Chapter 12

Automating System Tasks

In This Chapter

- Understanding shell scripts
- System initialization
- System startup and shutdown
- Scheduling system tasks

You'd never get any work done if you typed every command that needs to be run on your Fedora or RHEL system when it starts. Likewise, you could work more efficiently if you grouped together sets of commands that you run all the time. Shell scripts can handle these tasks.

A *shell script* is a group of commands, functions, variables, or just about anything else you can use from a shell. These items are typed into a plain-text file. That file can then be run as a command. Fedora and RHEL use system initialization shell scripts during system startup to run commands needed to get things going. You can create your own shell scripts to automate the tasks you need to do regularly.

This chapter provides a rudimentary overview of the inner workings of shell scripts and how they can be used. You learn how shell scripts are responsible for the messages that scroll by on the system console during booting and how simple scripts can be harnessed to a scheduling facility (such as `cron` or `at`) to simplify administrative tasks.

You also learn to fine-tune your machine to start at the most appropriate run level and to run only services you need. With that understanding, you'll be able to personalize your computer and cut down on the amount of time you spend repetitively typing the same commands.

Understanding Shell Scripts

Have you ever had a task that you needed to do over and over that took a lot of typing on the command line? Do you ever think to yourself, "Wow, I wish there was just one command I could type to do all this of this"? Maybe a shell script is what you're after.

Shell scripts are the equivalent of batch files in MS-DOS, and can contain long lists of commands, complex flow control, arithmetic evaluations, user-defined variables, user-defined functions, and sophisticated condition testing. Shell scripts are capable of handling everything from simple one-line commands to something as complex as starting up your Fedora or RHEL system.

In fact, as you will read in this chapter, Fedora and RHEL do just that. They use shell scripts (`/etc/rc.d/rc.sysinit` and `/etc/rc`) to check and mount all your file systems, set up your consoles, configure your network, launch all your system services, and eventually provide you with your login screen. While there are nearly a dozen different shells available in Fedora and RHEL, the default shell is called `bash,` the Bourne-Again SHell.

Executing and debugging shell scripts

One of the primary advantages of shell scripts is that they can be opened in any text editor to see what they do. A big disadvantage is that large or complex shell scripts often execute more slowly than compiled programs. There are two basic ways to execute a shell script:

- The filename is used as an argument to the shell (as in `bash` *`myscript`*). In this method, the file does not need to be executable; it just contains a list of shell commands. The shell specified on the command line is used to interpret the commands in the script file. This is most common for quick, simple tasks.
- The shell script may also have the name of the interpreter placed in the first line of the script preceeded by `#!` (as in `#!/bin/bash`), and have its execute bit set (using `chmod +x`). You can then run your script just like any other program in your path simply by typing the name of the script on the command line.

CROSS-REFERENCE: See Chapter 4 for more details on `chmod` and read/write/execute permissions.

When scripts are executed in either manner, options for the program may be specified on the command line. Anything following the name of the script is referred to as a *command-line argument*.

As with writing any software, there is no substitute to clear and thoughtful design and lots of comments. The pound sign (#) prefaces comments and can take up an entire line or exist on the same line after script code. It’s best to implement more complex shell scripts in stages, making sure the logic is sound at each step before continuing. Here are a few good, concise tips to make sure things are working as expected during testing:

- Place an `echo` statement at the beginning of lines within the body of a loop. That way, rather than executing the code, you can see what will be executed without making any permanent changes.

- To achieve the same goal, you could place dummy `echo` statements throughout the code. If these lines get printed, you know the correct logic branch is being taken.
- You could use `set -x` near the beginning of the script to display each command that is executed or launch your scripts using `bash -x` *`myscript`*.
- Because useful scripts have a tendency to grow over time, keeping your code readable as you go along is extremely important. Do what you can to keep the logic of your code clean and easy to follow.

Understanding shell variables

Often within a shell script, you want to reuse certain items of information. During the course of processing the shell script, the name or number representing this information may change. To store information used by a shell script in such a way that it can be easily reused, you can set variables. Variable names within shell scripts are case-sensitive and can be defined in the following manner:

```
NAME=value
```

The first part of a variable is the variable name, and the second part is the value set for that name. Be sure that the `NAME` and `value` touch the equal sign, without any spaces. Variables can be assigned from constants, such as text or numbers. This is useful for initializing values or saving lots of typing for long constants. Here are examples where variables are set to a string of characters (`CITY`) and a numeric value (`PI`):

```
CITY="Springfield"
PI=3.14159265
```

Variables can contain the output of a command or command sequence. You can accomplish this by preceding the command with a dollar sign and open parenthesis, and following it with a closing parenthesis. For example, `MYDATE=$(date)` assigns the output from the `date` command to the `MYDATE` variable. Enclosing the command in backticks (`) can have the same effect.

NOTE: Keep in mind that characters such as dollar sign ($), backtick (`), asterisk (*), exclamation point (!), and others have special meaning to the shell, as you will see as you proceed through this chapter. To use those characters in an option to a command, and not have the shell use its special meaning, you need to precede that character with a backslash (\) or surround it in quotes. One place you will encounter this is in files created by Windows users that might include spaces, exclamation points, or other characters. In Linux, to properly interpret a file named **`my big! file!`**, you either need to surround it in double quotes or type: **`my\ big\! file\!`**

These are great ways to get information that can change from computer to computer or from day to day. The following example sets the output of the `uname -n` command to the `MACHINE` variable. Then I use parentheses to set NUM_FILES to the number of files in the current directory by piping (|) the output of the ls command to the word count command (wc -l).

```
MACHINE=`uname -n`
NUM_FILES=$(/bin/ls | wc -l)
```

Variables can also contain the value of other variables. This is useful when you have to preserve a value that will change so you can use it later in the script. Here BALANCE is set to the value of the CurBalance variable.

```
BALANCE="$CurBalance"
```

NOTE: When assigning variables, use only the variable name (for example, BALANCE). When referenced, meaning you want the *value* of the variable, precede it with a dollar sign (as in $CurBalance). The result of the latter is that you get the value of the variable, and not the variable name itself.

Special shell variables

There are special variables that the shell assigns for you. The most commonly used variables are called the *positional parameters* or *command line arguments* and are referenced as $0, $1, $2, $3 . . . $*n*. $0 is special and is assigned the name used to invoke your script; the others are assigned the values of the parameters passed on the command line. For instance, if the shell script named myscript were called as:

```
myscript foo bar
```

the positional parameter $0 would be myscript, $1 would be foo, and $2 would be bar.

Another variable, $#, tells you how many parameters your script was given. In our example, $# would be 2. Another particularly useful special shell variable is $?, which receives the exit status of the last command executed. Typically, a value of zero means everything is okay, and anything other than zero indicates an error of some kind. For a complete list of special shell variables, refer to the bash man page.

Parameter expansion in bash

As mentioned earlier, if you want the value of a variable, you precede it with a $ (for example, $CITY). This is really just shorthand for the notation ${CITY}; curly braces are used when the value of the parameter needs to be placed next to other text without a space. Bash has special rules that allow you to expand the value of a variable in different ways. Going into all the rules is probably a little overkill for a quick introduction to shell scripts, but Table 12-1 presents some common constructs that you're likely to see in bash scripts you find on your Fedora or RHEL box.

Table 12-1: Examples of bash Parameter Expansion

Construction	*Meaning*
`${var:-value}`	If variable is unset or empty, expand this to *value*
`${var#pattern}`	Chop the shortest match for *pattern* from the front of *var*'s value
`${var##pattern}`	Chop the longest match for *pattern* from the front of *var*'s value
`${var%pattern}`	Chop the shortest match for *pattern* from the end of *var*'s value
`${var%%pattern}`	Chop the longest match for *pattern* from the end of *var*'s value

Try typing the following commands from a shell to test how parameter expansion works:

```
$ THIS="Example"
$ THIS=${THIS:-"Not Set"}
$ THAT=${THAT:-"Not Set"}
$ echo $THIS
Example
$ echo $THAT
Not Set
```

In the examples here, the `THIS` variable is initially set to the word `Example`. In the next two lines, the `THIS` and `THAT` variables are set to their current values or to `Not Set`, if they are not currently set. Notice that because I just set `THIS` to the string `Example`, when I echo the value of `THIS` it appears as `Example`. However, since `THAT` was not set, it appears as `Not Set`.

NOTE: For the rest of this section, I show how variables and commands may appear in a shell script. To try out any of those examples, however, you can simply type them into a shell as shown in the previous example.

In the following example, `MYFILENAME` is set to `/home/digby/myfile.txt`. Next, the `FILE` variable is set to `myfile.txt` and `DIR` is set to `/home/digby`. In the `NAME` variable, the file name is cut down to simply `myfile`, then in the `EXTENSION` variable the file extension is set to `txt`. (To try these out, you can type them at a shell prompt as in the previous example, then echo the value of each variable to see how it is set.)

```
MYFILENAME="/home/digby/myfile.txt"
FILE=${MYFILENAME##*/}            #FILE becomes "myfile.txt"
DIR=${MYFILENAME%/*}              #DIR becomes "/home/digby"
NAME=${FILE%.*}                   #NAME becomes "myfile"
EXTENSION=${FILE#*.}              #EXTENSION becomes "txt"
```

Performing arithmetic in shell scripts

Bash uses *untyped* variables, meaning it normally treats variables as strings or text, but can change them on the fly if you want it to. Unless you tell it otherwise with `declare`, your variables are just a bunch of letters to bash. But when you start trying to do arithmetic with them, bash will convert them to integers if it can. This makes it possible to do some fairly complex arithmetic in bash.

Integer arithmetic can be performed using the built-in `let` command or through the external `expr` or `bc` commands. After setting the variable `BIGNUM` value to `1024`, the three commands that follow would all store the value `64` in the `RESULT` variable. The last command gets a random number between 0 and 10 and echoes the results back to you.

```
BIGNUM=1024
let RESULT=$BIGNUM/16
RESULT=`expr $BIGNUM / 16`
RESULT=`echo "$BIGNUM / 16" | bc -l`
let foo=$RANDOM%10; echo $foo
```

NOTE: While most elements of shell scripts are relatively freeform (where whitespace, such as spaces or tabs, is insignificant), both `let` and `expr` are particular about spacing. The `let` command insists on no spaces between each operand and the mathematical operator, whereas the syntax of the `expr` command requires whitespace between each operand and its operator. In contrast to those, `bc` isn't picky about spaces, but can be trickier to use because it does floating-point arithmetic.

To see a complete list of the kinds of arithmetic you can perform using the `let` command, type **help let** at the bash prompt.

Using programming constructs in shell scripts

One of the features that make shell scripts so powerful is that their implementation of looping and conditional execution constructs is similar to those found in more complex scripting and programming languages. You can use several different types of loops, depending on your needs.

The "if…then" statements

The most commonly used programming construct is conditional execution, or the `if` statement. It is used to perform actions only under certain conditions. There are several variations, depending on whether you're testing one thing, or want to do one thing if a condition is true, but another thing if a condition is false, or if you want to test several things one after the other.

The first `if...then` example tests if `VARIABLE` is set to the number `1`. If it is, then the `echo` command is used to say that it is set to `1`. The `fi` then indicates that the `if` statement is complete and processing can continue.

```
VARIABLE=1
if [ $VARIABLE -eq 1 ] ; then
echo "The variable is 1"
fi
```

Instead of using `-eq`, you can use the equals sign (`=`), as shown in the following example. The `=` works best for comparing string values, while `-eq` is often better for comparing numbers. Using the `else` statement, different words can be echoed if the criterion of the `if` statement isn't met (`$STRING = "Friday"`). Keep in mind that it's good practice to put strings in double quotes.

```
STRING="Friday"
if [ $STRING = "Friday" ] ; then
echo "WhooHoo.  Friday."
else
echo "Will Friday ever get here?"
fi
```

You can also reverse tests with an exclamation mark (!). In the following example, if `STRING` is not `Monday`, then `"At least it's not Monday"` is echoed.

```
STRING="FRIDAY"
if ["$STRING" != "Monday" ] ; then
   echo "At least it's not Monday"
fi
```

In the following example, `elif` (which stands for "else if") is used to test for an additional condition (is `filename` a file or a directory).

```
filename="$HOME"

if [ -f "$filename" ] ; then
   echo "$filename is a regular file"
elif [ -d "$filename" ] ; then
   echo "$filename is a directory"
else
   echo "I have no idea what $filename is"
fi
```

As you can see from the preceding examples, the condition you are testing is placed between square brackets `[ ]`. When a test expression is evaluated, it will return either a value of 0, meaning that it is true, or a 1, meaning that it is false. Notice that the echo lines are indented. This is optional and done only to make the script more readable.

Table 12-2 lists the conditions that are testable and is quite a handy reference. (If you're in a hurry, you can type **help test** on the command line to get the same information.)

Table 12-2: Operators for Test Expressions

Operator	***What Is Being Tested?***
`-a` *`file`*	Does the file exist? (same as `-e`)
`-b` *`file`*	Is the file a special block device?
`-c` *`file`*	Is the file character special (for example, a character device)? Used to identify serial lines and terminal devices.
`-d` *`file`*	Is the file a directory?
`-e` *`file`*	Does the file exist? (same as `-a`)
`-f` *`file`*	Does the file exist, and is it a regular file (for example, not a directory, socket, pipe, link, or device file)?
`-g` *`file`*	Does the file have the set-group-id bit set?
`-h` *`file`*	Is the file a symbolic link? (same as `-L`)
`-k` *`file`*	Does the file have the sticky bit set?
`-L` *`file`*	Is the file a symbolic link?
`-n` *`string`*	Is the length of the string greater than 0 bytes?
`-O` *`file`*	Do you own the file?
`-p` *`file`*	Is the file a named pipe?
`-r` *`file`*	Is the file readable by you?
`-s` *`file`*	Does the file exist, and is it larger than 0 bytes?
`-S` *`file`*	Does the file exist, and is it a socket?
`-t` *`fd`*	Is the file descriptor connected to a terminal?
`-u` *`file`*	Does the file have the set-user-id bit set?
`-w` *`file`*	Is the file writable by you?
`-x` *`file`*	Is the file executable by you?
`-z` *`string`*	Is the length of the string 0 (zero) bytes?
`expr1 -a` *`expr2`*	Are both the first expression and the second expression true?
`expr1 -o` *`expr2`*	Is either of the two expressions true?
`file1 -nt` *`file2`*	Is the first file newer than the second file (using the modification timestamp)?
`file1 -ot` *`file2`*	Is the first file older than the second file (using the modification timestamp)?
`file1 -ef` *`file2`*	Are the two files associated by a link (a hard link or a symbolic link)?

Operator	What Is Being Tested?
var1 = *var2*	Is the first variable equal to the second variable?
var1 -eq *var2*	Is the first variable equal to the second variable?
var1 -ge *var2*	Is the first variable greater than or equal to the second variable?
var1 -gt *var2*	Is the first variable greater than the second variable?
var1 -le *var2*	Is the first variable less than or equal to the second variable?
var1 -lt *var2*	Is the first variable less than the second variable?
var1 != *var2*	Is the first variable not equal to the second variable?
var1 -ne *var2*	Is the first variable not equal to the second variable?

There is also a special shorthand method of performing tests that can be useful for simple *one-command* actions. In the following example, the two pipes (||) indicate that if the directory being tested for doesn't exist (-d dirname), then make the directory (mkdir $dirname).

```
# [ test] || {action}
# Perform simple single command {action} if test is false
dirname="/tmp/testdir"
[ -d "$dirname" ] || mkdir "$dirname"
```

Instead of pipes, you can use two ampersands to test if something is true. In the following example, a command is being tested to see if it includes at least three command-line arguments.

```
# [ test ] && {action}
# Perform simple single command {action} if test is true
[ $# -ge 3 ] && echo "There are at least 3 command line arguments."
```

The case command

Another frequently used construct is the case command. Similar to a switch statement in programming languages, this can take the place of several nested if statements. A general form of the case statement is as follows:

```
case "VAR" in
   Result1)
      { body };;
   Result2)
      { body };;
   *)
      { body } ;;
esac
```

One use for the case command might be to help with your backups. The following case statement tests for the first three letters of the current day (case `date +%a` in). Then, depending on the day, a particular backup directory (BACKUP) and tape drive (TAPE) is set.

```
# Our VAR doesn't have to be a variable,
# it can be the output of a command as well
# Perform action based on day of week
case `date +%a` in
   "Mon")
         BACKUP=/home/myproject/data0
         TAPE=/dev/rft0
# Note the use of the double semi-colon to end each option
         ;;
# Note the use of the "|" to mean "or"
   "Tue" | "Thu")
         BACKUP=/home/myproject/data1
         TAPE=/dev/rft1
         ;;
   "Wed" | "Fri")
         BACKUP=/home/myproject/data2
         TAPE=/dev/rft2
         ;;
# Don't do backups on the weekend.
   *)
         BACKUP="none"
         TAPE=/dev/null
         ;;
esac
```

The asterisk (`*`) is used as a catchall, similar to the `default` keyword in the C programming language. In this example, if none of the other entries are matched on the way down the loop, the asterisk is matched, and the value of `BACKUP` becomes `none`. Note the use of `esac`, or `case` spelled backwards, to end the case statement.

The "for...do" loop

Loops are used to perform actions over and over again until a condition is met or until all data has been processed. One of the most commonly used loops is the `for...do` loop. It iterates through a list of values, executing the body of the loop for each element in the list. The syntax and a few examples are presented here:

```
for VAR in LIST
do
    { body }
done
```

The `for` loop assigns the values in `LIST` to `VAR` one at a time. Then for each value, the body in braces between `do` and `done` is executed. `VAR` can be any variable name, and `LIST` can be composed of pretty much any list of values or anything that generates a list.

```
for NUMBER in 0 1 2 3 4 5 6 7 8 9
do
   echo The number is $NUMBER
done
```

```
for FILE in `/bin/ls`
do
   echo $FILE
done
```

You can also write it this way, which is somewhat cleaner.

```
for NAME in John Paul Ringo George ; do
   echo $NAME is my favorite Beatle
done
```

Each element in the `LIST` is separated from the next by whitespace. This can cause trouble if you're not careful because some commands, such as `ls -l`, output multiple fields per line, each separated by whitespace. The string `done` ends the `for` statement.

If you're a die-hard C programmer, bash allows you to use C syntax to control your loops:

```
LIMIT=10
# Double parentheses, and no $ on LIMIT even though it's a variable!
for ((a=1; a <= LIMIT ; a++)) ; do
  echo  "$a"
done
```

The "while...do" and "until...do" loops

Two other possible looping constructs are the `while...do` loop and the `until...do` loop. The structure of each is presented here:

```
while condition       until condition
do                    do
   { body }             { body }
done                  done
```

The `while` statement executes while the condition is true. The `until` statement executes until the condition is true, in other words, while the condition is false.

Here is an example of a `while` loop that will output the number `0123456789`:

```
N=0
while [ $N -lt 10 ] ; do
   echo -n $N
   let N=$N+1
done
```

Another way to output the number 0123456789 is to use an `until` loop as follows:

```
N=0
until [ $N -eq 10 ] ; do
   echo -n $N
   let N=$N+1
done
```

Some useful external programs

Bash is great and has lots of built-in commands, but it usually needs some help to do anything really useful. Some of the most common useful programs you'll see used are `grep`, `cut`, `tr`, `awk`, and `sed`. As with all the best UNIX tools, most of these programs are designed to work with standard input and standard output, so you can easily use them with pipes and shell scripts.

The general regular expression parser (grep)

The name *general regular expression parser* sounds intimidating, but `grep` is just a way to find patterns in files or text. Think of it as a useful search tool. Getting really good with regular expressions is quite a challenge, but many useful things can be accomplished with just the simplest forms.

For example, you can display a list of all regular user accounts by using `grep` to search for all lines that contain the text `/home` in the `/etc/passwd` file as follows:

```
grep /home /etc/passwd
```

Or you could find all environment variables that begin with `HO` using the following command:

```
env | grep ^HO
```

NOTE: The ^ above is the actual caret character, ^, not what you'll commonly see for a backspace, ^H. Type **^**, **H**, and **O** (the uppercase letter) to see what items start with the uppercase characters *HO*.

To find a list of options to use with the `grep` command, type **man grep**.

Remove sections of lines of text (cut)

The `cut` command can extract specific fields from a line of text or from files. It is very useful for parsing system configuration files into easy-to-digest chunks. You can specify the field separator you want to use and the fields you want, or you can break up a line based on bytes.

The following example lists all home directories of users on your system. Using an earlier example of the `grep` command, this line pipes a list of regular users from the `/etc/passwd` file, then displays the sixth field (`-f6`) as delimited by a colon (`-d':'`). The hyphen at the end tells `cut` to read from standard input (from the pipe).

```
grep /home /etc/passwd | cut -f6 -d':' -
```

Translate or delete characters (tr)

The `tr` command is a character-based translator that can be used to replace one character or set of characters with another or to remove a character from a line of text.

The following example translates all uppercase letters to lowercase letters and displays the words `"mixed upper and lower case"` as a result:

```
FOO="Mixed UPpEr aNd LoWeR cAsE"
echo $FOO | tr [A-Z] [a-z]
```

In the next example, the `tr` command is used on a list of filenames to rename any files in that list so that any tabs or spaces (as indicated by the `[:blank:]` option) contained in a filename are translated into underscores. Try running the following code in a test directory:

```
for file in * ; do
   f=`echo $file | tr [:blank:] [_]`
   ["$file" = "-d" ] || mv -i "$file" "$f"
done
```

The Stream Editor (sed)

The `sed` command is a simple scriptable editor, and as such can perform only simple edits, such as removing lines that have text matching a certain pattern, replacing one pattern of characters with another, and other simple edits. To get a better idea of how `sed` scripts work, there's no substitute for the online documentation, but here are some examples of common uses.

You can use the `sed` command to essentially do what I did earlier with the `grep` example: search the `/etc/passwd` file for the word `home`. Here the `sed` command searches the entire `/etc/passwd` file, searches for the word `home`, and prints any line containing the word `home`.

```
sed –n '/home/p' /etc/passwd
```

In this example, `sed` searches the file `somefile.txt` and replaces every instance of the string `Mac` with `Linux`. Notice that the letter `g` is needed at the end of the substitution command to cause every occurrence of Mac on each line to be changed to Linux. (Otherwise, only the first instance of Mac on each line is changed.) The output is then sent to the `fixed_file.txt` file. The output from sed goes to stdout, so this command redirects the output to a file for safekeeping.

```
sed 's/Mac/Linux/g' somefile.txt > fixed_file.txt
```

You can get the same result using a pipe:

```
cat somefile.txt | sed 's/Mac/Linux/g' > fixed_file.txt
```

By searching for a pattern and replacing it with a null pattern, you delete the original pattern. This example searches the contents of the `somefile.txt` file and replaces extra blank spaces at the end of each line (`s/ *$`) with nothing (`//`). Results go to the `fixed_file.txt` file.

```
cat somefile.txt | sed 's/ *$//' > fixed_file.txt
```

Trying some simple shell scripts

Sometimes the simplest of scripts can be the most useful. If you type the same sequence of commands repetitively, it makes sense to store those commands (once!) in a file. Here are a couple of simple, but useful, shell scripts.

A simple telephone list

This idea has been handed down from generation to generation of old UNIX hacks. It's really quite simple, but it employs several of the concepts just introduced.

```
#!/bin/bash
# (@)/ph
# A very simple telephone list
# Type "ph new name number" to add to the list, or
# just type "ph name" to get a phone number

PHONELIST=~/.phonelist.txt

# If no command line parameters ($#), there
# is a problem, so ask what they're talking about.
if [ $# -lt 1 ] ; then
   echo "Whose phone number did you want? "
   exit 1
fi

# Did you want to add a new phone number?
if [ $1 = "new" ] ; then
   shift
   echo $* >> $PHONELIST
   echo $* added to database
   exit 0
fi

# Nope. But does the file have anything in it yet?
# This might be our first time using it, after all.
if [ ! -s $PHONELIST ] ; then
   echo "No names in the phone list yet! "
   exit 1
else
   grep -i -q "$*" $PHONELIST    # Quietly search the file
   if [ $? -ne 0 ] ; then        # Did we find anything?
      echo "Sorry, that name was not found in the phone list"
      exit 1
   else
      grep -i "$*" $PHONELIST
   fi
fi
exit 0
```

So, if you created the file ph in your current directory, you could type the following from the shell to try out your ph script:

```
$ chmod 755 ph
$ ./ph new "Mary Jones" 608-555-1212
Mary Jones 608-555-1212 added to database
$ ./ph Mary
Mary Jones 608-555-1212
```

The chmod command makes the ph script executable. The ./ph command runs the ph command from the current directory with the new option. This adds Mary Jones as the name and 608-555-1212 as the phone number to the database ($HOME/.phone.txt). The next ph command searches the database for the name Mary and displays the phone entry for Mary. If the script works, add it to a directory in your PATH (such as $HOME/bin).

A simple backup script

Because nothing works forever and mistakes happen, backups are just a fact of life when dealing with computer data. This simple script backs up all the data in the home directories of all the users on your Fedora or RHEL system.

```
#!/bin/bash
# (@)/my_backup
# A very simple backup script
#

# Change the TAPE device to match your system.
# Check /var/log/messages to determine your tape device.
# You may also need to add scsi-tape support to your kernel.
TAPE=/dev/rft0

# Rewind the tape device $TAPE
mt $TAPE rew
# Get a list of home directories
HOMES=`grep /home /etc/passwd | cut –f6 –d': '`
# Backup the data in those directories
tar cvf $TAPE $HOMES
# Rewind and eject the tape.
mt $TAPE rewoffl
```

CROSS-REFERENCE: See Chapter 13 for details on backing up and restoring files and getting the mt command (part of the ftape-tools packages that must be installed separately).

System Initialization

When you turn on your computer, a lot happens even before Fedora or RHEL starts up. Here are the basic steps that occur each time you boot up your computer to run Fedora or RHEL:

1. **Boot hardware —** Based on information in the computer's read-only memory (referred to as the BIOS), your computer checks and starts up the hardware. Some of that information tells the computer which devices (floppy disk, CD, hard disk, and so on) to check to find the bootable operating system.
2. **Start boot loader —** After checking that no bootable operating system is ready to boot in your floppy, CD, or DVD drive, typically, the BIOS checks the master boot record on the primary hard disk to see what to load next. With Fedora or RHEL installed, the GRUB boot loader is started, allowing you to choose to boot Fedora, RHEL, or another installed operating system.
3. **Boot the kernel —** Assuming that you selected to boot Fedora or RHEL, the Linux kernel is loaded. That kernel mounts the basic file systems and transfers control to the init process. The rest of this section describes what happens after the kernel hands off control of system startup to the init process.

Starting init

In the boot process, the transfer from the kernel phase (the loading of the kernel, probing for devices, and loading drivers) to `init` is indicated by the following lines:

```
                Welcome to Fedora
                Press "I" to enter interactive startup.
```

The init program, part of the upstart RPM package, is now in control. The output from `ps` always lists `init` (known as "the father of all processes") as PID (process identifier) 1.

Prior to Fedora 9, a special script, `/etc/inittab`, directed the actions of the init program. Starting in Fedora 9, a new initialization system called upstart replaced the older SysV Unix-style init program.

With upstart, the `/etc/inittab` file controls only the default run level. Everything else gets run from a special script for each run level, for example, `/etc/event.d/rc5` for run level 5. Upstart is based on the concept of launching programs based on *events* rather than run levels. Upstart can restart services that terminate, and isn't as fragile as the older SysV init system.

Since so many Linux programs make assumptions about the init system, though, upstart operates under a compatibility mode. In addition, to understand upstart, you need to understand the older Linux and Unix run levels, described in the following section. Over a number of Fedora releases, upstart will gradually phase out older means to initialize Linux. For now, though, the following concepts still apply.

The inittab file

The following example shows the contents of the `/etc/inittab` file as it is delivered with Fedora and RHEL in versions prior to Fedora 9:

```
#
# inittab       This file describes how the INIT process should set up
#               the system in a certain run level.
      .
      .
      .
id:5:initdefault:

# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit

l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6

# Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now

# When our UPS tells us power has failed, assume we have a few minutes
# of power left. Schedule a shutdown for 2 minutes from now.
# This does, of course, assume you have powerd installed and your
# UPS connected and working correctly.
pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"

# If power was restored before the shutdown kicked in, cancel it.
pr:12345:powerokwait:/sbin/shutdown -c "Power Restored; Shutdown Cancelled"

# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6

# Run xdm in runlevel 5
x:5:once:/etc/X11/prefdm -nodaemon
```

The plain-text `inittab` file consists of several colon-separated fields in the format:

```
id:runlevels:action:command
```

The `id` field is a unique identifier, one to four alphanumeric characters in length, that represents a particular action to take during system startup. The `runlevels` field contains a list of run levels in which the command will be run. Common run levels are 0, 1, 2, 3, 4, 5, and 6 (s and S represent single-user mode, which is equivalent to 1). Run levels 7, 8, and 9 can also be used as the special run levels associated with the on-demand action (a, b, and c, which are equivalent to A, B, and C). The next field represents the type of action to be taken by `init` (valid actions and the results of those actions are listed in Table 12-3), and the last field is the actual command that is to be executed.

Table 12-3: Valid init Actions

Action	***How the Command Is Run***
`once`	The command is executed once when entering the specified run level.
`wait`	The same as `once`, but `init` waits for the command to finish before continuing with other `inittab` entries.
`respawn`	The process is monitored, and a new instance is started if the original process terminates.
`powerfail`	The command is executed on receiving a `SIGPWR` signal from software associated with a UPS unit. This assumes a few minutes of power are left so, by default, the system can shut down the system in two minutes.
`powerwait`	The same as `powerfail`, but `init` waits for the command to finish.
`powerokwait`	The command is executed on receiving a `SIGPWR` signal if the `/etc/powerstatus` file contains the word OK. This is generally accomplished by the UPS software and indicates that a normal power level has been restored.
`powerfailnow`	The local system may be able to detect that an external UPS is running out of power and is about to fail almost immediately.
`ondemand`	The command is executed when `init` is manually instructed to enter one of the special run levels a, b, or c (equivalent to A, B, and C, respectively). No change in run level actually takes place. The program is restarted if the original process terminates.
`sysinit`	The command is executed during the system boot phase; the `runlevels` field is ignored.
`boot`	The command is executed during the system boot phase, after all `sysinit` entries have been processed; the `runlevels` field is ignored.
`bootwait`	The same as `boot`, but `init` waits for the command to finish before continuing with other `inittab` entries; the `runlevels` field is also ignored.
`initdefault`	The run level to enter after completing the `boot` and `sysinit` actions.

Action	***How the Command Is Run***
`off`	Nothing happens (perhaps useful for testing and debugging).
`ctrlaltdel`	Traps the Ctrl+Alt+Del key sequence and is typically used to gracefully shut down the system.
`kbrequest`	Used to trap special key sequences, as interpreted by the keyboard handler.

Because the `inittab` file is a configuration file, not a sequential shell script, the order of lines is not significant. Lines beginning with a hash (#) character are comments and are not processed.

The first non-commented line in the preceding sample `inittab` file sets the default run level to 5. A default of 5 (which is the common run level for desktop systems) means that, following the completion of all commands associated with the `sysinit`, `boot`, and `bootwait` actions, run level 5 is entered (booting to a text-based login). The other common `initdefault` level is run level 3 (often used for servers that boot up in text mode and often have no GUI). Table 12-4 describes each of the run levels and helps you choose the run level that is best suited as the default in your environment.

Table 12-4: Possible Run Levels

Run Level	***What Happens in This Run Level***
0	All processes are terminated and the machine comes to an orderly halt. As the `inittab` comments point out, this is not a good choice for `initdefault` because as soon as the kernel, modules, and drivers are loaded, the machine halts.
1, s, S	This is single-user mode, frequently used for system maintenance and instances where it may be preferable to have few processes running and no services activated. In single-user mode, the network is nonexistent, the X server is not running, and it is possible that some file systems are not mounted.
2	Multiuser mode. Multiple user logins are allowed, all configured file systems are mounted, and all processes except X, the `at` daemon, the `xinetd` daemon, and NIS/NFS are started. If your machine doesn't have (or perhaps doesn't need) a permanent network connection, this is a good choice for `initdefault`.
3	Multiuser mode with network services. Run level 3 is the typical value for `initdefault` on a Fedora or RHEL server.
4	Run level 4 is available as a user-defined run level. It is nearly identical to runlevel 3 in a default Fedora or RHEL configuration.

Run Level	***What Happens in This Run Level***
5	Multiuser mode with network services and X. This run level starts the X server and presents a graphical login window, visually resembling any of the more expensive UNIX-based workstations. This is a common `initdefault` value for a Fedora or RHEL workstation or desktop system.
6	All processes are terminated and the machine is gracefully rebooted. Again, the comments in the `inittab` file mention that this is not a good choice for `initdefault`, perhaps even worse than run level 0. The effect is a possibly infinite cycle of booting, followed by rebooting.
7, 8, 9	Generally unused and undefined, these run levels have the potential to meet any needs not covered by the default options.
a, b, c, A, B, C	Used in conjunction with the `ondemand` action. These don't really specify a run level but can launch a program or daemon "on demand" if so instructed.

NOTE: If there is no `initdefault` specified in the `inittab` file, the boot sequence will be interrupted and you will be prompted to specify a default run level into which the machine will boot.

The next line in the `inittab` file instructs `init` to execute the `/etc/rc.d/rc.sysinit` script before entering the default run level. This script performs many initialization routines such as choosing a `keymap` file, checking and mounting `root` and `proc` file systems, setting the clock and hostname, configuring swap space, cleaning up `temp` files, and loading modules.

The seven following lines control the commands executed within each major run level. In each, the `/etc/rc.d/rc` script is called, using the desired run level as an argument. In turn, it descends into the appropriate directory tree (for example, the `/etc/rc3.d` directory is entered for run level 3).

The `ctrlaltdel` action in the `inittab` file tells `init` to perform exactly what PC users would expect if the Ctrl, Alt, and Delete keys were pressed simultaneously. The system reboots itself in an orderly fashion (a switch to run level 6) after a three-second delay.

The next two lines (with their comments) deal with graceful shutdowns if you have an uninterruptible power supply (UPS) and software installed. The first line initiates a halt (a switch to run level 0) two minutes after receiving a signal from the UPS indicating a power failure. The second line cancels the shutdown in the event that power is restored.

The six `getty` lines start up virtual consoles to allow logins. These processes are always running in any of the multiuser run levels. When someone connected to a virtual console logs out, that `getty` process dies, and then `respawn` action tells `init` to start a new `getty` process. You can switch between virtual consoles by pressing Ctrl+Alt+F1, Ctrl+Alt+F2, and so on.

The last line indicates that as long as the system is in run level 5, the "preferred display manager" (xdm, gnome, KDE, and so on) will be running. This presents a graphical login prompt rather than the usual text-based login, and eliminates the need to run `startx` to start the GUI.

Starting with Fedora 10, the `inittab` file contains a simple one-line setting to define the default run level:

```
id:5:initdefault
```

This file specifies run level 5 as the default.

System Startup and Shutdown

During system startup, a series of scripts are run to start the services that you need. These include scripts to start network interfaces, mount directories, and monitor your system. Most of these scripts are run from subdirectories of `/etc/rc.d`. The program that starts most of these services up when you boot and stops them when you shut down is the `/etc/rc.d/rc` script. The following sections describe run-level scripts and what you can do with them.

Starting run-level scripts

As previously mentioned, the `/etc/rc.d/rc` script is integral to the concept of run levels. Any change of run level causes the script to be executed, with the new run level as an argument. Here's a quick run-down of what the `/etc/rc.d/rc` script does:

- **Checks that run-level scripts are correct** — The `rc` script checks to find each run-level script that exists and excludes those that represent backup scripts left by `rpm` updates.
- **Determines current and previous run levels** — Determines the current and previous run levels to know which run-level scripts to stop (previous level) and start (current level).
- **Decides whether to enter interactive startup** — If the `confirm` option is passed to the boot loader at boot time, all server processes must be confirmed at the system console before starting.
- **Kills and starts run-level scripts** — Stops run-level scripts from the previous level, then starts run-level scripts from the current level.

In Fedora and RHEL, most of the services that are provided to users and computers on the network are started from run-level scripts.

Understanding run-level scripts

A software package that has a service to start at boot time (or when the system changes run levels) can add a script to the `/etc/init.d` directory. That script can then be linked to an appropriate run-level directory and run with either the `start` or `stop` option (to start or stop the service).

Table 12-5 lists many of the typical run-level scripts that are found in `/etc/init.d` and explains their function. Depending on the Fedora or RHEL software packages you installed on your system, you may have dozens more run-level scripts than you see here. (Later, I describe how these files are linked into particular run-level directories.)

Each script representing a service that you want to start or stop is linked to a file in each of the run-level directories. For each run level, a script beginning with `K` stops the service, whereas a script beginning with `S` starts the service.

The two digits following the `K` or `S` in the filename provide a mechanism to select the priority in which the programs are run. For example, `S12syslog` is run before `S90crond`. However, the file `S110my_daemon` is run before `S85gpm`, even though you can readily see that 85 is less than 110. This is because the ASCII collating sequence orders the files, which simply means that one positional character is compared to another. Therefore, a script beginning with the characters `S110` is executed between `S10network` and `S15netfs` in run level 3.

All of the programs within the `/etc/rcX.d` directories (where *X* is replaced by a run-level number) are symbolic links, usually to a file in `/etc/init.d`. The `/etc/rcX.d` directories include the following:

- `/etc/rc0.d`: Run level 0 directory
- `/etc/rc1.d`: Run level 1 directory
- `/etc/rc2.d`: Run level 2 directory
- `/etc/rc3.d`: Run level 3 directory
- `/etc/rc4.d`: Run level 4 directory
- `/etc/rc5.d`: Run level 5 directory
- `/etc/rc6.d`: Run level 6 directory

In this manner, `/etc/rc0.d/K05atd`, `/etc/rc1.d/K05atd`, `/etc/rc2.d/K05atd`, `/etc/rc3.d/S95atd`, `/etc/rc4.d/S95atd`, `/etc/rc5.d/S95atd`, and `/etc/rc6.d/K05atd` are all symbolic links to `/etc/init.d/atd`. Using this simple, consistent mechanism, you can customize which programs are started at boot time.

Table 12-5: Run-Level Scripts Contained in /etc/init.d

Run-Level Scripts	***What Does It Do?***
`acpid`	Controls the Advanced Configuration and Power Interface daemon, which monitors events in the kernel and reports them to user level.
`anacron`	Runs cron jobs that were not run at their intended times due to the system being down.
`apmd`	Controls the Advanced Power Management daemon, which monitors battery status, and which can safely suspend or shut down all or part of a machine that supports it.
`atd`	Starts or stops the `at` daemon to receive, queue, and run jobs submitted via the `at` or `batch` commands. (The `anacron` run-level script runs `at` and `batch` jobs that were not run because the computer was down.)

Run-Level Scripts	***What Does It Do?***
`autofs`	Starts and stops the `automount` daemon, for automatically mounting file systems (so, for example, a CD can be automatically mounted when it is inserted).
`bluetooth`	Starts services such as authentication, discovery, and human interface devices for communicating with Bluetooth devices.
`ConsoleKit`	Maintains a list of user sessions.
`crond`	Starts or stops the `cron` daemon to periodically run routine commands.
`cups`	Controls the printer daemon (`cupsd`) that handles spooling printing requests.
`dhcpd`	Starts or stops the `dhcpd` daemon, which automatically assigns IP addresses to computers on a LAN.
`dovecot`	Starts the dovecot IMAP server, which allows e-mail clients to request and view their mail messages from the mail server.
`firstboot`	Checks to see if firstboot needs to be run and, if so, runs it. This is typically done after Fedora or RHEL is first installed.
`gpm`	Controls the `gpm` daemon, which allows the mouse to interact with console- and text-based applications.
`haldaemon`	Starts hald daemon to discover and set up hardware. Used to mount removable media, manage power, or auto-play multimedia.
`halt`	Terminates all processes, writes out accounting records, removes swap space, unmounts all file systems, and either shuts down or reboots the machine (depending on how the command was called).
`hplip`	Starts the HP Linux Imaging and Printing (HPLIP) service for running HP multi-function peripherals.
`httpd`	Starts the `httpd` daemon, which allows your computer to act as an HTTP server (that is, to serve Web pages).
`iptables`	Starts the `iptables` firewall daemon, which manages any `iptables`-style firewall rules set up for your computer.
`keytable`	Loads the predefined keyboard map.
`killall`	Shuts down any subsystems that may still be running prior to a shutdown or reboot.
`ldap`	Starts the Lightweight Directory Access Protocol daemon (`sldap`), which listens for LDAP requests from the network.
`messagebus`	Runs the dbus-daemon for broadcasting system message to interested applications.

Run-Level Scripts	***What Does It Do?***
`mysqld`	Runs the MySQL database daemon (`mysqld`) to listen for request to MySQL databases.
`named`	Starts and stops the BIND DNS server daemon (`named`) to listen for and resolve domain name system requests.
`netfs`	Mounts or unmounts network (NFS, SMB, and NCP) file systems.
`network`	Starts or stops all configured network interfaces and initializes the TCP/IP and IPX protocols.
`NetworkManager`	Switches automatically to the best available network connections.
`nfs`	Starts or stops the NFS-related daemons (`rpc.nfsd`, `rpc.mountd`, `rpc.statd`, and `rcp.rquotad`) and exports shared file systems.
`ntpd`	Runs the Network Time Protocol daemon (ntpd), which synchronizes system time with Internet standard time servers.
`openvpn`	Runs the OpenVPN virtual private network service.
`portmap`	Starts or stops the `portmap` daemon, which manages programs and protocols that utilize the Remote Procedure Call (RPC) mechanism.
`routed`	Starts or stops the `routed` daemon, which controls dynamic-routing table updates via the Router Information Protocol (RIP).
`rsyslog`	Starts or stops the `klogd` and r`syslogd` daemons that handle logging events from the kernel and other processes, respectively.
`rwhod`	Starts or stops the `rwhod` daemon, which enables others on the network to obtain a list of all currently logged-in users.
`sendmail`	Controls the `sendmail` daemon, which handles incoming and outgoing SMTP (Simple Mail Transport Protocol) mail messages.
`single`	Terminates running processes and enters run level 1 (single-user mode).
`smb`	Starts or stops the `smbd` and `nmbd` daemons for allowing access to Samba file and print services.
`snmpd`	Starts or stops the `snmpd` (Simple Network Management Protocol) daemon, which enables others to view machine-configuration information.
`spamassassin`	Starts and stops the `spamd` daemon to automate the process of checking e-mail messages for spam.
`squid`	Starts or stops the `squid` services, which enables proxy service to clients on your network.
`sshd`	Runs the secure shell daemon (`sshd`), which listens for requests from ssh clients for remote login or remote execution requests.

Run-Level Scripts	***What Does It Do?***
`vsftpd`	Runs the Very Secure FTP server (`vsftpd`) to provide FTP sessions to remote clients for downloading and uploading files.
`winbind`	Runs the winbind service for Samba file and print services.
`xfs`	Starts or stops `xfs`, the X Window font server daemon.
`xinetd`	Sets the machine's hostname, establishes network routes, and controls `xinetd`, the network services daemon that listens for incoming TCP/IP connections to the machine.
`yum`	Enables you to run automatic nightly updates of your software using the yum facility.
`xfs`	Regenerates font lists and starts and stops the X font server.

Understanding what startup scripts do

Despite all the complicated rc*X*s, Ss, and Ks, the form of each startup script is really quite simple. Because they are in plain text, you can just open one with a text editor to take a look at what it does. For the most part, a run-level script can be run with a `start` option, a `stop` option, and possibly a `restart` option. For example, the following lines are part of the contents of the `smb` script that defines what happens when the script is run with different options to start or stop the Samba file and print service:

```
#!/bin/sh
#
# chkconfig: - 91 35
# description: Starts and stops the Samba smbd daemon \
#              used to provide SMB network services.
        .
        .
        .
start() {
        KIND="SMB"
        echo -n $"Starting $KIND services: "
        daemon smbd $SMBDOPTIONS
        RETVAL=$?
        echo

        [ $RETVAL -eq 0 -a $RETVAL2 -eq 0 ] && touch /var/lock/subsys/smb || \
           RETVAL=1
        return $RETVAL
}

stop() {
        KIND="SMB"
        echo -n $"Shutting down $KIND services: "
        killproc smbd
        RETVAL=$?
        echo
        [ $RETVAL -eq 0 -a $RETVAL2 -eq 0 ] && rm -f /var/lock/subsys/smb
        return $RETVAL
}

restart() {
        stop
        start
}
      .
      .
       .
```

To illustrate the essence of what this script does, I skipped some of the beginning and end of the script (where it checked if the network was up and running and set some values). Here are the actions `smb` takes when it is run with `start` or `stop`:

- **start** — This part of the script starts the `smbd` server when the script is run with the start option.
- **stop** — When run with the `stop` option, the `/etc/init.d/smb` script stops the `smbd` server.

The `restart` option runs the script with a `stop` option followed by a `start` option. If you want to start the `smb` service yourself, type the following command (as root user):

```
# service smb start
Starting SMB services:                     [ OK ]
```

To stop the service, type the following command:

```
# service smb stop
Shutting down SMB services:                [ OK ]
```

The `smb` run-level script is different from other run-level scripts in that it supports several other options than `start` and `stop`. For example, this script has options (not shown in the example) that allow you to reload the `smb.conf` configuration file (`reload`) and check the status of the service (`status`).

Changing run-level script behavior

Modifying the startup behavior of any such script merely involves opening the file in a text editor.

For example, the `atd` daemon queues jobs submitted from the `at` and `batch` commands. Jobs submitted via `batch` are executed only if the system load is not above a particular value, which can be set with a command-line option to the `atd` command.

The default *limiting load factor* value of 0.8 is based on the assumption that a single-processor machine with less than 80 percent CPU utilization could handle the additional load of the batch job. However, if you were to add another CPU to your machine, 0.8 would only represent 40 percent of the computer's processing power. So you could safely raise that limit without impacting overall system performance.

You can change the limiting load factor from 0.8 to 1.6 to accommodate the increased processing capacity. To do this, simply modify the following line (in the `start` section) of the `/etc/init.d/atd` script:

```
daemon /usr/sbin/atd
```

Replace it with this line, using the `-l` argument to specify the new minimum system load value:

```
daemon /usr/sbin/atd -l 1.6
```

After saving the file and exiting the editor, you can reboot the machine or just run any of the following three commands to begin using the new batch threshold value:

```
# service atd reload
# service atd restart
# service atd stop ; service atd start
```

NOTE: Always make a copy of a run-level script before you change it. Also, keep track of changes you make to run-level scripts before you upgrade the packages they come from. You need to make those changes again after the upgrade.

If you are uncomfortable editing startup scripts and you simply want to add options to the daemon process run by the script, there may be a way of entering these changes without editing the startup script directly. Check the `/etc/sysconfig` directory and see if there is a file by the same name as the script you want to modify. If there is, that file probably provides values that you can set to pass options to the startup script. Sysconfig files exist for `apmd`, `arpwatch`, `dhcpd`, `ntpd`, `samba`, `squid`, and others.

Reorganizing or removing run-level scripts

There are several ways to deal with removing programs from the system startup directories, adding them to particular run levels, or changing when they are executed. From a Terminal window, you can use the `chkconfig` command. From a GUI, use the Service Configuration window.

CAUTION: You should never remove the run-level file from the `/etc/init.d` directory. Because no scripts are run from the `/etc/init.d` directory automatically, it is okay to keep them there. Scripts in `/etc/init.d` are only accessed as links from the `/etc/rcX.d` directories. Keep scripts in the `init.d` directory so you can add them later by re-linking them to the appropriate run-level directory.

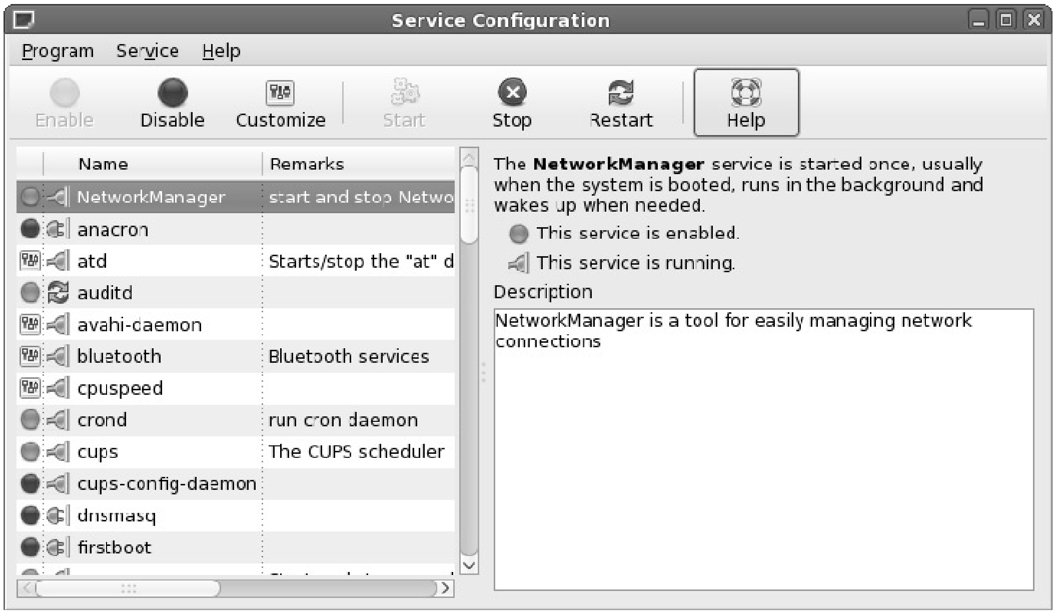
To reorganize or remove run-level scripts from the GUI, use the Service Configuration window. Either select System → Administration → Services or log in as root user and type the following command in a Terminal window:

```
# system-config-services &
```

Figure 12-1 shows an example of the Service Configuration window.

The Service Configuration window enables you to reconfigure services for run levels 2, 3, 4, and 5. Icons next to each service indicate whether the service is currently enabled (green) or disabled (red) for the current run level and whether or not the service is currently running. Select a service to see a description of that service. Here is what you can do from this window:

- **Enable** — With a service selected, click the Enable button to enable the service to start when you start your computer (run levels 2, 3, 4, and 5).
- **Disable** — With a service selected, click Disable to not have the service not start when you boot your computer (or otherwise enter run levels 2, 3, 4, or 5).
- **Customize** — With a service selected, click Customize and select the run levels at which you want the service to start.
- **Start** — Click a service on the list. Select Start to request the service to immediately start.



Name	Remarks
NetworkManager	start and stop Netwo
anacron	
atd	Starts/stop the "at" d
auditd	
avahi-daemon	
bluetooth	Bluetooth services
cpuspeed	
crond	run cron daemon
cups	The CUPS scheduler
cups-config-daemon	
dnsmasq	
firstboot	

The **NetworkManager** service is started once, usually when the system is booted, runs in the background and wakes up when needed.

This service is enabled.

This service is running.

Description

NetworkManager is a tool for easily managing network connections

Figure 12-1: Reorganize, add, and remove run-level scripts from the Service Configuration window.

Some administrators prefer text-based commands for managing run-level scripts and for managing other system services that start automatically. The `chkconfig` command can be used to list whether services that run-level scripts start, as well as services the `xinetd` daemon starts, are on or off. To see a list of all system services, with indications that they are on or off, type the following:

```
# chkconfig --list | less
```

You can then page through the list to see those services. If you want to view the status of an individual service, you can add the service at the end of the list option. For example, to see whether the nfs service starts in each run level, type the following:

```
# chkconfig --list nfs
nfs       0:off   1:off   2:off   3:on   4:on   5:on   6:off
```

This example shows that the nfs service is set to be on for run levels 3, 4, and 5, but that it is set to off for run levels 0, 1, 2, and 6.

Another tool that can be run from the shell to change which services start and do not start at various levels is the `ntsysv` command. Type the following as root user from the shell:

```
# ntsysv
```

A screen appears with a list of available services. Use the up and down arrow keys to locate the service you want. With the cursor on a service, press the Spacebar to toggle the service on or off. Press the Tab key to highlight the OK button, and press the Spacebar to save the change and exit. The `ntsysv` tool only changes services for the current run level. You can run `ntsysv` with the `--level #` option, where # is replaced by the run level for which you want to change services.

Adding run-level scripts

Suppose you want to create and configure your own run-level script. For example, after installing the binaries for the fictitious `my_daemon` program, it needs to be configured to start up in run levels 3, 4, and 5, and terminated in any other run level. You can add the script to the `/etc/init.d` directory, then use the `chkconfig` command to configure it.

To use `chkconfig`, ensure that the following lines are included in the `/etc/init.d/my_daemon` script:

```
# chkconfig: 345 82 28
# description: Does something pretty cool - you really \
#    have to see it to believe it!
# processname: my_daemon
```

NOTE: The line `chkconfig: 345 82 28` sets the script to start in run levels 3, 4, and 5. It sets start scripts to be set to 82 for those run levels. It sets stop scripts to be set to 28 in all other levels.

With those lines in place, simply run the following command:

```
# chkconfig --add my_daemon
```

Appropriate links are created automatically. This can be verified with the following command:

```
# chkconfig --list my_daemon
```

The resulting output should look like this:

```
my_daemon 0:off 1:off 2:off 3:on 4:on 5:on 6:off
```

The script names that are created by `chkconfig` to make this all work are:

```
/etc/rc0.d/K28my_daemon
/etc/rc1.d/K28my_daemon
/etc/rc2.d/K28my_daemon
/etc/rc3.d/S82my_daemon
/etc/rc4.d/S82my_daemon
/etc/rc5.d/S82my_daemon
/etc/rc6.d/K28my_daemon
```

Managing xinetd services

There are a bunch of services, particularly network services, which are not handled by separate run-level scripts. Instead, a single run-level script called `xinetd` (formerly `inetd`) is run to handle incoming requests for these services. For that reason, `xinetd` is sometimes referred to as the *super-server*. The `xinetd` run-level script (along with the `xinetd` daemon that it runs) offers the following advantages:

- **Fewer daemon processes** — Instead of one (or more) daemon processes running on your computer to monitor incoming requests for each service, the `xinetd` daemon can listen for requests for many different services. As a result, when you type `ps -ax` to see what processes are running, dozens of fewer daemon processes will be running than there would be if each service had its own daemon.
- **Access control and logging** — By using `xinetd` to oversee the management of services, consistent methods of access control (such as PAM) and consistent logging methods (such as the `/var/log/messages` file) can be used across all of the services.

When a request comes into your computer for a service that `xinetd` is monitoring, `xinetd` uses the `/etc/xinetd.conf` file to read configuration files contained in the `/etc/xinetd.d` directory. Then, based on the contents of the `xinetd.d` file for the requested service, a server program is launched to handle the service request (provided that the service is not disabled).

Each server process is one of two types: single-thread or multithread. A single-thread server handles only the current request, whereas a multithread server handles all incoming requests for the service as long as there is still a client holding the process open. Then the multithread server closes and `xinetd` begins monitoring that service again.

The following are a few examples of services that are monitored by `xinetd`. The daemon process that is started up to handle each service is also listed.

- **eklogin** (`/usr/kerberos/sbin/klogind`) — Kerberos-related login daemon
- **finger** (`/usr/sbin/in.fingerd`) — Handles incoming finger requests for information from remote users about local users
- **gssftp** (`/usr/kerberos/sbin/ftpd`) — Kerberos-related daemon for handling file transfer requests (FTP)
- **ntalk** (`/usr/sbin/in.ntalkd`) — Daemon for handling requests to set up chats between a remote user and a local one (using the `talk` command)
- **rlogin** (`/usr/sbin/in.rlogind`) — Daemon for responding to remote login requests (from a remote `rlogin` command)
- **rsh** (`/usr/sbin/in.rshd`) — Handles requests from a remote client to run a command on the local computer

NOTE: You should normally not run `finger`, `rlogin`, or `rsh` due to security concerns with these commands.

Other services that can be launched by requests that come to `xinetd` include services for remote telnet requests, Samba configuration requests (swat), and Amanda network backups. A short description of each service is included in its `/etc/xinetd.d` file. Many of the services handled by `xinetd` are legacy services, including `rlogin`, `rsh`, and `finger`, that are considered insecure by today's security standards because they use clear-text passwords.

Manipulating run levels

Aside from the run level chosen at boot time (usually 3 or 5) and the shutdown or reboot levels (0 and 6, respectively), you can change the run level at any time while you're logged in (as root user). The `telinit` command (really just a symbolic link to `init`) enables you to specify a desired run level, causing the termination of all system processes that shouldn't exist in that run level, and starting all processes that should be running.

NOTE: The `telinit` command is also used to instruct `init` to reload its configuration file, `/etc/inittab`. This is accomplished with either the `telinit q` or the `telinit Q` commands.

For example, if you encountered a problem with your hard disk on startup, you may be placed in single-user mode (run level 1) to perform system maintenance. After the machine is stable, you can execute the command as follows:

```
# telinit 5
```

The `init` command handles terminating and starting all processes necessary to present you with a graphical login window.

Determining the current run level

You can determine the machine's current run level with the aptly named `runlevel` command. Using the previous example of booting into single-user mode and then manually changing the run level, the output of the `runlevel` command would be:

```
# runlevel
S 5
```

This means that the previous run level was S (for single-user mode) and the current run level is 5. If the machine had booted properly, the previous run level would be listed as N to indicate that there really wasn't a previous run level.

Changing to a shutdown run level

Shutting down the machine is simply a change in run level. With that in mind, other ways to change the run level include the `reboot`, `halt`, `poweroff`, and `shutdown` commands. The `reboot` command, which is a symbolic link to the `consolehelper` command, which in turn runs the `halt` command, executes a `shutdown -r now`, terminating all processes and rebooting the machine. The `halt` command executes `shutdown -h now`, terminating all processes and leaving the machine in an idle state (but still powered on).

Similarly, the `poweroff` command, which is also a link to the `consolehelper` command, executes a change to run level 0, but if the machine's BIOS supports Advanced Power Management (APM), it will switch off the power to the machine.

NOTE: A time must be given to the `shutdown` command, either specified as `+m` (representing the number of minutes to delay before beginning shutdown) or as `hh:mm` (an absolute time value, where `hh` is the hour and `mm` is the minute that you would like the shutdown to begin). Alternatively, `now` is commonly used to initiate the shutdown immediately.

Scheduling System Tasks

Frequently, you need to run a process unattended or at off-hours. The `at` facility is designed to run such jobs at specific times. Jobs you submit are spooled in the directory `/var/spool/at`, awaiting execution by the `at` daemon `atd`. The jobs are executed using the current directory and environment that was active when the job was submitted. Any output or error messages that haven't been redirected elsewhere are e-mailed to the user who submitted the job.

The following sections describe how to use the `at`, `batch`, and `cron` facilities to schedule tasks to run at specific times. These descriptions also include ways of viewing which tasks are scheduled and deleting scheduled tasks that you don't want to run anymore.

Using at.allow and at.deny

There are two access control files designed to limit which users can use the `at` facility. The file `/etc/at.allow` contains a list of users who are granted access, and the file `/etc/at.deny` contains a similar list of those who may not submit `at` jobs. If neither file exists, only the superuser is granted access to `at`. If a blank `/etc/at.deny` file exists (as in the default configuration), all users are allowed to utilize the `at` facility to run their own `at` jobs. If you use either `at.allow` or `at.deny`, you aren't required to use both.

Specifying when jobs are run

There are many different ways to specify the time at which an `at` job should run (most of which look like spoken commands). Table 12-6 has a few examples. These are not complete commands — they only provide an example of how to specify the time that a job should run.

Table 12-6: Samples for Specifying Times in an at Job

Command Line	***When the Command Is Run***
`at now`	The job is run immediately.
`at now + 2 minutes`	The job will start two minutes from the current time.
`at now + 1 hour`	The job will start one hour from the current time.
`at now + 5 days`	The job will start five days from the current time.
`at now + 4 weeks`	The job will start four weeks from the current time.
`at now next minute`	The job will start in exactly 60 seconds.
`at now next hour`	The job will start in exactly 60 minutes.
`at now next day`	The job will start at the same time tomorrow.
`at now next month`	The job will start on the same day and at the same time next month.
`at now next year`	The job will start on the same date and at the same time next year.
`at now next fri`	The job will start at the same time next Friday.
`at teatime`	The job will run at 4 p.m. They keywords noon and midnight can also be used.
`at 16:00 today`	The job will run at 4 p.m. today.
`at 16:00 tomorrow`	The job will run at 4 p.m. tomorrow.
`at 2:45pm`	The job will run at 2:45 p.m. on the current day.
`at 14:45`	The job will run at 2:45 p.m. on the current day.
`at 5:00 Apr 14 2008`	The job will begin at 5 a.m. on April14, 2008.
`at 5:00 4/14/08`	The job will begin at 5 a.m. on April 14, 2008.

Submitting scheduled jobs

The at facility offers a lot of flexibility in how you can submit scheduled jobs. There are three ways to submit a job to the at facility:

- **Piped in from standard input** — For example, the following command will attempt to build the Perl distribution from source in the early morning hours while the machine is likely to be less busy, assuming the perl sources are stored in /tmp/perl:

  ```
  echo "cd /tmp/perl; make ; ls -al" | at 2am tomorrow
  ```

  An ancillary benefit to this procedure is that a full log of the compilation process will be e-mailed to the user who submitted the job.

- **Read as standard input** — If no command is specified, at will prompt you to enter commands at the special at> prompt, as shown in the following example. You must indicate the end of the commands by pressing Ctrl+D, which signals an End of Transmission (<EOT>) to at.

  ```
  $ at 23:40
  at> cd /tmp/perl
  at> make
  at> ls -al
  at> <Ctrl-d>
  ```

- **Read from a file** — When the -f command-line option is followed by a valid filename, the contents of that file are used as the commands to be executed, as in the following example:

  ```
  $ at -f /root/bin/runme now + 5 hours
  ```

  This runs the commands stored in /root/bin/runme in five hours. The file can either be a simple list of commands or a shell script to be run in its own subshell (that is, the file begins with #!/bin/bash or the name of another shell).

Viewing scheduled jobs

You can use the atq command (effectively the same as at -l) to view a list of your pending jobs in the at queue, showing each job's sequence number, the date and time the job is scheduled to run, and the queue in which the job is being run.

The two most common queue names are a (which represents the at queue) and b (which represents the batch queue). All other letters (upper- and lowercase) can be used to specify queues with lower priority levels. If the atq command lists a queue name as =, it indicates that the job is currently running. Here is an example of output from the atq command:

```
# atq
2    2008-09-02 00:51 a  ericfj
3    2008-09-02 00:52 a  ericfj
4    2008-09-05 23:52 a  ericfj
```

Here you can see that there are three at jobs pending (job numbers 2, 3, and 4, all indicated as a). After the job number, the output shows the date and hour each job is scheduled to run.

Deleting scheduled jobs

If you decide that you'd like to cancel a particular job, you can use the atrm command (equivalent to at -d) with the job number (or more than one) as reported by the atq command. For example, using the following output from atq:

```
# atq
18      2008-09-01 03:00 a  ericfj
19      2008-09-29 05:27 a  ericfj
20      2008-09-30 05:27 a  ericfj
21      2008-09-14 00:01 a  ericfj
22      2008-09-01 03:00 a  ericfj
```

you can remove the jobs scheduled to run at 5:27 a.m. on September 29 and September 30 from the queue with the following command:

```
# atrm 19 20
```

Using the batch command

If system resources are at a premium on your machine, or if the job you submit can run at a priority lower than normal, the batch command (equivalent to at -q b) may be useful. It is controlled by the same atd daemon, and it allows job submissions in the same format as at submissions (although the time specification is optional).

However, to prevent your job from usurping already scarce processing time, the job will run only if the system load average is below a particular value. The default value is 0.8, but specifying a command-line option to atd can modify this. This was used as an example in the earlier section describing startup and shutdown. Here is an example of the batch command:

```
$ batch
at> du -h /home > /tmp/duhome
at> <Ctrl+d>
```

In this example, after I type the batch command, the at facility is invoked to enable me to enter the command(s) I want to run. Typing the du -h /home > /tmp/duhome command line has the disk usages for everything in the /home directory structure output to the /tmp/duhome file. On the next line, pressing Ctrl+D ends the batch job. As soon as the load average is low enough, the command is run. (Run the top command to view the current load average.)

Using the cron facility

Another way to run commands unattended is via the cron facility. Part of the cronie rpm package, cron addresses the need to run commands periodically or routinely (at least, more often than you'd care to manually enter them) and allows lots of flexibility in automating the execution of the command. (The cronie package contains an extended version of the cron utility for running scheduling tasks to run at a particular time, as well as adding security enhancements.)

As with the `at` facility, any output or error messages that haven't been redirected elsewhere are e-mailed to the user who submitted the job. Unlike using `at`, however, cron jobs are intended to run more than once and at a regular interval (even if that interval is only once per month or once per year).

Also like the `at` facility, `cron` includes two access control files designed to limit which users can use it. The file `/etc/cron.allow` contains a list of users who are granted access, and the file `/etc/cron.deny` contains a similar list of those who may not submit `cron` jobs. If neither file exists (or if `cron.deny` is empty), all users are granted access to `cron`.

There are four places where a job can be submitted for execution by the `cron` daemon `crond`:

- **The `/var/spool/cron/`*`username`* file** — This method, where each individual user (indicated by *username*) controls his or her own separate file, is the method used on UNIX System V systems.
- **The `/etc/crontab` file** — This is referred to as the *system crontab file*, and was the original crontab file from BSD UNIX and its derivatives. Only root has permission to modify this file.
- **The `/etc/cron.d` directory** — Files placed in this directory have the same format as the `/etc/crontab` file. Only root is permitted to create or modify files in this directory.
- **The `/etc/cron.hourly`, `/etc/cron.daily`, `/etc/cron.weekly`, and `/etc/cron.monthly` directories** — Each file in these directories is a shell script that runs at the times specified in the `/etc/crontab` file (by default, at one minute after the hour every hour; at 4:02 a.m. every day; Sunday at 4:22 a.m.; and 4:42 a.m. on the first day of the month, respectively). Only root is allowed to create or modify files in these directories.

The standard format of an entry in the `/var/spool/cron/`*`username`* file consists of five fields specifying when the command should run: minute, hour, day of the month, month, and day of the week. The sixth field is the actual command to be run.

The files in the `/etc/cron.d` directory and the `/etc/crontab` file use the same first five fields to determine when the command should run. However, the sixth field represents the name of the user submitting the job (because it cannot be inferred by the name of the file as in a `/var/spool/cron/`*`username`* directory), and the seventh field is the command to be run. Table 12-7 lists the valid values for each field common to both types of files.

Table 12-7: Valid /etc/crontab Field Values

Field Number	*Field*	*Acceptable Values*
1	minute	Any integer between 0 and 59
2	hour	Any integer between 0 and 23, using a 24-hour clock
3	day of the month	Any integer between 0 and 31
4	month	Any integer between 0 and 12, or an abbreviation for the name of the month (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec)
5	day of the week	Any integer between 0 and 7 (where both 0 and 7 can represent Sunday, 1 is Monday, 2 is Tuesday, and so on), or abbreviation for the day (Sun, Mon, Tue, Wed, Thu, Fri, Sat)

The latest version of cron (cronie and crontabs packages) includes the ability to indicate that a cron job be run at boot time.

Refer to the crontab man page (type **man 5 crontab**) for information on using the reboot option to have a command run once at startup time.

NOTE: The cronie package replaces the older vixie-cron.

An asterisk (*) in any field indicates all possible values for that field. For example, an asterisk in the second column is equivalent to 0,1,2 . . . 22,23, and an asterisk in the fourth column means Jan,Feb,Mar . . . Nov,Dec. In addition, lists of values, ranges of values, and increments can be used. For example, to specify the days Monday, Wednesday, and Friday, the fifth field could be represented as the list Mon,Wed,Fri. To represent the normal working hours in a day, the range 9–17 could be specified in the second field. Another option is to use an increment, as in specifying 0–31/3 in the third field to represent every third day of the month, or */5 in the first field to denote every five minutes.

Lines beginning with a # character in any of the `crontab`-format files are comments, which can be very helpful in explaining what task each command is designed to perform. It is also possible to specify environment variables (in Bourne shell syntax, for example, `NAME="value")` within the `crontab` file. Any variable can be specified to fine-tune the environment in which the job runs, but one that may be particularly useful is `MAILTO`. The following line sends the results of the `cron` job to a user other than the one who submitted the job:

```
MAILTO=otheruser
```

If the following line appears in a `crontab` file, all output and error messages that haven't already been redirected will be discarded:

```
MAILTO=
```

Modifying scheduled tasks with crontab

The files in /var/spool/cron should not be edited directly. They should only be accessed via the crontab command. To list the current contents of your own personal crontab file, type the following command:

```
$ crontab -l
```

All crontab entries can be removed with the following command:

```
$ crontab -r
```

Even if your personal crontab file doesn't exist, you can use the following command to begin editing it:

```
$ crontab -e
```

The file automatically opens in the text editor that is defined in your EDITOR or VISUAL environment variables, with vi as the default. When you're done, simply exit the editor. Provided there were no syntax errors, your crontab file will be installed. For example, if your user name is jsmith, you have just created the file /var/spool/cron/jsmith. If you add a line (with a descriptive comment, of course) to remove any old core files from your source code directories, that file may look similar to this:

```
# Find and remove core files from /home/jsmith/src
5 1 * * Sun,Wed find /home/jsmith/src -name core.[0-9]* -exec rm {} \; >
/dev/null 2>&1
```

Because core files in Fedora and RHEL consist of the word core, followed by a dot (.) and process ID, this example will match all files beginning with core. and followed by a number. The root user can access any user's individual crontab file by using the -u *username* option to the crontab command.

Understanding cron files

Separate cron directories are set up to contain cron jobs that run hourly, daily, weekly, and monthly. These cron jobs are all set up to run from the /etc/crontab file. The default /etc/crontab file is empty. Under the hood, though, cron runs the hourly, daily, weekly, and monthly jobs as if the crontab file looks like this:

```
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
HOME=/

# run-parts
01 * * * * root run-parts /etc/cron.hourly
02 4 * * * root run-parts /etc/cron.daily
22 4 * * 0 root run-parts /etc/cron.weekly
42 4 1 * * root run-parts /etc/cron.monthly
```

The first four lines initialize the run-time environment for all subsequent jobs (the subshell in which jobs run, the executable program search path, the recipient of output and error messages, and that user's home directory). The next five lines execute (as the user root) the `run-parts` program that controls programs that you may want to run periodically.

`run-parts` is a shell script that takes a directory as a command-line argument. It then sequentially runs every program within that directory (shell scripts are most common, but binary executables and links are also evaluated). The default configuration executes programs in `/etc/cron.hourly` at one minute after every hour of every day; `/etc/cron.daily` at 4:02 a.m. every day; `/etc/cron.weekly` at 4:22 a.m. on Sundays; and `/etc/cron.monthly` at 4:42 a.m. on the first day of each month.

Here are examples of files that are installed in `cron` directories for different software packages:

- **`/etc/cron.daily/logrotate`** — Automates rotating, compressing, and manipulating system logfiles.
- **`/etc/cron.daily/makewhatis.cron`** — Updates the whatis database (contains descriptions of man pages), which is used by the `man -k`, `apropos`, and `whatis` commands to find man pages related to a particular word.
- **`/etc/cron.daily/mlocate.cron`** — Updates the `/var/lib/mlocate/mlocate.db` database (using the `updatedb` command), which contains a searchable list of files on the machine.
- **`/etc/cron.daily/tmpwatch`** — Removes files from `/tmp`, `/var/tmp`, and `/var/catman` that haven't been accessed in ten days.

The `makewhatis.cron` script installed in `/etc/cron.weekly` is similar to the one in `/etc/cron.daily`, but it completely rebuilds the `whatis` database, rather than just updating the existing database.

Finally, in the `/etc/cron.d` directory are files that have the same format as `/etc/crontab` files.

NOTE: If you are not comfortable working with `cron` from the command line, there is a KCron Task Scheduler window that comes in the kdeadmin package for managing cron tasks. To launch KCron, type **kcron** from a Terminal window.

Summary

Shell scripts are an integral part of a Fedora or RHEL system for configuring, booting, administering, and customizing Fedora or RHEL. They are used to eliminate typing repetitive commands. They are frequently executed from the scheduling facilities within Fedora or RHEL, allowing much flexibility in determining when and how often a process should run. They also control the startup of most daemons and server processes at boot time.

The `init` daemon and its configuration file, `/etc/inittab`, also factor heavily in the initial startup of your Fedora or RHEL system. They implement the concept of run levels that is carried out by the shell scripts in `/etc/rc.d/init.d`, and they provide a means by which the machine can be shut down or rebooted in an orderly manner.

To have shell scripts configured to run on an ongoing basis, you can use the `cron` facility. `Cron` jobs can be added by editing `cron` files directly or by running commands such as `at` and `batch` to enter the commands to be run.