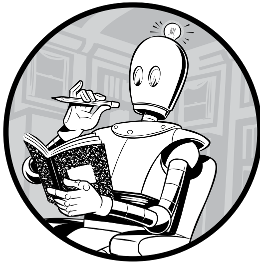
2

IMPROVING ON USER COMMANDS



A typical Unix or Linux system includes hundreds of commands by default, which, when you factor in flags and the possible ways to combine commands with pipes, produces millions of different ways to work on the command line.

Before we go any further, Listing 2-1 shows a bonus script that will tell you how many commands are in your PATH.

```
#!/bin/bash

# How many commands: a simple script to count how many executable
#   commands are in your current PATH


IFS=":"
count=0 ; nonex=0
for directory in $PATH ;  do
  if [ -d "$directory" ] ; then
```

```
    for command in "$directory"/* ; do
      if [ -x "$command" ] ; then
        count="$(( $count + 1 ))"
      else
        nonex="$(( $nonex + 1 ))"
      fi
    done
  fi
done

echo "$count commands, and $nonex entries that weren't executable"

exit 0
```

Listing 2-1: Counting the number of executables and nonexecutables in the current PATH

This script counts the number of executable files rather than just the number of files, and it can be used to reveal how many commands and nonexecutables are in the default PATH variables of many popular operating systems (see Table 2-1).

Table 2-1: Typical Command Count by OS

Operating system	Commands	Nonexecutables
Ubuntu 15.04 (including all developer libraries)	3,156	5
OS X 10.11 (with developer options installed)	1,663	11
FreeBSD 10.2	954	4
Solaris 11.2	2,003	15

Clearly, the different flavors of Linux and Unix offer a large number of commands and executable scripts. Why are there so many? The answer is based on the foundational Unix philosophy: commands should do one thing, and do it well. Word processors that have spellcheck, find file, and email capabilities might work well in the Windows and Mac world, but on the command line, each of these functions should be separate and discrete.

There are lots of advantages to this design philosophy, the most important being that each function can be modified and extended individually, giving all applications that utilize it access to these new capabilities. With any task you might want to perform on Unix, you can usually cobble together something that'll do the trick easily, whether by downloading some nifty utility that adds capabilities to your system, creating some aliases, or dipping a toe into the shell-scripting pond.

The scripts throughout the book not only are helpful but also are a logical extension of the Unix philosophy. After all, 'tis better to extend and expand than to build complex, incompatible versions of commands for your own installation.

The scripts explored in this chapter are all similar to the script in Listing 2-1 in that they add fun or useful features and capabilities without a high degree of complexity. Some of the scripts accept different command flags to allow even greater flexibility in their use, and some also demonstrate how a shell script can be used as a *wrapper*, a program that intercedes to allow users to specify commands or command flags in a common notation and then translates those flags into the proper format and syntax required by the actual Unix command.

#14 Formatting Long Lines

If you're lucky, your Unix system already includes the `fmt` command, a program that's remarkably useful if you work with text regularly. From reformatting emails to making lines use up all the available width in documents, `fmt` is a helpful utility to know.

However, some Unix systems don't include `fmt`. This is particularly true of legacy systems, which often have fairly minimalistic implementations.

As it turns out, the `nroff` command, which has been part of Unix since the very beginning and is a shell script wrapper in its own right, can be used in short shell scripts to wrap long lines and fill in short lines to even out line lengths, as shown in Listing 2-2.

The Code

```
#!/bin/bash

# fmt--Text formatting utility that acts as a wrapper for nroff
#   Adds two useful flags: -w X for line width
#   and -h to enable hyphenation for better fills
❶ while getopts "hw:" opt; do
  case $opt in
    h ) hyph=1             ;;
    w ) width="$OPTARG"    ;;
  esac
done
❷ shift $(($OPTIND - 1))

❸ nroff << EOF
❹ .ll ${width:-72}
.na
.hy ${hyph:-0}
.pl 1
❺ $(cat "$@")
EOF

exit 0
```

Listing 2-2: The `fmt` shell script for formatting long texts nicely

How It Works

This succinct script offers two different command flags: `-w X` to specify that lines should be wrapped when they exceed `X` characters (the default is 72) and `-h` to enable hyphenated word breaks across lines. Notice the check for flags at ❶. The `while` loop uses `getopts` to read each option passed to the script one at a time, and the inner `case` block decides what to do with them. Once the options are parsed, the script calls `shift` at ❷ to throw away all the option flags using `$OPTIND` (which holds the index of the next argument to be read by `getopts`) and leaves the remaining arguments to continue getting processed.

This script also makes use of a *here document* (discussed in Script #9 on page 34), which is a type of code block that can be used to feed multiple lines of input to a command. Using this notational convenience, the script at ❸ feeds `nroff` all the necessary commands to achieve the desired output. In this document, we use a bashism to replace a variable that isn't defined ❹, in order to provide a sane default value if the user does not specify one as an argument. Finally, the script calls the `cat` command with the requested file names to process. To complete the task, the `cat` command's output is also fed directly to `nroff` ❺. This is a technique that will appear frequently in the scripts presented in this book.

Running the Script

This script can be invoked directly from the command line, but it would more likely be part of an external pipe invoked from within an editor like `vi` or `vim` (for example, `!}fmt`) to format a paragraph of text.

The Results

Listing 2-3 enables hyphenation and specifies a maximum width of 50 characters.

```
$ fmt -h -w 50 014-ragged.txt
So she sat on, with closed eyes, and half believed
herself in Wonderland, though she knew she had but
to open them again, and all would change to dull
reality--the grass would be only rustling in the
wind, and the pool rippling to the waving of the
reeds--the rattling teacups would change to tin-
kling sheep-bells, and the Queen's shrill cries
to the voice of the shepherd boy--and the sneeze
of the baby, the shriek of the Gryphon, and all
the other queer noises, would change (she knew) to
the confused clamour of the busy farm-yard--while
the lowing of the cattle in the distance would
take the place of the Mock Turtle's heavy sobs.
```

Listing 2-3: Formatting text with the `fmt` script to hyphenate wrapped words at 50 characters

Compare Listing 2-3 (note the newly hyphenated word `tinkling`, highlighted on lines 6 and 7) with the output in Listing 2-4, generated using the default width and no hyphenation.

```
$ fmt 014-ragged.txt
So she sat on, with closed eyes, and half believed herself in
Wonderland, though she knew she had but to open them again, and all
would change to dull reality--the grass would be only rustling in the
wind, and the pool rippling to the waving of the reeds--the rattling
teacups would change to tinkling sheep-bells, and the Queen's shrill
cries to the voice of the shepherd boy--and the sneeze of the baby, the
shriek of the Gryphon, and all the other queer noises, would change (she
knew) to the confused clamour of the busy farm-yard--while the lowing of
the cattle in the distance would take the place of the Mock Turtle's
heavy sobs.
```

Listing 2-4: The default formatting of the `fmt` script with no hyphenation

#15 Backing Up Files as They're Removed

One of the most common problems that Unix users have is that there is no easy way to recover a file or folder that has been accidentally removed. There's no user-friendly application like Undelete 360, WinUndelete, or an OS X utility that allows you to easily browse and restore deleted files at the touch of a button. Once you press ENTER after typing `rm` *`filename`*, the file is history.

A solution to this problem is to secretly and automatically archive files and directories to a *.deleted-files* archive. With some fancy footwork in a script (as Listing 2-5 shows), this process can be made almost completely invisible to users.

The Code

```
#!/bin/bash

# newrm--A replacement for the existing rm command.
#   This script provides a rudimentary unremove capability by creating and
#   utilizing a new directory within the user's home directory. It can handle
#   directories of content as well as individual files. If the user specifies
#   the -f flag, files are removed and NOT archived.

# Big Important Warning: You'll want a cron job or something similar to keep
#   the trash directories tamed. Otherwise, nothing will ever actually
#   be deleted from the system, and you'll run out of disk space!

archivedir="$HOME/.deleted-files"
realrm="$(which rm)"
copy="$(which cp) -R"
```

```
if [ $# -eq 0 ] ; then             # Let 'rm' output the usage error.
  exec $realrm                     # Our shell is replaced by /bin/rm.
fi

# Parse all options looking for '-f'

flags=""

while getopts "dfiPRrvW" opt
do
  case $opt in
    f ) exec $realrm "$@"     ;;  # exec lets us exit this script directly.
    * ) flags="$flags -$opt"  ;;  # Other flags are for rm, not us.
  esac
done
shift $(( $OPTIND - 1 ))

# BEGIN MAIN SCRIPT
# =================

# Make sure that the $archivedir exists.

❶ if [ ! -d $archivedir] ; then
    if [ ! -w $HOME ] ; then
      echo "$0 failed: can't create $archivedir in $HOME" >&2
      exit 1
    fi
    mkdir $archivedir
❷   chmod 700 $archivedir            # A little bit of privacy, please.
  fi

  for arg
  do
❸   newname="$archivedir/$(date "+%S.%M.%H.%d.%m").$(basename "$arg")"
    if [ -f "$arg" -o -d "$arg" ] ; then
      $copy "$arg" "$newname"
    fi
  done

❹ exec $realrm $flags "$@"          # Our shell is replaced by realrm.
```

Listing 2-5: The `newrm` *shell script, which backs up files before they are deleted from the disk*

How It Works

There are a bunch of cool things to consider in this script, not the least of which is the significant effort it puts forth to ensure that users aren't aware it exists. For example, this script doesn't generate error messages in situations where it can't work; it just lets `realrm` generate them by invoking (typically) */bin/rm* with possibly bad parameters. The calls to `realrm` are done with the `exec` command, which replaces the current process with the new process specified. As soon as `exec` invokes `realrm` ❹, it effectively exits this script, and the return code from the `realrm` process is given to the invoking shell.

Because this script secretly creates a directory in the user's home directory ❶, it needs to ensure that the files there aren't suddenly readable by others simply because of a badly set `umask` value. (The `umask` value defines the default permissions for a newly created file or directory.) To avoid such oversharing, the script at ❷ uses `chmod` to ensure that the directory is set to read/write/execute for the user and is closed for everyone else.

Finally at ❸, the script uses `basename` to strip out any directory information from the file's path, and it adds a date- and timestamp to every deleted file in the form *second.minute.hour.day.month.filename*:

```
newname="$archivedir/$(date "+"%S.%M.%H.%d.%m").$(basename "$arg")"
```

Notice the use of multiple `$( )` elements in the same substitution. Though perhaps a bit complicated, it's nonetheless helpful. Remember, anything between `$(` and `)` is fed into a subshell, and the whole expression is then replaced by the result of that command.

So why bother with a timestamp anyway? To support storing multiple deleted files with the same name. Once the files are archived, the script makes no distinction between */home/oops.txt* and */home/subdir/oops.txt*, other than by the times they were deleted. If multiple files with same name are deleted simultaneously (or within the same second), the files that were archived first will get overwritten. One solution to this problem would be to add the absolute paths of the original files to the archived filenames.

Running the Script

To install this script, add an alias so that when you enter `rm`, you actually run this script, not the `/bin/rm` command. A bash or ksh alias would look like this:

```
alias rm=yourpath/newrm
```

The Results

The results of running this script are hidden by design (as Listing 2-6 shows), so let's keep an eye on the *.deleted-files* directory along the way.

```
$ ls ~/.deleted-files
ls: /Users/taylor/.deleted-files/: No such file or directory
$ newrm file-to-keep-forever
$ ls ~/.deleted-files/
51.36.16.25.03.file-to-keep-forever
```

Listing 2-6: Testing the `newrm` shell script

Exactly right. While the file was deleted from the local directory, a copy of it was secretly squirreled away in the *.deleted-files* directory. The timestamp allows other deleted files with the same name to be stored in the same directory without overwriting each other.

Hacking the Script

One useful tweak would be to change the timestamp so that it's in reverse time order to produce file listings from `ls` in chronological order. Here's the line to modify the script:

```
newname="$archivedir/$(date "+"%S.%M.%H.%d.%m").$(basename "$arg")"
```

You could reverse the order of tokens in that formatted request so that the original filename is first and the date is second in the backed-up filename. However, since our time granularity is seconds, you might remove more than one version of an identically named file within the same second (for example, `rm test testdir/test`), resulting in two identically named files. Therefore, another useful modification would be to incorporate the location of the file into the archived copy. This would produce, for example, *timestamp.test* and *timestamp.testdir.test*, which are clearly two different files.

#16 Working with the Removed File Archive

Now that a directory of deleted files is hidden within the user's home directory, a script to let the user choose between different versions of deleted files would be useful. However, it's quite a task to address all the possible situations, ranging from not finding the specified file at all to finding multiple deleted files that match the given criteria. In the case of more than one match, for example, should the script automatically pick the newest file to undelete? Throw an error indicating how many matches there are? Or present the different versions and let the user pick? Let's see what we can do with Listing 2-7, which details the `unrm` shell script.

The Code

```
#!/bin/bash

# unrm--Searches the deleted files archive for the specified file or
#   directory. If there is more than one matching result, it shows a list
#   of results ordered by timestamp and lets the user specify which one
#   to restore.

archivedir="$HOME/.deleted-files"
realrm="$(which rm)"
move="$(which mv)"

dest=$(pwd)

if [ ! -d $archivedir ] ; then
  echo "$0: No deleted files directory: nothing to unrm" >&2
  exit 1
fi
```

```
  cd $archivedir

  # If given no arguments, just show a listing of the deleted files.
❶ if [ $# -eq 0 ] ; then
    echo "Contents of your deleted files archive (sorted by date):"
❷   ls -FC | sed -e 's/\([[:digit:]][[:digit:]]\.\)\{5\}//g' \
      -e 's/^/  /'
    exit 0
  fi

  # Otherwise, we must have a user-specified pattern to work with.
  #   Let's see if the pattern matches more than one file or directory
  #   in the archive.

❸ matches="$(ls -d *"$1" 2> /dev/null | wc -l)"

  if [ $matches -eq 0 ] ; then
    echo "No match for \"$1\" in the deleted file archive." >&2
    exit 1
  fi

❹ if [ $matches -gt 1 ] ; then
    echo "More than one file or directory match in the archive:"
    index=1
    for name in $(ls -td *"$1")
    do
      datetime="$(echo $name | cut -c1-14| \
❺       awk -F. '{ print $5"/"$4" at "$3":"$2":"$1 }')"
      filename="$(echo $name | cut -c16-)"
      if [ -d $name ] ; then
❻      filecount="$(ls $name | wc -l | sed 's/[^[:digit:]]//g')"
        echo " $index)   $filename  (contents = ${filecount} items," \
             " deleted = $datetime)"
      else
❼       size="$(ls -sdk1 $name | awk '{print $1}')"
        echo " $index)   $filename  (size = ${size}Kb, deleted = $datetime)"
      fi
      index=$(( $index + 1))
    done
    echo ""
    /bin/echo -n "Which version of $1 should I restore ('0' to quit)? [1] : "
    read desired
    if [ ! -z "$(echo $desired | sed 's/[[:digit:]]//g')" ] ; then
      echo "$0: Restore canceled by user: invalid input." >&2
      exit 1
    fi

    if [ ${desired:=1} -ge $index ] ; then
      echo "$0: Restore canceled by user: index value too big." >&2
      exit 1
    fi
```

```
    if [ $desired -lt 1 ] ; then
      echo "$0: Restore canceled by user." >&2
      exit 1
    fi

❽   restore="$(ls -td1 *"$1" | sed -n "${desired}p")"

❾   if [ -e "$dest/$1" ] ; then
      echo "\"$1\" already exists in this directory. Cannot overwrite." >&2
      exit 1
    fi

    /bin/echo -n "Restoring file \"$1\" ..."
    $move "$restore" "$dest/$1"
    echo "done."

❿   /bin/echo -n "Delete the additional copies of this file? [y] "
    read answer

    if [ ${answer:=y} = "y" ] ; then
      $realrm -rf *"$1"
      echo "Deleted."
    else
      echo "Additional copies retained."
    fi
  else
    if [ -e "$dest/$1" ] ; then
      echo "\"$1\" already exists in this directory. Cannot overwrite." >&2
      exit 1
    fi

    restore="$(ls -d *"$1")"

    /bin/echo -n "Restoring file \"$1\" ... "
    $move "$restore" "$dest/$1"
    echo "Done."
  fi

  exit 0
```

Listing 2-7: The `unrm` shell script for restoring backed-up files

How It Works

The first chunk of code at ❶, the `if [$# -eq 0]` conditional block, executes if no arguments are specified, displaying the contents of the deleted files archive. However, there's a catch: we don't want to show the user the timestamp data we added to the filenames since that's only for the script's internal use. It would just clutter up the output. In order to display this data in a more attractive format, the `sed` statement at ❷ deletes the first five occurrences of *digit digit dot* in the `ls` output.

The user can specify the name of the file or directory to recover as an argument. The next step at ❸ is to ascertain how many matches there are for the name provided.

The unusual use of nested double quotes in this line (around `$1`) is to ensure `ls` matches filenames with embedded spaces, while the `*` wildcard expands the match to include any preceding timestamp. The `2> /dev/null` sequence is used to discard any error resulting from the command instead of showing it to the user. The errors being discarded will most likely be *No such file or directory*, when the specified filename isn't found.

If there are multiple matches for the given file or directory name, then the most complex part of this script, the `if [ $matches -gt 1 ]` block at ❹, is executed and displays all the results. Using the `-t` flag for the `ls` command in the main `for` loop causes the archive files to be presented from newest to oldest, and at ❺, a succinct call to the `awk` command translates the timestamp portion of the filename into a deletion date and time in parentheses. In the size calculation at ❼, the inclusion of the `-k` flag to `ls` forces the file sizes to be represented in kilobytes.

Rather than displaying the size of matching directory entries, the script displays the number of files within each matching directory, which is a more helpful statistic. The number of entries within a directory is easy to calculate. At ❻, we just count the number of lines given by `ls` and strip any spaces out of the `wc` output.

Once the user specifies one of the possible matching files or directories, the exact file is identified at ❽. This statement contains a slightly different use of `sed`. Specifying the `-n` flag with a line number (`${desired}`) followed by the `p` (print) command is a very fast way to extract only the specified line from the input stream. Want to see only line 37? The command `sed -n 37p` does just that.

Then there's a test at ❾ to ensure that `unrm` isn't going to step on an existing copy of the file, and the file or directory is restored with a call to `/bin/mv`. Once that's finished, the user is given a chance to remove the additional (probably superfluous) copies of the file ❿, and the script is done.

Note that using `ls` with `*"$1"` matches any filenames ending with the value in `$1`, so the list of multiple "matching files" may contain more than just the file the user wants to restore. For instance, if the deleted files directory contains the files *11.txt* and *111.txt*, running `unrm 11.txt` would signal that it found multiple matches and return listings for both *11.txt* and *111.txt*. While that might be okay, once the user chooses to restore the correct file (*11.txt*), accepting the prompt to delete additional copies of the file would also remove *111.txt*. Therefore, defaulting to delete under those circumstances might not be optimal. However, this could be easily overcome by using the `??.??.??.??.??."$1"` pattern instead, if you kept the same timestamp format for `newrm` as shown in Script #15 on page 55.

Running the Script

There are two ways to run this script. Without any arguments, the script will show a listing of all files and directories in the user's deleted files archive.

When given a filename as its argument, the script will try to restore that file or directory (if there's only one match), or it will show a list of candidates for restoration and allow the user to specify which version of the deleted file or directory to restore.

The Results

Without any arguments specified, the script shows what's in the deleted files archive as Listing 2-8 shows.

```
$ unrm
Contents of your deleted files archive (sorted by date):
  detritus                 this is a test
  detritus                 garbage
```

Listing 2-8: Running the `unrm` shell script with no arguments lists the current files available to restore

When a filename is specified, the script displays more information about the file if there are multiple files with that name, as shown in Listing 2-9.

```
$ unrm detritus
More than one file or directory match in the archive:
 1)   detritus (size = 7688Kb, deleted = 11/29 at 10:00:12)
 2)   detritus  (size = 4Kb, deleted = 11/29 at 09:59:51)

Which version of detritus should I restore ('0' to quit)? [1] : 0
unrm: Restore canceled by user.
```

Listing 2-9: Running the `unrm` shell script with a single argument attempts to restore the file

Hacking the Script

If you use this script, be aware that without any controls or limits, the files and directories in the deleted files archive will grow without bound. To avoid this, invoke `find` from within a `cron` job to prune the deleted files archive, using the `-mtime` flag to identify those files that have been sitting untouched for weeks. A 14-day archive is probably quite sufficient for most users and will keep the archival script from consuming too much disk space.

While we're at it, there are some improvements that could make this script more user friendly. Think about adding starting flags like `-l` to `restore latest` and `-D` to `delete 00additional copies of the file`. Which flags would you add, and how would they streamline processing?

#17 Logging File Removals

Instead of archiving deleted files, you may just want to keep track of what deletions are happening on your system. In Listing 2-10, file deletions with the `rm` command will be logged in a separate file without notifying the user.

This can be accomplished by using the script as a wrapper. The basic idea of wrappers is that they live between an actual Unix command and the user, offering the user useful functionality that's not available with the original command alone.

NOTE *Wrappers are such a powerful concept that you'll see them show up time and again as you go through this book.*

The Code

```
#!/bin/bash
# logrm--Logs all file deletion requests unless the -s flag is used

removelog="/var/log/remove.log"

❶ if [ $# -eq 0 ] ; then
    echo "Usage: $0 [-s] list of files or directories" >&2
    exit 1
  fi

❷ if [ "$1" = "-s" ] ; then
    # Silent operation requested ... don't log.
    shift
  else
❸   echo "$(date): ${USER}: $@" >> $removelog
  fi

❹ /bin/rm "$@"

  exit 0
```

Listing 2-10: The `logrm` shell script

How It Works

The first section ❶ tests the user input, generating a simple file listing if no arguments are given. Then at ❷, the script tests whether argument `1` is `-s`; if so, it skips logging the removal request. Finally, the timestamp, user, and command are added to the *$removelog* file ❸, and the user command is silently passed over to the real */bin/rm* program ❹.

Running the Script

Rather than giving this script a name like `logrm`, a typical way to install a wrapper program is to rename the underlying command it's intending to wrap and then install the wrapper using the original command's old name. If you choose this route, however, make sure that the wrapper invokes the newly renamed program, not itself! For example, if you rename */bin/rm* to */bin/rm.old*, and name this script */bin/rm*, then the last few lines of the script will need to be changed so that it invokes */bin/rm.old* instead of itself.

Alternatively, you can use an alias to replace standard `rm` calls with this command:

```
alias rm=logrm
```

In either case, you will need write and execute access to */var/log*, which might not be the default configuration on your particular system.

The Results

Let's create a few files, delete them, and then examine the remove log, as shown in Listing 2-11.

```
$ touch unused.file ciao.c /tmp/junkit
$ logrm unused.file /tmp/junkit
$ logrm ciao.c
$ cat /var/log/remove.log
Thu Apr  6 11:32:05 MDT 2017: susan: /tmp/central.log
Fri Apr  7 14:25:11 MDT 2017: taylor: unused.file /tmp/junkit
Fri Apr  7 14:25:14 MDT 2017: taylor: ciao.c
```

Listing 2-11: Testing the `logrm` shell script

Aha! Notice that on Thursday, user Susan deleted the file */tmp/central.log*.

Hacking the Script

There's a potential log file ownership permission problem here. Either the *remove.log* file is writable by all, in which case a user could clear its contents out with a command like `cat /dev/null > /var/log/remove.log`, or it isn't writable by all, in which case the script can't log the events. You could use a `setuid` permission—with the script running as root—so that the script runs with the same permissions as the log file. However, there are two problems with this approach. First, it's a really bad idea! Never run shell scripts under `setuid`! By using `setuid` to run a command as a specific user, no matter who is executing the command, you are potentially introducing security weaknesses to your system. Second, you could get into a situation where the users have permission to delete their files but the script doesn't, and because the effective uid set with `setuid` would be inherited by the `rm` command itself, things would break. Great confusion would ensue when users couldn't even remove their own files!

If you have an ext2, ext3, or ext4 filesystem (as is usually the case with Linux), a different solution is to use the `chattr` command to set a specific append-only file permission on the log file and then leave it writable to all without any danger. Yet another solution is to write the log messages to `syslog`, using the helpful `logger` command. Logging the `rm` commands with `logger` is straightforward, as shown here:

```
logger -t logrm "${USER:-LOGNAME}: $*"
```

This adds an entry to the syslog data stream, which is untouchable by regular users and is tagged with logrm, the username, and the command specified.

NOTE *If you opt to use `logger`, you'll want to check `syslogd(8)` to ensure that your configuration doesn't discard `user.notice` priority log events. It's almost always specified in the* /etc/syslogd.conf *file.*

#18 Displaying the Contents of Directories

One aspect of the `ls` command has always seemed pointless: when a directory is listed, `ls` either lists the directory's contents file by file or shows the number of 1,024-byte blocks required for the directory data. A typical entry in an `ls -l` output might be something like this:

```
drwxrwxr-x    2 taylor   taylor       4096 Oct 28 19:07 bin
```

But that's not very useful! What we really want to know is how many files are in the directory. That's what the script in Listing 2-12 does. It generates a nice multicolumn listing of files and directories, showing files with their sizes and directories with the number of files they contain.

The Code

```
#!/bin/bash

# formatdir--Outputs a directory listing in a friendly and useful format

# Note that you need to ensure "scriptbc" (Script #9) is in your current path
#   because it's invoked within the script more than once.

scriptbc=$(which scriptbc)


# Function to format sizes in KB to KB, MB, or GB for more readable output
❶ readablesize()
{

  if [ $1 -ge 1048576 ] ; then
    echo "$($scriptbc -p 2 $1 / 1048576)GB"
  elif [ $1 -ge 1024 ] ; then
    echo "$($scriptbc -p 2 $1 / 1024)MB"
  else
    echo "${1}KB"
  fi
}

#################
## MAIN CODE
```

```
if [ $# -gt 1 ] ; then
  echo "Usage: $0 [dirname]" >&2
  exit 1
❷ elif [ $# -eq 1 ] ; then   # Specified a directory other than the current one?
  cd "$@"                   # Then let's change to that one.
  if [ $? -ne 0 ] ; then    # Or quit if the directory doesn't exist.
    exit 1
  fi
fi

for file in *
do
  if [ -d "$file" ] ; then
❸   size=$(ls "$file" | wc -l | sed 's/[^[:digit:]]//g')
    if [ $size -eq 1 ] ; then
      echo "$file ($size entry)|"
    else
      echo "$file ($size entries)|"
    fi
  else
    size="$(ls -sk "$file" | awk '{print $1}')"
❹   echo "$file ($(readablesize $size))|"
  fi
done | \
❺ sed 's/ /^^^/g'  | \
  xargs -n 2     | \
  sed 's/\^\^\^/ /g' | \
❻ awk -F\| '{ printf "%-39s %-39s\n", $1, $2 }'

exit 0
```

Listing 2-12: The `formatdir` shell script for more readable directory listings

How It Works

One of the most interesting parts of this script is the `readablesize` function ❶, which accepts numbers in kilobytes and outputs their value in either kilobytes, megabytes, or gigabytes, depending on which unit is most appropriate. Instead of having the size of a very large file shown as 2,083,364KB, for example, this function will instead show a size of 2.08GB. Note that `readablesize` is called with the `$( )` notation ❹:

```
echo "$file ($(readablesize $size))|"
```

Since subshells automatically inherit any functions defined in the running shell, the subshell created by the `$()` sequence has access to the `readablesize` function. Handy.

Near the top of the script at ❷, there is also a shortcut that allows users to specify a directory other than the current directory and then changes the current working directory of the running shell script to the desired location, simply by using `cd`.

The main logic of this script involves organizing its output into two neat, aligned columns. One issue to deal with is that you can't simply replace spaces with line breaks in the output stream, because files and directories may have spaces within their names. To get around this problem, the script at ❺ first replaces each space with a sequence of three carets (^^^). Then it uses the xargs command to merge paired lines so that every group of two lines becomes one line separated by a real, expected space. Finally, at ❻ it uses the awk command to output columns in the proper alignment.

Notice how the number of (nonhidden) entries in a directory is easily calculated at ❸ with a quick call to wc and a sed invocation to clean up the output:

```
size=$(ls "$file" | wc -l | sed 's/[^[:digit:]]//g')
```

Running the Script

For a listing of the current directory, invoke the command without arguments, as Listing 2-13 shows. For information about the contents of a different directory, specify a directory name as the sole command line argument.

The Results

```
$ formatdir ~
Applications (0 entries)            Classes (4KB)
DEMO (5 entries)                    Desktop (8 entries)
Documents (38 entries)              Incomplete (9 entries)
IntermediateHTML (3 entries)        Library (38 entries)
Movies (1 entry)                    Music (1 entry)
NetInfo (9 entries)                 Pictures (38 entries)
Public (1 entry)                    RedHat 7.2 (2.08GB)
Shared (4 entries)                  Synchronize! Volume ID (4KB)
X Desktop (4KB)                     automatic-updates.txt (4KB)
bin (31 entries)                    cal-liability.tar.gz (104KB)
cbhma.tar.gz (376KB)                errata (2 entries)
fire aliases (4KB)                  games (3 entries)
junk (4KB)                          leftside navbar (39 entries)
mail (2 entries)                    perinatal.org (0 entries)
scripts.old (46 entries)            test.sh (4KB)
testfeatures.sh (4KB)               topcheck (3 entries)
tweakmktargs.c (4KB)                websites.tar.gz (18.85MB)
```

Listing 2-13: Testing the formatdir shell script

Hacking the Script

An issue worth considering is whether you happen to have a user who likes to use sequences of three carets in filenames. This naming convention is pretty unlikely—a 116,696-file Linux install that we spot-tested didn't have

even a single caret within any of its filenames—but if it did occur, you'd get some confusing output. If you're concerned, you could address this potential pitfall by translating spaces into another sequence of characters that's even less likely to occur in user filenames. Four carets? Five?

#19 Locating Files by Filename

One command that's quite useful on Linux systems, but isn't always present on other Unix flavors, is `locate`, which searches a prebuilt database of filenames for a user-specified regular expression. Ever want to quickly find the location of the master *.cshrc* file? Here's how that's done with `locate`:

```
$ locate .cshrc
/.Trashes/501/Previous Systems/private/etc/csh.cshrc
/OS9 Snapshot/Staging Archive/:home/taylor/.cshrc
/private/etc/csh.cshrc
/Users/taylor/.cshrc
/Volumes/110GB/WEBSITES/staging.intuitive.com/home/mdella/.cshrc
```

You can see that the master *.cshrc* file is in the */private/etc* directory on this OS X system. The version of `locate` we're going to build sees every file on the disk when building its internal file index, whether the file is in the trash queue or on a separate volume or even if it's a hidden dotfile. This is both an advantage and a disadvantage, as we will discuss shortly.

The Code

This method of finding files is simple to implement and comes in two scripts. The first (shown in Listing 2-14) builds a database of all filenames by invoking `find`, and the second (shown in Listing 2-15) is a simple `grep` of the new database.

```
#!/bin/bash

# mklocatedb--Builds the locate database using find. User must be root
#   to run this script.

locatedb="/var/locate.db"

❶ if [ "$(whoami)" != "root" ] ; then
  echo "Must be root to run this command." >&2
  exit 1
fi

find / -print > $locatedb

exit 0
```

Listing 2-14: The `mklocatedb` shell script

The second script is even shorter.

```
#!/bin/sh

# locate--Searches the locate database for the specified pattern

locatedb="/var/locate.db"

exec grep -i "$@" $locatedb
```

Listing 2-15: The `locate` shell script

How It Works

The `mklocatedb` script must be run as the root user to ensure that it can see all the files in the entire system, so this is checked at ❶ with a call to `whoami`. Running any script as root, however, is a security problem because if a directory is closed to a specific user's access, the `locate` database shouldn't store any information about the directory or its contents. This issue will be addressed in Chapter 5 with a new, more secure `locate` script that takes privacy and security into account (see Script #39 on page 127). For now, however, this script exactly emulates the behavior of the `locate` command in standard Linux, OS X, and other distributions.

Don't be surprised if `mklocatedb` takes a few minutes or longer to run; it's traversing the entire filesystem, which can take a while on even a medium-sized system. The results can be quite large, too. On one OS X system we tested, the *locate.db* file had over 1.5 million entries and ate up 1874.5MB of disk space.

Once the database is built, the `locate` script itself is a breeze to write; it's just a call to the `grep` command with whatever arguments are specified by the user.

Running the Script

To run the `locate` script, it's first necessary to run `mklocatedb`. Once that's done, `locate` invocations will almost instantly find all matching files on the system for any pattern specified.

The Results

The `mklocatedb` script has no arguments or output, as Listing 2-16 shows.

```
$ sudo mklocatedb
Password:
...
Much time passes
...
$
```

Listing 2-16: Running the `mklocatedb` shell script as root with the sudo command

We can check the size of the database with a quick `ls`, as shown here:

```
$ ls -l /var/locate.db
-rw-r--r--  1 root  wheel  174088165 Mar 26 10:02 /var/locate.db
```

Now we're ready to start finding files on the system using `locate`:

```
$ locate -i solitaire
/Users/taylor/Documents/AskDaveTaylor image folders/0-blog-pics/vista-search-
solitaire.png
/Users/taylor/Documents/AskDaveTaylor image folders/8-blog-pics/windows-play-
solitaire-1.png
/usr/share/emacs/22.1/lisp/play/solitaire.el.gz
/usr/share/emacs/22.1/lisp/play/solitaire.elc
/Volumes/MobileBackups/Backups.backupdb/Dave's MBP/2014-04-03-163622/BigHD/
Users/taylor/Documents/AskDaveTaylor image folders/0-blog-pics/vista-search-
solitaire.png
/Volumes/MobileBackups/Backups.backupdb/Dave's MBP/2014-04-03-163622/BigHD/
Users/taylor/Documents/AskDaveTaylor image folders/8-blog-pics/windows-play-
solitaire-3.png
```

This script also lets you ascertain other interesting statistics about your system, such as how many C source files you have, like this:

```
$ locate '\.c$' | wc -l
  1479
```

NOTE *Pay attention to the regular expression here. The* `grep` *command requires us to escape the dot (*`.`*) or it will match any single character. Also, the* `$` *denotes the end of the line or, in this case, the end of the filename.*

With a bit more work, we could feed each one of these C source files to the `wc` command and ascertain the total number of lines of C code on the system, but, um, that would be kinda daft, wouldn't it?

Hacking the Script

To keep the database reasonably up-to-date, it would be easy to schedule `mklocatedb` to run from `cron` in the wee hours of the night on a weekly basis—as most systems with built-in `locate` commands do—or even more frequently based on local usage patterns. As with any script executed by the root user, take care to ensure that the script itself isn't editable by non-root users.

One potential improvement to this script would be to have `locate` check its invocation and fail with a meaningful error message if no pattern is specified or if the *locate.db* file doesn't exist. As it's written now, the script will spit out a standard `grep` error instead, which isn't very useful. More importantly, as we discussed earlier, there's a significant security issue with letting users

have access to a listing of all filenames on the system, including those they wouldn't ordinarily be able to see. A security improvement to this script is addressed in Script #39 on page 127.

#20 Emulating Other Environments: MS-DOS

Though it's unlikely you'll ever need them, it's interesting and illustrative of some scripting concepts to create versions of classic MS-DOS commands, like `DIR`, as Unix-compatible shell scripts. Sure, we could just use a shell alias to map `DIR` to the Unix `ls` command, as in this example:

```
alias DIR=ls
```

But this mapping doesn't emulate the actual behavior of the command; it just helps forgetful people learn new command names. If you're hip to the ancient ways of computing, you'll remember that the `/W` option produces a wide listing format, for example. But if you specify `/W` to the `ls` command now, the program will just complain that the `/W` directory doesn't exist. Instead, the following `DIR` script in Listing 2-17 can be written so that it works with the forward-slash style of command flags.

The Code

```
#!/bin/bash
# DIR--Pretends we're the DIR command in DOS and displays the contents
#   of the specified file, accepting some of the standard DIR flags

function usage
{
cat << EOF >&2
  Usage: $0 [DOS flags] directory or directories
  Where:
   /D          sort by columns
   /H          show help for this shell script
   /N          show long listing format with filenames on right
   /OD         sort by oldest to newest
   /O-D        sort by newest to oldest
   /P          pause after each screenful of information
   /Q          show owner of the file
   /S          recursive listing
   /W          use wide listing format
EOF
  exit 1
}

#####################
### MAIN BLOCK

postcmd=""
flags=""
```

```
while [ $# -gt 0 ]
do
  case $1 in
    /D       ) flags="$flags -x"      ;;
    /H       ) usage                  ;;
❶   /[NQW]   ) flags="$flags -l"      ;;
    /OD      ) flags="$flags -rt"     ;;
    /O-D     ) flags="$flags -t"      ;;
    /P       ) postcmd="more"         ;;
    /S       ) flags="$flags -s"      ;;
           * ) # Unknown flag: probably a DIR specifier break;
               #   so let's get out of the while loop.
  esac
  shift        # Processed flag; let's see if there's another.
done

# Done processing flags; now the command itself:

if [ ! -z "$postcmd" ] ; then
  ls $flags "$@" | $postcmd
else
  ls $flags "$@"
fi

exit 0
```

Listing 2-17: The `DIR` *shell script for emulating the* `DIR` *DOS command on Unix*

How It Works

This script highlights the fact that shell case statement conditional tests are actually regular expression tests. You can see at ❶ that the DOS flags `/N`, `/Q`, and `/W` all map to the same `-l Unix` flag in the final invocation of the `ls` command and that all this is done in a simple regular expression `/[NQW]`.

Running the Script

Name this script `DIR` (and consider creating a system-wide shell alias of `dir=DIR` since DOS was case insensitive but Unix is most assuredly case sensitive). This way, whenever users type `DIR` at the command line with typical MS-DOS `DIR` flags, they'll get meaningful and useful output (shown in Listing 2-18) rather than a `command not found` error message.

The Results

```
$ DIR /OD /S ~/Desktop
total 48320
 7720 PERP - Google SEO.pdf            28816 Thumbs.db
    0 Traffic Data                     8 desktop.ini
    8 gofatherhood-com-crawlerrors.csv  80 change-lid-close-behavior-win7-1.png
   16 top-100-errors.txt               176 change-lid-close-behavior-win7-2.png
    0 $RECYCLE.BIN                     400 change-lid-close-behavior-win7-3.png
```

```
    0 Drive Sunshine                  264 change-lid-close-behavior-win7-4.png
   96 facebook-forcing-pay.jpg        32 change-lid-close-behavior-win7-5.png
10704 WCSS Source Files
```

Listing 2-18: Testing the `DIR` shell script to list files

This listing of the specified directory, sorted from oldest to newest, indicates file sizes (though directories always have a size of 0).

Hacking the Script

At this point, it might be tough to find someone who remembers the MS-DOS command line, but the basic concept is powerful and worth knowing. One improvement you could make, for example, would be to have the Unix or Linux equivalent command be displayed before being executed and then, after a certain number of system invocations, have the script show the translation but not actually invoke the command. The user would be forced to learn the new commands just to accomplish anything!

#21 Displaying Time in Different Time Zones

The most fundamental requirement for a working `date` command is that it displays the date and time in your time zone. But what if you have users across multiple time zones? Or, more likely, what if you have friends and colleagues in different locations, and you're always confused about what time it is in, say, Casablanca, Vatican City, or Sydney?

It turns out that the `date` command on most modern Unix flavors is built atop an amazing time zone database. Usually stored in the directory */usr/share/zoneinfo*, this database lists over 600 regions and details the appropriate time zone offset from UTC (Coordinated Universal Time, also often referred to as *GMT*, or *Greenwich Mean Time*) for each. The `date` command pays attention to the `TZ` time zone variable, which we can set to any region in the database, like so:

```
$ TZ="Africa/Casablanca" date
Fri Apr  7 16:31:01 WEST 2017
```

However, most system users aren't comfortable specifying temporary environment variable settings. Using a shell script, we can create a more user-friendly frontend to the time zone database.

The bulk of the script in Listing 2-19 involves digging around in the time zone database (which is typically stored across several files in the *zonedir* directory) and trying to find a file that matches a specified pattern. Once it finds a matching file, the script grabs the full time zone name (as with `TZ="Africa/Casablanca"` in this example) and invokes `date` with that as a subshell environment setting. The `date` command checks `TZ` to see what time zone it's in and has no idea if it's a one-off or the time zone you sit in most of the time.

The Code

```
#!/bin/bash

# timein--Shows the current time in the specified time zone or
#   geographic zone. Without any argument, this shows UTC/GMT.
#   Use the word "list" to see a list of known geographic regions.
#   Note that it's possible to match zone directories (regions),
#   but that only time zone files (cities) are valid specifications.

# Time zone database ref: http://www.twinsun.com/tz/tz-link.htm

zonedir="/usr/share/zoneinfo"

if [ ! -d $zonedir ] ; then
  echo "No time zone database at $zonedir." >&2
  exit 1
fi

if [ -d "$zonedir/posix" ] ; then
  zonedir=$zonedir/posix        # Modern Linux systems
fi

if [ $# -eq 0 ] ; then
  timezone="UTC"
  mixedzone="UTC"
❶ elif [ "$1" = "list" ] ; then
  ( echo "All known time zones and regions defined on this system:"
    cd $zonedir
    find -L * -type f -print | xargs -n 2 | \
      awk '{ printf "  %-38s %-38s\n", $1, $2 }'
  ) | more
  exit 0
else

  region="$(dirname $1)"
  zone="$(basename $1)"

  # Is the given time zone a direct match? If so, we're good to go.
  #   Otherwise we need to dig around a bit to find things. Start by
  #   just counting matches.

  matchcnt="$(find -L $zonedir -name $zone -type f -print |\
        wc -l | sed 's/[^[:digit:]]//g' )"

  # Check if at least one file matches.
  if [ "$matchcnt" -gt 0 ] ; then
    # But exit if more than one file matches.
    if [ $matchcnt -gt 1 ] ; then
      echo "\"$zone\" matches more than one possible time zone record." >&2
      echo "Please use 'list' to see all known regions and time zones." >&2
      exit 1
```

```
    fi
    match="$(find -L $zonedir -name $zone -type f -print)"
    mixedzone="$zone"
  else # Maybe we can find a matching time zone region, rather than a specific
       #   time zone.
    # First letter capitalized, rest of word lowercase for region + zone
    mixedregion="$(echo ${region%${region#?}} \
                 | tr '[[:lower:]]' '[[:upper:]]')\
                 $(echo ${region#?} | tr '[[:upper:]]' '[[:lower:]]')"
    mixedzone="$(echo ${zone%${zone#?}} | tr '[[:lower:]]' '[[:upper:]]') \
               $(echo ${zone#?} | tr '[[:upper:]]' '[[:lower:]]')"

    if [ "$mixedregion" != "." ] ; then
      # Only look for specified zone in specified region
      #   to let users specify unique matches when there's
      #   more than one possibility (e.g., "Atlantic").
      match="$(find -L $zonedir/$mixedregion -type f -name $mixedzone -print)"
    else
      match="$(find -L $zonedir -name $mixedzone -type f -print)"
    fi

    # If file exactly matched the specified pattern
    if [ -z "$match"  ] ; then
      # Check if the pattern was too ambiguous.
      if [ ! -z $(find -L $zonedir -name $mixedzone -type d -print) ] ; then
❷       echo "The region \"$1\" has more than one time zone. " >&2
      else  # Or if it just didn't produce any matches at all
        echo "Can't find an exact match for \"$1\". " >&2
      fi
      echo "Please use 'list' to see all known regions and time zones." >&2
      exit 1
    fi
  fi
❸ timezone="$match"
fi

nicetz=$(echo $timezone | sed "s|$zonedir/||g")    # Pretty up the output.

echo It\'s $(TZ=$timezone date '+%A, %B %e, %Y, at %l:%M %p') in $nicetz

exit 0
```

Listing 2-19: The `timein` shell script for reporting the time in a certain time zone

How It Works

This script exploits the ability of the `date` command to show the date and time for a specified time zone, regardless of your current environment settings. In fact, the entire script is all about identifying a valid time zone name so that the `date` command will work when invoked at the very end.

Most of the complexity of this script comes from trying to anticipate names of world regions entered by users that do not match the names of regions in the time zone database. The time zone database is laid out with *timezonename* and *region/locationname* columns, and the script tries to display

useful error messages for typical input problems, like a time zone that's not found because the user is specifying a country like *Brazil*, which has more than one time zone.

For example, although `TZ="Casablanca" date` would fail to find a matching region and display the UTC/GMT time instead, the city Casablanca does exist in the time zone database. The issue is that you have to use its proper region name of *Africa/Casablanca* in order for it to work, as was shown in the introduction to this script.

This script, on the other hand, can find Casablanca in the Africa directory on its own and identify the zone accurately. However, just specifying *Africa* wouldn't be specific enough, as the script knows there are subregions within Africa, so it produces an error message indicating that the information is insufficient to uniquely identify a specific time zone ❷. You can also just use `list` to list all time zones ❶ or an actual time zone name ❸ (for example, UTC or WET), which can be used as an argument to this script.

NOTE *An excellent reference to the time zone database can be found online at* http://www.twinsun.com/tz/tz-link.htm.

Running the Script

To check the time in a region or city, specify the region or city name as an argument to the `timein` command. If you know both the region and the city, you can also specify them as *`region/city`* (for example, `Pacific/Honolulu`). Without any arguments, `timein` shows UTC/GMT. Listing 2-20 shows the `timein` script running with a variety of time zones.

The Results

```
$  timein
It's Wednesday, April 5, 2017, at 4:00 PM in UTC
$ timein London
It's Wednesday, April 5, 2017, at 5:00 PM in Europe/London
$ timein Brazil
The region "Brazil" has more than one time zone. Please use 'list'
to see all known regions and time zones.
$ timein Pacific/Honolulu
It's Wednesday, April 5, 2017, at 6:00 AM in Pacific/Honolulu
$ timein WET
It's Wednesday, April 5, 2017, at 5:00 PM in WET
$ timein mycloset
Can't find an exact match for "mycloset". Please use 'list'
to see all known regions and time zones.
```

Listing 2-20: Testing the `timein` shell script with various time zones

Hacking the Script

Knowing the time in a specific time zone across the world is a great ability, especially for a systems admin who manages global networks. But sometimes, you really just want to know the *difference* in time between two time zones quickly. The `timein` script could be hacked to provide just this functionality. By creating a new script, perhaps called `tzdiff`, based on the `timein` script, you could accept two arguments instead of one.

Using both of the arguments, you could determine the current time in both time zones and then print the hour difference between the two. Keep in mind, though, that a two-hour difference between two time zones could be two hours *forward* or two hours *backward*, and this makes a big difference. Distinguishing between a two-hour difference going forward or backward is crucial in making this hack a useful script.