CHAPTER 10

Working with the File and Operating System

It's quite rare to write an application that is entirely self-sufficient—that is, a program that does not rely on at least some level of interaction with external resources, such as the underlying file and operating system, and even other programming languages. The reason for this is simple: As languages, file systems, and operating systems have matured, the opportunities for creating much more efficient, scalable, and timely applications have increased greatly as a result of the developer's ability to integrate the tried-and-true features of each component into a singular product. Of course, the trick is to choose a language that offers a convenient and efficient means for doing so. Fortunately, PHP satisfies both conditions quite nicely, offering the programmer a wonderful array of tools not only for handling file system input and output, but also for executing programs at the shell level. This chapter serves as an introduction to all such functionality, describing how to work with the following:

- **Files and directories:** You'll learn how to perform file system forensics, revealing details such as file and directory size and location, modification and access times, file pointers (both the hard and symbolic types), and more.
- **File ownership and permissions:** All mainstream operating systems offer a means for securing system data through a permission system based on user and group ownership and rights. You'll learn how to both identify and manipulate these controls.
- **File I/O:** You'll learn how to interact with data files, which will let you perform a variety of practical tasks, including creating, deleting, reading, and writing files.
- **Directory contents:** You'll learn how to easily retrieve directory contents.
- **Shell commands:** You can take advantage of operating system and other language-level functionality from within a PHP application through a number of built-in functions and mechanisms. You'll learn all about them. This chapter also demonstrates PHP's input sanitization capabilities, showing you how to prevent users from passing data that could potentially cause harm to your data and operating system.

■ **Note** PHP is particularly adept at working with the underlying file system, so much so that it is gaining popularity as a command-line interpreter, a capability introduced in version 4.2.0. Although this topic is out of the scope of this book, you can find additional information in the PHP manual.

Learning About Files and Directories

Organizing related data into entities commonly referred to as files and directories has long been a core concept in the computing environment. For this reason, programmers need to have a means for obtaining important details about files and directories, such as the location, size, last modification time, last access time, and other defining information. This section introduces many of PHP's built-in functions for obtaining these important details.

Parsing Directory Paths

It's often useful to parse directory paths for various attributes, such as the tailing extension name, directory component, and base name. Several functions are available for performing such tasks, all of which are introduced in this section.

basename()

```
string basename (string path [, string suffix])
```

The `basename()` function returns the filename component of `path`. If the optional `suffix` parameter is supplied, that suffix will be omitted if the returned file name contains that extension. An example follows:

```
<?php
   $path = "/home/www/data/users.txt";
   $filename = basename($path); // $filename contains "users.txt"
   $filename2 = basename($path, ".txt"); // $filename2 contains "users"
?>
```

dirname()

```
string dirname (string path)
```

The `dirname()` function is essentially the counterpart to `basename()`, providing the directory component of `path`. Reconsidering the previous example:

```
<?php
   $path = "/home/www/data/users.txt";
   $dirname = dirname($path); // $dirname contains "/home/www/data"
?>
```

pathinfo()

```
array pathinfo (string path)
```

The `pathinfo()` function creates an associative array containing three components of the path specified by `path`: directory name, base name, and extension, referred to by the array keys `dirname`, `basename`, and `extension`, respectively. Consider the following path:

```
/home/www/htdocs/book/chapter10/index.html
```

As is relevant to `pathinfo()`, this path contains three components:

- `dirname`: `/home/www/htdocs/book/chapter10`
- `basename`: `index.html`
- `extension`: `html`

Therefore, you can use `pathinfo()` like this to retrieve this information:

```
<?php
   $pathinfo = pathinfo("/home/www/htdocs/book/chapter10/index.html");
   echo "Dir name: $pathinfo[dirname]<br />\n";
   echo "Base name: $pathinfo[basename] <br />\n";
   echo "Extension: $pathinfo[extension] <br />\n";
?>
```

This returns:

```
Dir name: /home/www/htdocs/book/chapter10
Base name: index.html
Extension: html
```

realpath()

```
string realpath (string path)
```

The useful `realpath()` function converts all symbolic links, and relative path references located in `path`, to their absolute counterparts. For example, suppose your directory structure assumed the following path:

```
/home/www/htdocs/book/images/
```

You can use `realpath()` to resolve any local path references:

```
<?php
     $imgPath = "../../images/cover.gif";
     $absolutePath = realpath($imgPath);
     // Returns /www/htdocs/book/images/cover.gif
?>
```

File Types and Links

Numerous functions are available for learning various details about files and links (or file pointers) found on a file system. Those functions are introduced in this section.

filetype()

```
string filetype (string filename)
```

The `filetype()` function determines and returns the file type of `filename`. Eight values are possible:

- `block`: A block device such as a floppy disk drive or CD-ROM.
- `char`: A character device, which is responsible for a nonbuffered exchange of data between the operating system and a device such as a terminal or printer.
- `dir`: A directory.
- `fifo`: A named pipe, which is commonly used to facilitate the passage of information from one process to another.
- `file`: A hard link, which serves as a pointer to a file inode. This type is produced for anything you would consider to be a file, such as a text document or executable.
- `link`: A symbolic link, which is a pointer to the pointer of a file.
- `socket`: A socket resource. At the time of writing, this value is undocumented.
- `unknown`: The type is unknown.

Let's consider three examples. In the first example, you determine the type of a CD-ROM drive:

```
echo filetype("/mnt/cdrom"); // char
```

Next, you determine the type of a Linux partition:

```
echo filetype("/dev/sda6"); // block
```

Finally, you determine the type of a regular old HTML file:

```
echo filetype("/home/www/htdocs/index.html"); // file
```

link()

```
int link (string target, string link)
```

The `link()` function creates a hard link, `link`, to `target`, returning `TRUE` on success and `FALSE` otherwise. Note that because PHP scripts typically execute under the guise of the server daemon process owner, this function will fail unless that user has write permissions within the directory in which `link` is to reside.

linkinfo()

`int linkinfo (string *path*)`

The `lstat()` function is used to return useful information about a symbolic link, including items such as the size, time of last modification, and the owner's user ID. The `linkinfo()` function returns one particular item offered by the `lstat()` function, used to determine whether the symbolic link specified by `path` really exists. This function isn't available for the Windows platform.

lstat()

`array lstat (string *symlink*)`

The `lstat()` function returns numerous items of useful information regarding the symbolic link referenced by `symlink`. See the following section on `fstat()` for a complete accounting of the returned array.

fstat()

`array fstat (resource *filepointer*)`

The `fstat()` function retrieves an array of useful information pertinent to a file referenced by a file pointer, `filepointer`. This array can be accessed either numerically or via associative indices, each of which is listed in its numerically indexed position:

- **dev (0):** The device number upon which the file resides.
- **ino (1):** The file's inode number. The inode number is the unique numerical identifier associated with each file name and is used to reference the associated entry in the inode table that contains information about the file's size, type, location, and other key characteristics.
- **mode (2):** The file's inode protection mode. This value determines the access and modification privileges assigned to the file.
- **nlink (3):** The number of hard links associated with the file.
- **uid (4):** The file owner's user ID (UID).
- **gid (5):** The file group's group ID (GID).
- **rdev (6):** The device type, if the inode device is available. Note that this element is not available for the Windows platform.
- **size (7):** The file size, in bytes.
- **atime (8):** The time of the file's last access, in Unix timestamp format.
- **mtime (9):** The time of the file's last modification, in Unix timestamp format.
- **ctime (10):** The time of the file's last change, in Unix timestamp format.

- **blksize (11):** The file system's block size. Note that this element is not available on the Windows platform.
- **blocks (12):** The number of blocks allocated to the file.

Consider the example shown in Listing 10-1.

Listing 10-1. *Retrieving Key File Information*

```
<?php

   /* Convert timestamp to desired format. */
   function tstamp_to_date($tstamp) {
      return date("m-d-y  g:i:sa", $tstamp);
   }

   $file = "/usr/local/apache2/htdocs/book/chapter10/stat.php";
   /* Open the file */
   $fh = fopen($file, "r");

   /* Retrieve file information */
   $fileinfo = fstat($fh);

   /* Output some juicy information about the file. */
   echo "Filename: ".basename($file)."<br />";
   echo "Filesize: ".round(($fileinfo["size"]/1024), 2)." kb <br />";
   echo "Last accessed: ".tstamp_to_date($fileinfo["atime"])."<br />";
   echo "Last modified: ".tstamp_to_date($fileinfo["mtime"])."<br />";
?>
```

This code returns:

```
Filename: stat.php
Filesize: 2.16 kb
Last accessed: 06-09-05 12:03:00pm
Last modified: 06-09-05 12:02:59pm
```

stat()

```
array stat (string filename)
```

The `stat()` function returns an array of useful information about the file specified by `filename`, or `FALSE` if it fails. This function operates exactly like `fstat()`, returning all of the same array elements; the only difference is that `stat()` requires an actual file name and path rather than a resource handle.

If `filename` is a symbolic link, then the information will be pertinent to the file the symbolic link points to, and not the symbolic link itself. To retrieve information about a symbolic link, use `lstat()`, introduced a bit earlier in this chapter.

readlink()

```
string readlink (string path)
```

The `readlink()` function returns the target of the symbolic link specified by `path`, or `FALSE` if an error occurs. Therefore, if link `test-link.txt` is a symbolic link pointing to `test.txt`, the following will return the absolute pathname to the file:

```
echo readlink("/home/jason/test-link.txt");
// returns /home/jason/myfiles/test.txt
```

symlink()

```
int symlink (string target, string link)
```

The `symlink()` function creates a symbolic link named `link` to the existing `target`, returning `TRUE` on success and `FALSE` otherwise. Note that because PHP scripts typically execute under the guise of the server daemon process owner, this function will fail unless that daemon owner has write permissions within the directory in which `link` is to reside. Consider this example, in which symbolic link "03" is pointed to the directory "2003":

```
<?php
     $link = symlink("/www/htdocs/stats/2003", "/www/htdocs/stats/03");
?>
```

Calculating File, Directory, and Disk Sizes

Calculating file, directory, and disk sizes is a common task in all sorts of applications. This section introduces a number of standard PHP functions suited to this task.

filesize()

```
int filesize (string filename)
```

The `filesize()` function returns the size, in bytes, of `filename`. An example follows:

```
<?php
   $file = "/www/htdocs/book/chapter1.pdf";
   $bytes = filesize("$file"); // Returns 91815
   echo "File ".basename($file)." is $bytes bytes, or
        ".round($bytes / 1024, 2)." kilobytes.";
?>
```

This returns the following:

```
File 852Chapter16R.rtf is 91815 bytes, or 89.66 kilobytes
```

disk_free_space()

```
float disk_free_space (string directory)
```

The `disk_free_space()` function returns the available space, in bytes, allocated to the disk partition housing the directory specified by `directory`. An example follows:

```
<?php
     $drive = "/usr";
     echo round((disk_free_space($drive) / 1048576), 2);
?>
```

This returns:

```
2141.29
```

Note that the returned number is in megabytes (MB), because the value returned from `disk_free_space()` was divided by 1,048,576, which is equivalent to 1MB.

disk_total_space()

```
float disk_total_space (string directory)
```

The `disk_total_space()` function returns the total size, in bytes, consumed by the disk partition housing the directory specified by `directory`. If you use this function in conjunction with `disk_free_space()`, it's easy to offer useful space allocation statistics:

```
<?php
   $systempartitions = array("/", "/home","/usr", "/www");
   foreach ($systempartitions as $partition) {
      $totalSpace = disk_total_space($partition) / 1048576;
      $usedSpace = $totalSpace - disk_free_space($partition) / 1048576;
      echo "Partition: $partition (Allocated: $totalSpace MB.
            Used: $usedSpace MB.)";
   }
?>
```

This returns:

```
Partition: / (Allocated: 3099.292 MB. Used: 343.652 MB.)
Partition: /home (Allocated: 5510.664 MB. Used: 344.448 MB.)
Partition: /usr (Allocated: 4127.108 MB. Used: 1985.716 MB.)
Partition: /usr/local/apache2/htdocs (Allocated: 4127.108 MB. Used: 1985.716 MB.)
```

Retrieving a Directory Size

PHP doesn't currently offer a standard function for retrieving the total size of a directory, a task more often required than retrieving total disk space (see `disk_total_space()`). And although you could make a system-level call to `du` using `exec()` or `system()` (both of which are introduced later in this chapter), such functions are often disabled for security reasons. The alternative solution is to write a custom PHP function that is capable of carrying out this task. A recursive function seems particularly well-suited for this task. One possible variation is offered in Listing 10-2.

■ **Note** The `du` command will summarize disk usage of a file or directory. See the appropriate man page for usage information.

Listing 10-2. *Determining the Size of a Directory's Contents*

```
<?php
   function directory_size($directory) {
      $directorySize=0;

      /* Open the directory and read its contents. */
      if ($dh = @opendir($directory)) {

         /* Iterate through each directory entry. */
         while (($filename = readdir ($dh))) {

            /* Filter out some of the unwanted directory entries. */
            if ($filename != "." && $filename != "..")
            {

               // File, so determine size and add to total.
               if (is_file($directory."/".$filename))
                  $directorySize += filesize($directory."/".$filename);

                  // New directory, so initiate recursion. */
                  if (is_dir($directory."/".$filename))
                     $directorySize += directory_size($directory."/".$filename);
            }
         } #endWHILE
     } #endIF

     @closedir($dh);
     return $directorySize;

   } #end directory_size()
   $directory = "/usr/local/apache2/htdocs/book/chapter10/";
   $totalSize = round((directory_size($directory) / 1024), 2);
   echo "Directory $directory: ".$totalSize. "kb.";

?>
```

Access and Modification Times

The ability to determine a file's last access and modification time plays an important role in many administrative tasks, especially in Web applications that involve network or CPU-intensive update operations. PHP offers three functions for determining a file's access, creation, and last modification time, all of which are introduced in this section.

fileatime()

`int fileatime (string *filename*)`

The `fileatime()` function returns `filename`'s last access time in Unix timestamp format, or `FALSE` on error. An example follows:

```
<?php
   $file = "/usr/local/apache2/htdocs/book/chapter10/stat.php";
   echo "File last accessed: ".date("m-d-y  g:i:sa", fileatime($file));
?>
```

This returns:

```
File last accessed: 06-09-03 1:26:14pm
```

filectime()

`int filectime (string *filename*)`

The `filectime()` function returns `filename`'s last changed time in Unix timestamp format, or `FALSE` on error. An example follows:

```
<?php
   $file = "/usr/local/apache2/htdocs/book/chapter10/stat.php";
   echo "File inode last changed: ".date("m-d-y  g:i:sa", fileatime($file));
?>
```

This returns:

```
File inode last changed: 06-09-03 1:26:14pm
```

■Note The “last changed time” differs from the “last modified time” in that the last changed time refers to any change in the file’s inode data, including changes to permissions, owner, group, or other inode-specific information, whereas the last modified time refers to changes to the file’s content (specifically, byte size).

filemtime()

`int filemtime (string` *`filename`*`)`

The `filemtime()` function returns `filename`’s last modification time in Unix timestamp format, or `FALSE` otherwise. The following code demonstrates how to place a “last modified” timestamp on a Web page:

```
<?php
   $file = "/usr/local/apache2/htdocs/book/chapter10/stat.php";
   echo "File last updated: ".date("m-d-y  g:i:sa", filemtime($file));
?>
```

This returns:

```
File last updated: 06-09-03 1:26:14pm
```

File Ownership and Permissions

These days, security is paramount to any server installation, large or small. Most modern operating systems have embraced the concept of the separation of file rights via a user/group ownership paradigm, which, when properly configured, offers a wonderfully convenient and powerful means for securing data. In this section, you’ll learn how to use PHP’s built-in functionality to review and manage these permissions.

Note that because PHP scripts typically execute under the guise of the server daemon process owner, some of these functions will fail unless highly insecure actions are taken to run the server as a privileged user. Thus, keep in mind that some of the functionality introduced in this chapter is much better suited for use when running PHP as a command-line interface (CLI), since scripts executed by way of the CLI could conceivably be run as any system user.

chown()

`int chown (string` *`filename`*`, mixed` *`user`*`)`

The `chown()` function attempts to change the owner of `filename` to `user` (specified either by the user’s username or UID), returning `TRUE` on success and `FALSE` otherwise.

chgrp()

`int chgrp (string filename, mixed group)`

The `chgrp()` function attempts to change the group membership of `filename` to `group`, returning `TRUE` on success and `FALSE` otherwise.

fileperms()

`int fileperms (string filename)`

The `fileperms()` function returns `filename`'s permissions in decimal format, or `FALSE` in case of error. Because the decimal permissions representation is almost certainly not the desired format, you'll need to convert `fileperms()`'s return value. This is easily accomplished using the `base_convert()` function in conjunction with `substr()`. The `base_convert()` function converts a value from one number base to another; therefore, you can use it to convert `fileperms()`'s returned decimal value from base 10 to the desired base 8. The `substr()` function is then used to retrieve only the final three digits of `base_convert()`'s returned value, which are the only digits referred to when discussing Unix file permissions. Consider the following example:

```
<?php
   echo substr(base_convert(fileperms("/etc/passwd"), 10, 8), 3);
?>
```

This returns:

```
644
```

filegroup()

`int filegroup (string filename)`

The `filegroup()` function returns the group ID (GID) of the `filename` owner, and `FALSE` if the GID cannot be determined:

```
<?php
   $gid = filegroup("/etc/passwd");
   // Returns "0" on Unix, because root usually has GID of 0.
?>
```

Note that `filegroup()` returns the GID, and not the group name.

fileowner()

`int fileowner (string filename)`

The `fileowner()` function returns the user ID (UID) of the `filename` owner, or `FALSE` if the UID cannot be determined. Consider this example:

```
<?php
   $uid = fileowner("/etc/passwd");
   // Returns "0" on Linux, as root typically has UID of 0.
?>
```

Note that `fileowner()` returns the UID, and not the username.

isexecutable()

```
boolean isexecutable (string filename)
```

The `isexecutable()` function returns `TRUE` if `filename` exists and is executable, and `FALSE` otherwise. Note that this function is not available on the Windows platform.

isreadable()

```
boolean isreadable (string filename)
```

The `isreadable()` function returns `TRUE` if `filename` exists and is readable, and `FALSE` otherwise. If a directory name is passed in as `filename`, `isreadable()` will determine whether that directory is readable.

iswriteable()

```
boolean iswriteable (string filename)
```

The `iswriteable()` function returns `TRUE` if `filename` exists and is writable, and `FALSE` otherwise. If a directory name is passed in as `filename`, `iswriteable()` will determine whether that directory is writable.

■ **Note** The function `iswritable()` is an alias of `iswriteable()`.

---

umask()

```
int umask ([int mask])
```

The `umask()` function determines the level of permissions assigned to a newly created file. The `umask()` function calculates PHP's umask to be the result of `mask` bitwise ANDed with 0777, and returns the old mask. Keep in mind that `mask` is a three- or four-digit code representing the permission level. PHP then uses this umask when creating files and directories throughout the script. Omitting the optional parameter `mask` results in the retrieval of PHP's currently configured umask value.

File I/O

Writing exciting, useful programs almost always requires that the program work with some sort of external data source. Two prime examples of such data sources are files and databases. In this section, we delve deep into working with files. Before we introduce PHP's numerous standard file-related functions, however, it's worth introducing a few basic concepts pertinent to this topic.

The Concept of a Resource

The term "resource" is commonly attached to any entity from which an input or output stream can be initiated. Standard input or output, files, and network sockets are all examples of resources.

Newline

The newline character, which is represented by the `\n` character sequence, represents the end of a line within a file. Keep this in mind when you need to input or output information one line at a time. Several functions introduced throughout the remainder of this chapter offer functionality tailored to working with the newline character. Some of these functions include `file()`, `fgetcsv()`, and `fgets()`.

End-of-File

Programs require a standardized means for discerning when the end of a file has been reached. This standard is commonly referred to as the end-of-file, or EOF, character. This is such an important concept that almost every mainstream programming language offers a built-in function for verifying whether or not the parser has arrived at the EOF. In the case of PHP, this function is `feof()`, described next.

feof()

```
int feof (string resource)
```

The `feof()` function determines whether `resource`'s EOF has been reached. It is used quite commonly in file I/O operations. An example follows:

```
<?php
   $fh = fopen("/home/www/data/users.txt", "rt");
   while (!feof($fh)) echo fgets($fh);
   fclose($fh);
?>
```

Opening and Closing a File

You'll often need to establish a connection to a file resource before you can do anything with its contents. Likewise, once you've finished working with that resource, you should close the connection. Two standard functions are available for such tasks, both of which are introduced in this section.

fopen()

```
resource fopen (string resource, string mode [, int use_include_path
               [, resource zcontext]])
```

The `fopen()` function binds a `resource` to a stream, or handler. Once bound, the script can interact with this resource via the handle. Most commonly, it's used to open files for reading and manipulation. However, `fopen()` is also capable of opening resources via a number of protocols, including HTTP, HTTPS, and FTP, a concept discussed in Chapter 16.

The `mode`, assigned at the time a `resource` is opened, determines the level of access available to that resource. The various modes are defined in Table 10-1.

Table 10-1. *File Modes*

Mode	Description
`r`	Read-only. The file pointer is placed at the beginning of the file.
`r+`	Read and write. The file pointer is placed at the beginning of the file.
`w`	Write only. Before writing, delete the file contents and return the file pointer to the beginning of the file. If the file does not exist, attempt to create it.
`w+`	Read and write. Before reading or writing, delete the file contents and return the file pointer to the beginning of the file. If the file does not exist, attempt to create it.
`a`	Write only. The file pointer is placed at the end of the file. If the file does not exist, attempt to create it. This mode is better known as Append.
`a+`	Read and write. The file pointer is placed at the end of the file. If the file does not exist, attempt to create it. This process is known as appending to the file.
`b`	Open the file in binary mode.
`t`	Open the file in text mode.

If the resource is found on the local file system, PHP expects the resource to be available by either the local or relative path prefacing it. Alternatively, you can assign `fopen()`'s `use_include_path` parameter the value of `1`, which will cause PHP to consider the paths specified in the `include_path` configuration directive.

The final parameter, `zcontext`, is used for setting configuration parameters specific to the file or stream, and for sharing file- or stream-specific information across multiple `fopen()` requests. This topic is discussed in further detail in Chapter 16.

Let's consider a few examples. The first opens a read-only stream to a text file residing on the local server:

```
$fh = fopen("/usr/local/apache/data/users.txt","rt");
```

The next example demonstrates opening a write stream to a Microsoft Word document. Because Word documents are binary, you should specify the binary `b` mode variation.

```
$fh = fopen("/usr/local/apache/data/docs/summary.doc","wb");
```

The next example refers to the same Word document, except this time PHP will search for the file in the paths specified by the `include_path` directive:

```
$fh = fopen("summary.doc","wb", 1);
```

The final example opens a read-only stream to a remote `index.html` file:

```
$fh = fopen("http://www.example.com/", "rt");
```

You'll see this function in numerous examples throughout this and the next chapter.

fclose()

```
boolean fclose (resource filehandle)
```

Good programming practice dictates that you should destroy pointers to any resources once you're finished with them. The `fclose()` function handles this for you, closing the previously opened file pointer specified by `filehandle`, returning `TRUE` on success and `FALSE` otherwise. The `filehandle` must be an existing file pointer opened using `fopen()` or `fsockopen()`.

Reading from a File

PHP offers numerous methods for reading data from a file, ranging from reading in just one character at a time to reading in the entire file with a single operation. Many of the most useful functions are introduced in this section.

file()

```
array file (string filename [int use_include_path [, resource context]])
```

The immensely useful `file()` function is capable of reading a file into an array, separating each element by the newline character, with the newline still attached to the end of each element. Although simplistic, the importance of this function can't be understated, and therefore it warrants a simple demonstration. Consider the following sample text file, named `users.txt`:

```
Ale ale@example.com
Nicole nicole@example.com
Laura laura@example.com
```

The following script reads in `users.txt` and parses and converts the data into a convenient Web-based format:

```
<?php
   $users = file("users.txt");

   foreach ($users as $user) {
      list($name, $email) = explode(" ", $user);

      // Remove newline from $email
      $email = trim($email);
      echo "<a href=\"mailto:$email\">$name</a> <br />\n";
   }
?>
```

This script produces the following HTML output:

```
<a href="ale@example.com">Ale</a><br />
<a href="nicole@example.com">Nicole</a><br />
<a href="laura@example.com">Laura</a><br />
```

Like `fopen()`, you can tell `file()` to search through the paths specified in the `include_path` configuration parameter by setting `use_include_path` to `1`. The `context` parameter refers to a stream context. You'll learn more about this topic in Chapter 16.

file_get_contents()

```
string file_get_contents (string filename [, int use_include_path
                         [resource context]])
```

The `file_get_contents()` function reads the contents of `filename` into a string. By revising the script from the preceding section to use this function instead of `file()`, you get the following code:

```
<?php
   $userfile= file_get_contents("users.txt");
   // Place each line of $userfile into array
   $users = explode("\n",$userfile);
   foreach ($users as $user) {
      list($name, $email) = explode(" ", $user);
      echo "<a href=\"mailto:$email\">$name/a> <br />";
   }
?>
```

The `context` parameter refers to a stream context. You'll learn more about this topic in Chapter 16.

fgetc()

```
string fgetc (resource handle)
```

The `fgetc()` function reads a single character from the open resource stream specified by `handle`. If the EOF is encountered, a value of `FALSE` is returned.

fgetcsv()

```
array fgetcsv (resource handle, int length [, string delimiter
             [, string enclosure]])
```

The convenient `fgetcsv()` function parses each line of a file specified by `handle` and delimited by `delimiter`, placing each field into an array. Reading does not stop on a newline; rather, it stops either when `length` characters have been read or when the closing `enclosure` character is located. Therefore, it is always a good idea to choose a number that will certainly surpass the longest line in the file.

Consider a scenario in which weekly newsletter subscriber data is cached to a file for perusal by the corporate marketing staff. Always eager to barrage the IT department with dubious requests, the marketing staff asks that the information also be made available for viewing on the Web. Thankfully, this is easily accomplished with `fgetcsv()`. The following example parses the already cached file:

```
<?php
   $fh = fopen("/home/www/data/subscribers.csv", "r");
   while (list($name, $email, $phone) = fgetcsv($fh, 1024, ",")) {
      echo "<p>$name ($email) Tel. $phone</p>";
   }
?>
```

Note that you don't have to use `fgetcsv()` to parse such files; the `file()` and `list()` functions accomplish the job quite nicely. Reconsidering the preceding example:

```
<?php
   $users = file("users.txt");
   foreach ($users as $user) {
      list($name, $email, $phone) = explode(",", $user);
      echo "<p>$name ($email) Tel. $phone</p>";
   }
?>
```

■ **Note** Comma-separated value (CSV) files are commonly used when importing files between applications. Microsoft Excel and Access, MySQL, Oracle, and PostgreSQL are just a few of the applications and databases capable of both importing and exporting CSV data. Additionally, languages such as Perl, Python, and PHP are particularly efficient at parsing delimited data.

fgets()

`fgets (resource` *`handle`* `[, int` *`length`*`])`

The `fgets()` function returns either `length` – 1 bytes from the opened resource referred to by `handle`, or everything it has read up to the point that a newline or the EOF is encountered. If the optional `length` parameter is omitted, 1,024 characters is assumed. In most situations, this means that `fgets()` will encounter a newline character before reading 1,024 characters, thereby returning the next line with each successive call. An example follows:

```
<?php
   $fh = fopen("/home/www/data/users.txt", "rt");
   while (!feof($fh)) echo fgets($fh);
   fclose($fh);
?>
```

fgetss()

`string fgetss (resource handle, int length [, string allowable_tags])`

The `fgetss()` function operates similarly to `fgets()`, except that it strips any HTML and PHP tags from `handle`. If you'd like certain tags to be ignored, include them in the `allowable_tags` parameter. As an example, consider a scenario in which authors are expected to submit their work in HTML format using a specified subset of HTML tags. Of course, the authors don't always follow instructions, so the file must be scanned for tag misuse before it can be published. With `fgetss()`, this is trivial:

```
<?php
   /* Build list of acceptable tags */
   $tags = "<h2><h3><p><b><a><img>";

   /* Open the article, and read its contents. */
   $fh = fopen("article.html", "rt");

   while (!feof($fh)) {
      $article .= fgetss($fh, 1024, $tags);
   }
   fclose($fh);

   /* Open the file up in write mode
      and write $article contents. */
   $fh = fopen("article.html", "wt");
   fwrite($fh, $article);
   fclose($fh);
?>
```

■**Tip** If you want to remove HTML tags from user input submitted via a form, check out the `strip_tags()` function, introduced in Chapter 9.

fread()

`string fread (resource handle, int length)`

The `fread()` function reads `length` characters from the resource specified by `handle`. Reading stops when the EOF is reached or when `length` characters have been read. Note that, unlike other read functions, newline characters are irrelevant when using `fread()`; therefore, it's often convenient to read the entire file in at once using `filesize()` to determine the number of characters that should be read in:

```
<?php
   $file = "/home/www/data/users.txt";
   $fh = fopen($file, "rt");
   $userdata = fread($fh, filesize($file));
   fclose($fh);
?>
```

The variable `$userdata` now contains the contents of the `users.txt` file.

readfile()

`int readfile (string` *`filename`* `[, int` *`use_include_path`*`])`

The `readfile()` function reads an entire file specified by `filename` and immediately outputs it to the output buffer, returning the number of bytes read. Enabling the optional `use_include_path` parameter tells PHP to search the paths specified by the `include_path` configuration parameter. After sanitizing the article discussed in the `fgetss()` section, it can be output to the browser quite easily using `readfile()`. This revised example is shown here:

```
<?php
   $file = "/home/www/articles/gilmore.html";

   /* Build list of acceptable tags */
   $tags = "<h2><h3><p><b><a><img>";

   /* Open the article, and read its contents. */
   $fh = fopen($file, "rt");

   while (!feof($fh))
      $article .= fgetss($fh, 1024, $tags);

   fclose($fh);

   /* Open the article, overwriting it with the sanitized material */
   $fh = fopen($file, "wt");
   fwrite($fh, $article);
   fclose($fh);

   /* Output the article to the browser. */
   $bytes = readfile($file);
?>
```

Like many of PHP's other file I/O functions, remote files can be opened via their URL if the configuration parameter `fopen_wrappers` is enabled.

fscanf()

`mixed fscanf (resource` *`handle`*`, string` *`format`* `[, string` *`var1`*`])`

The `fscanf()` function offers a convenient means for parsing the resource specified by `handle` in accordance with the format specified by `format`. Suppose you want to parse the following file consisting of social security (SSN) numbers (`socsecurity.txt`):

```
123-45-6789
234-56-7890
345-67-8901
```

The following example parses the `socsecurity.txt` file:

```
<?php
   $fh = fopen("socsecurity.txt", "r");

   /* Parse each SSN in accordance with
      integer-integer-integer format. */

   while ($user = fscanf($fh, "%d-%d-%d")) {
      list ($part1,$part2,$part3) = $user;
      ...
   }

   fclose($fh);
?>
```

With each iteration, the variables `$part1`, `$part2`, and `$part3` are assigned the three components of each SSN, respectively.

Moving the File Pointer

It's often useful to jump around within a file, reading from and writing to various locations. Several PHP functions are available for doing just this.

fseek()

```
int fseek (resource handle, int offset [, int whence])
```

The `fseek()` function moves the `handle`'s pointer to the location specified by `offset`. If the optional parameter `whence` is omitted, the position is set `offset` bytes from the beginning of the file. Otherwise, `whence` can be set to one of three possible values, which affect the pointer's position:

- `SEEK_CUR`: Sets the pointer position to the current position plus `offset` bytes.
- `SEEK_END`: Sets the pointer position to the EOF plus `offset` bytes. In this case, `offset` must be set to a negative value.
- `SEEK_SET`: Sets the pointer position to `offset` bytes. This has the same effect as omitting `whence`.

ftell()

```
int ftell (resource handle)
```

The `ftell()` function retrieves the current position of the file pointer's offset within the resource specified by `handle`.

rewind()

```
int rewind (resource handle)
```

The `rewind()` function moves the file pointer back to the beginning of the resource specified by `handle`.

Writing to a File

This section highlights several of the functions used to output data to a file.

fwrite()

```
int fwrite (resource handle, string string [, int length])
```

The `fwrite()` function outputs the contents of `string` to the resource pointed to by `handle`. If the optional `length` parameter is provided, `fwrite()` will stop writing when `length` characters have been written. Otherwise, writing will stop when the end of the `string` is found. Consider this example:

```
<?php
   $subscriberInfo = "Jason Gilmore|wj@example.com";
   $fh = fopen("/home/www/data/subscribers.txt", "at");
   fwrite($fh, $subscriberInfo);
   fclose($fh);
?>
```

■ **Tip** If the optional `length` parameter is not supplied to `fwrite()`, the `magic_quotes_runtime` configuration parameter will be disregarded. See Chapters 2 and 9 for more information about this parameter.

fputs()

```
int fputs (resource handle, string string [, int length])
```

The `fputs()` function operates identically to `fwrite()`. Presumably, it was incorporated into the language to satisfy the terminology preferences of C/C++ programmers.

Reading Directory Contents

The process required for reading a directory's contents is quite similar to that involved in reading a file. This section introduces the functions available for this task, and also introduces a function new to PHP 5 that reads a directory's contents into an array.

opendir()

```
resource opendir (string path)
```

Just as `fopen()` opens a file pointer to a given file, `opendir()` opens a directory stream specified by `path`.

closedir()

```
void closedir (resource directory_handle)
```

The `closedir()` function closes the directory stream pointed to by `directory_handle`.

readdir()

```
string readdir (int directory_handle)
```

The `readdir()` function returns each element in the directory specified by `directory_handle`. You can use this function to list all files and child directories in a given directory:

```
<?php
   $dh = opendir('/usr/local/apache2/htdocs/');
   while ($file = readdir($dh))
      echo "$file <br>";
   closedir($dh);
?>
```

Sample output follows:

```
.
..
articles
images
news
test.php
```

Note that `readdir()` also returns the `.` and `..` entries common to a typical Unix directory listing. You can easily filter these out with an `if` statement:

```
if($file != "." AND $file != "..")...
```

scandir()

```
array scandir (string directory [,int sorting_order [, resource context]])
```

The `scandir()` function, which is new to PHP 5, returns an array consisting of files and directories found in `directory`, or returns `FALSE` on error. Setting the optional `sorting_order` parameter to 1 sorts the contents in descending order, overriding the default of ascending order. Revisiting the example from the previous section:

```
<?php
   print_r(scandir("/usr/local/apache2/htdocs"));
?>
```

This returns:

```
Array ( [0] => . [1] => .. [2] => articles [3] => images
[4] => news [5] => test.php )
```

The `context` parameter refers to a stream context. You'll learn more about this topic in Chapter 16.

Executing Shell Commands

The ability to interact with the underlying operating system is a crucial feature of any programming language. This section introduces PHP's capabilities in this regard.

PHP's Built-in System Commands

Although you could conceivably execute any system-level command using a function like `exec()` or `system()`, some of these functions are so commonplace that the developers thought it a good idea to incorporate them directly into the language. Several such functions are introduced in this section.

rmdir()

```
int rmdir (string dirname)
```

The `rmdir()` function removes the directory specified by `dirname`, returning `TRUE` on success and `FALSE` otherwise. As with many of PHP's file system functions, permissions must be properly set in order for `rmdir()` to successfully remove the directory. Because PHP scripts typically execute under the guise of the server daemon process owner, `rmdir()` will fail unless that user has write permissions to the directory. Also, the directory must be empty.

To remove a nonempty directory, you can either use a function capable of executing a system-level command, like `system()` or `exec()`, or write a recursive function that will remove all file contents before attempting to remove the directory. Note that in either case, the executing user (server daemon process owner) requires write access to the parent of the target directory. Here is an example of the latter approach:

```
<?php
   function delete_directory($dir)
   {
      if ($dh = @opendir($dir))
      {

         /* Iterate through directory contents. */
         while (($file = readdir ($dh)) != false)
         {
            if (($file == ".") || ($file == "..")) continue;
            if (is_dir($dir . '/' . $file))
               delete_directory($dir . '/' . $file);
            else
               unlink($dir . '/' . $file);
          } #endWHILE

          @closedir($dh);
          rmdir($dir);
     } #endIF
   } #end delete_directory()

   $dir = "/usr/local/apache2/htdocs/book/chapter10/test/";
   delete_directory($dir);
?>
```

rename()

`boolean rename (string` *`oldname`*`, string` *`newname`*`)`

The `rename()` function renames a file specified by `oldname` to the new name `newname`, returning `TRUE` on success and `FALSE` otherwise. Because PHP scripts typically execute under the guise of the server daemon process owner, `rename()` will fail unless that user has write permissions to that file.

touch()

`int touch (string` *`filename`* `[, int` *`time`* `[, int` *`atime`*`]])`

The `touch()` function sets the file `filename`'s last-modified and last-accessed times, returning `TRUE` on success or `FALSE` on error. If `time` is not provided, the present time (as specified by the server) is used. If the optional `atime` parameter is provided, the access time will be set to this value; otherwise, like the modification time, it will be set to either `time` or the present server time.

Note that if `filename` does not exist, it will be created, assuming that the script's owner possesses adequate permissions.

System-Level Program Execution

Truly lazy programmers know how to make the most of their entire server environment when developing applications, which includes exploiting the functionality of the operating system, file system, installed program base, and programming languages whenever necessary. In this section, you'll learn how PHP can interact with the operating system to call both OS-level programs and third-party installed applications. Done properly, it adds a whole new level of functionality to your PHP programming repertoire. Done poorly, it can be catastrophic not only to your application, but also to your server's data integrity. That said, before delving into this powerful feature, take a moment to consider the topic of sanitizing user input before passing it to the shell level.

Sanitizing the Input

Neglecting to sanitize user input that may subsequently be passed to system-level functions could allow attackers to do massive internal damage to your information store and operating system, deface or delete Web files, and otherwise gain unrestricted access to your server. And that's only the beginning.

■ **Note** See Chapter 21 for a discussion of secure PHP programming.

As an example of why sanitizing the input is so important, consider a real-world scenario. Suppose that you offer an online service that generates PDFs from an input URL. A great tool for accomplishing just this is HTMLDOC, a program that converts HTML documents to indexed HTML, Adobe PostScript, and PDF files. HTMLDOC (`http://www.htmldoc.org/`) is released under the GNU General Public License. HTMLDOC can be invoked from the command line, like so:

```
%>htmldoc --webpage –f webpage.pdf http://www.wjgilmore.com/
```

This would result in the creation of a PDF named `webpage.pdf`, which would contain a snapshot of the Web site's index page. Of course, most users will not have command-line access to your server; therefore, you'll need to create a much more controlled interface to the service, perhaps the most obvious of which being via a Web page. Using PHP's `passthru()` function (introduced later in this chapter), you can call HTMLDOC and return the desired PDF, like so:

```
$document = $_POST['userurl'];
passthru("htmldoc --webpage -f webpage.pdf $document);
```

What if an enterprising attacker took the liberty of passing through additional input, unrelated to the desired HTML page, entering something like this:

```
http://www.wjgilmore.com/ ; cd /usr/local/apache/htdocs/; rm -rf *
```

Most Unix shells would interpret the `passthru()` request as three separate commands. The first is:

```
htmldoc --webpage -f webpage.pdf http://www.wjgilmore.com/
```

The second command is:

```
cd /usr/local/apache/htdocs/
```

And the final command is:

```
rm -rf *
```

Those last two commands were certainly unexpected, and could result in the deletion of your entire Web document tree. One way to safeguard against such attempts is to sanitize user input before it is passed to any of PHP's program execution functions. Two standard functions are conveniently available for doing so: `escapeshellarg()` and `escapeshellcmd()`. Each is introduced in this section.

escapeshellarg()

```
string escapeshellarg (string arguments)
```

The `escapeshellarg()` function delimits `arguments` with single quotes and prefixes (escapes) quotes found within `arguments`. The effect is that when `arguments` is passed to a shell command, it will be considered a single argument. This is significant because it lessens the possibility that an attacker could masquerade additional commands as shell command arguments. Therefore, in the aforementioned nightmarish scenario, the entire user input would be enclosed in single quotes, like so:

```
'http://www.wjgilmore.com/ ; cd /usr/local/apache/htdoc/; rm -rf *'
```

The result would be that HTMLDOC would simply return an error, because it could not resolve a URL possessing this syntax, rather than delete an entire directory tree.

escapeshellcmd()

```
string escapeshellcmd (string command)
```

The `escapeshellcmd()` function operates under the same premise as `escapeshellarg()`, sanitizing potentially dangerous input by escaping shell metacharacters. These characters include the following: # & ; ` , | * ? , ~ < > ^ ( ) [ ] { } $ \\.

PHP's Program Execution Functions

This section introduces several functions (in addition to the backticks execution operator) used to execute system-level programs via a PHP script. Although at first glance they all appear to be operationally identical, each offers its own syntactical nuances.

exec()

```
string exec (string command [, array output [, int return_var]])
```

The `exec()` function is best-suited for executing an operating system–level application (designated by `command`) intended to continue executing in the server background. Although the last line of output will be returned, chances are that you'd like to have all of the output returned for review; you can do this by including the optional parameter `output`, which will be populated with each line of output upon completion of the command specified by `exec()`. In addition, you can discover the executed command's return status by including the optional parameter `return_var`.

Although we could take the easy way out and demonstrate how `exec()` can be used to execute an `ls` command (`dir` for the Windows folks), returning the directory listing, it's more informative to offer a somewhat more practical example: how to call a Perl script from PHP. Consider the following Perl script (`languages.pl`):

```
#! /usr/bin/perl
my @languages = qw[perl php python java c];
foreach $language (@languages) {
     print $language."<br />";
}
```

The Perl script is quite simple; no third-party modules are required, so you could test this example with little time investment. If you're running Linux, chances are very good that you could run this example immediately, because Perl is installed on every respectable distribution. If you're running Windows, check out ActiveState's (`http://www.activestate.com/`) ActivePerl distribution.

Like `languages.pl`, the PHP script shown here isn't exactly rocket science; it simply calls the Perl script, specifying that the outcome be placed into an array named `$results`. The contents of `$results` are then output to the browser.

```
<?php
   $outcome = exec("languages.pl", $results);
   foreach ($results as $result) echo $result;
?>
```

The results are as follows:

```
perl
php
python
java
c
```

system()

```
string system (string command [, int return_var])
```

The `system()` function is useful when you want to output the executed command's results. Rather than return output via an optional parameter, as is the case with `exec()`, the output is returned directly to the caller. However, if you would like to review the execution status of the called program, you need to designate a variable using the optional parameter `return_var`.

For example, suppose you'd like to list all files located within a specific directory:

```
$mymp3s = system("ls -1 /home/jason/mp3s/");
```

Or, revising the previous PHP script to again call the `languages.pl` using `system()`:

```
<?php
   $outcome = exec("languages.pl", $results);
   echo $outcome
?>
```

passthru()

```
void passthru (string command [, int return_var])
```

The `passthru()` function is similar in function to `exec()`, except that it should be used if you'd like to return binary output to the caller. For example, suppose you want to convert GIF images to PNG before displaying them to the browser. You could use the Netpbm graphics package, available at `http://netpbm.sourceforge.net/` under the GPL license:

```
<?php
   header("ContentType:image/png");
   passthru("giftopnm cover.gif | pnmtopng > cover.png");
?>
```

Backticks

Delimiting a string with backticks signals to PHP that the string should be executed as a shell command, returning any output. Note that backticks are not single quotes, but rather are a slanted cousin, commonly sharing a key with the tilde (~) on most American keyboards. An example follows:

```
<?php
   $result = `date`;
   echo "<p>The server timestamp is: $result</p>";
?>
```

This returns something similar to:

```
The server timestamp is: Sun Jun 15 15:32:14 EDT 2003
```

The backtick operator is operationally identical to the `shellexec()` function, introduced next.

shell_exec()

```
string shell_exec (string command)
```

The `shell_exec()` function offers a syntactical alternative to backticks, executing a shell command and returning the output. Reconsidering the preceding example:

```
<?php
     $result = shell_exec("date");
     echo "<p>The server timestamp is: $result</p>";
?>
```

Summary

Although you can certainly go a very long way using solely PHP to build interesting and powerful Web applications, such capabilities are greatly expanded when functionality is integrated with the underlying platform and other technologies. As applied to this chapter, these technologies include the underlying operating and file systems. You'll see this theme repeatedly throughout the remainder of this book, as PHP's ability to interface with a wide variety of technologies like LDAP, SOAP, and Web Services is introduced.

In the next chapter, you'll examine two key aspects of any Web application: Web forms and navigational cues.