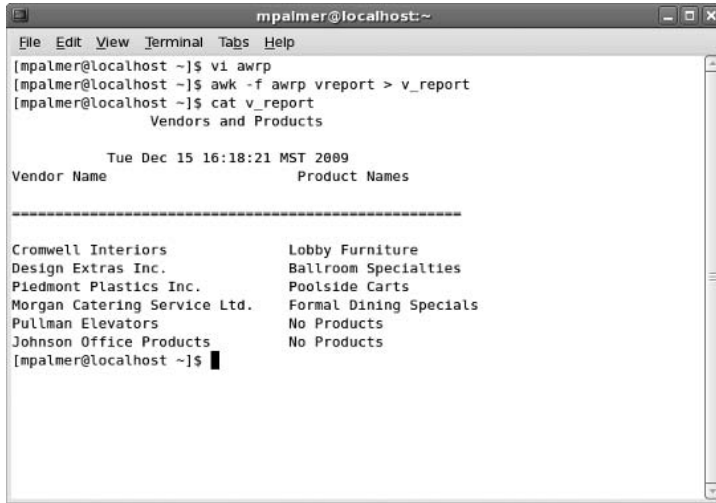


**Figure 4-23** Creating the awrp file using the vi editor

3. Press **Esc**.
4. Type **:wq** or **:x** and press **Enter** to exit the vi editor.
5. Type **awk -f awrp vreport > v\_report** and press **Enter**.

This means “using the Awk program, combine the fields from the awrp file with the fields from the vreport file, and send them to a new file called v\_report.”

6. Type **cat v\_report** and press **Enter**. Your screen should look similar to Figure 4-24.
7. To print the report on the default printer, type **lpr v\_report** and press **Enter**.



```

mpalmer@localhost ~]$ vi awrp
mpalmer@localhost ~]$ awk -f awrp vreport > v_report
mpalmer@localhost ~]$ cat v_report
      Tue Dec 15 16:18:21 MST 2009
Vendor Name      Product Names
-----
Cromwell Interiors      Lobby Furniture
Design Extras Inc.      Ballroom Specialties
Piedmont Plastics Inc.  Poolside Carts
Morgan Catering Service Ltd. Formal Dining Specials
Pullman Elevators       No Products
Johnson Office Products No Products
mpalmer@localhost ~]$

```

Figure 4-24 Viewing the contents of v\_report

## DISCOVERY EXERCISES

1. How can you create a file called history by using a redirection operator?
2. Wait one minute or more and then change the time stamp on the history file you just created.
3. Back up the history file to the file history.bak.
4. Sort the corp\_phones1 file by the last four digits of the phone number.
5. Create and use a command that displays only the last names and telephone numbers (omitting the area code) of people in the corp\_phones2 file. Place a space between the telephone number and the last name.
6. Assume you have a subdirectory named datafiles directly under your current working directory, and you have two files named data1 and data2 in your current directory. What command can you use to copy the data1 and data2 files from your current working directory to the datafiles directory?
7. Assume you have four files: accounts1, accounts2, accounts3, and accounts4. Write the *paste* command that combines these files and separates the fields on each line with a “/” character, displaying the results to the screen.
8. How would you perform the action in Exercise 7, but write the results to the file total\_accounts?
9. Assume you have 10 subdirectories and you want to locate all files that end with the extension “.c”. What command can you use to search all 10 of your subdirectories for this file?

10. After you create a script file, what are the next steps to run it?
11. Change the *awk* script that you created earlier so that the column headings are “Vendor” and “Product” and the name of the report is “Vendor Data.”
12. Create the subdirectory *mytest*. Copy a file into your new subdirectory. Delete the *mytest* subdirectory and its contents using one command.
13. Use the *cut* command to create a file called *descriptions* that contains only the product descriptions from the *products* file you created earlier in this chapter.
14. You are worried about copying over an existing or newer file in another directory when you use the *move* command. What are your options in this situation?
15. What command enables you to find all empty files in your source directory?
16. How can you find all files in your home directory that were modified in the last seven days?
17. How can you put the contents of each line of the *product1* file side by side with the contents of the *product2* file, but with only a dash between them instead of a tab?
18. Make a copy of the *corp\_phones2* file and call it *testcorp*. Next, create a single-line command that enables you to cut characters in the fifth column of the *testcorp* file and paste them back to the first column in the same file. (*Hint*: Two good solutions exist, one in which you use a semicolon and one with more finesse in which you use a pipe character.)
19. How can you use a command you have learned in this chapter to list the names of all users on your system? (*Hint*: Find out the name of the file in which user information is stored.)
20. Type *who* and press **Enter** to view a list of logged-in users, along with other information. Now use the *who* command (which you learned about in Chapter 1) with a command you learned in this chapter to view who is logged in, but to suppress all other information that normally accompanies the *who* command.

*This page intentionally left blank*