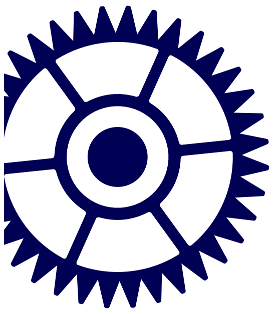
Shell scripting

Automate common tasks

If repetitive operations bore you, Bob Moss can help you automate them.



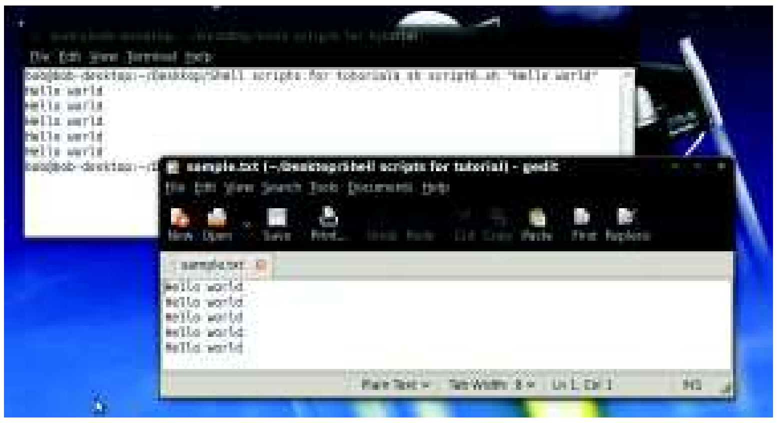
We all spend endless hours completing repetitive tasks on our Linux boxes when we could be doing something far more interesting (like playing *Alien Arena*, for example). Wouldn't it be great if you could just write a small script that does it all for you, then schedule it to run automatically?

Those of you who have ever automated tasks in Windows will be familiar with the idea of writing a VBScript, and shell-scripting is not entirely dissimilar. You simply create a text file with a **.sh** extension, type a set of terminal commands on each line in the order they should be executed, then launch your script from a terminal using:

```
sh script.sh
```

The key difference is that your shell script runs with root permissions only if you specify this on the command line, meaning a rogue shell script can't wreak havoc through your entire system. Also, shell scripts are interpreted from the shell (as the name suggests) while a VBScript relies on the presence of the Windows Script Host. An added bonus is that the *Bash* terminal can also be used in OS X, BSD and other Unix variants, so your scripts will be portable.

› Don't leave visitors stumped with the default 'Drupal Powered By Bitnami' title.



```
bob@bob-desktop:~/Desktop/shell scripts for tutorial$ sh script6.sh "Hello world"
Hello world
Hello world
Hello world
Hello world
Hello world
bob@bob-desktop:~/
```

sample.txt (~/Desktop/shell scripts for tutorial) - gedit

```
Hello world
Hello world
Hello world
Hello world
Hello world
```

File Edit View Terminal Help
File Edit View Search Tools Documents Help
New Open Save Print... Undo Redo Cut Copy Paste Find Replace
sample.txt
Plain Text Tab Width: 8 Ln 1, Col 1 INS

It's almost obligatory these days to run a "Hello, world!" tutorial to introduce coding concepts, and there'll be no break from tradition here. Open your favourite text editor, create a file called **script_tutorial.sh** and add the following lines to it:

```
#!/bin/sh
# My first shell script
#
clear
echo "Hello, world!"
```

The first line defines which terminal should be used to execute the script (we've chosen *sh* because it's portable), and in the second line we've included a nice name inside the comments field so we know what the shell script does. The commands underneath then clear the terminal before printing our **"Hello, world"** line to the terminal window. This is the general structure that seems to be prevalent in most shell script files, and this is also the form that people will expect if you head to the forums or redistribute your scripts.

My first shell script

With the next step we acknowledge that hard-coding values is not good programming practice. We need to add some variables to our shell script – to do this, simply replace the last line of your shell script with:

```
var="Hello, world!"
echo $var
```

Here we declare and set the variable named **var** to store **"Hello, world!"** and then output the variable contents. You'll notice there's no type-checking with shell variables so you will need to add your own tests later as necessary.

We can now extend this further by enabling you to define the output text as an argument from the command line rather than from the script itself. Replace the **"Hello, World!"** in the previous snippet with **$1**. Now if you run the following in a terminal:

```
sh script_tutorial.sh "Hello, world\!"
```

you should see the this text appear as the output. Notice how we have to escape the exclamation mark with a backslash to provide the correct result.

This is all well and good, but what if we want to store this useful information more permanently? Let's take a look at the following code, which you can use to replace everything below the **clear** line:

```
echo $1 >> sample.txt
cat < sample.txt
#rm -rf sample.txt
```

The top line appends our **"Hello, World!"** string to **sample.txt**, but if the file doesn't exist already then it's created automatically. The next line then reads the contents of the file and prints the output to the terminal for you to read. You can optionally uncomment the bottom line by removing the hash. The reason this is commented is to show that if the file already exists, the script will keep appending the argument text to a new line each time. However, if you uncomment the final line the text file will be removed each time you run the script, which can be useful if you want to see the text you've entered as an argument on this occasion only.

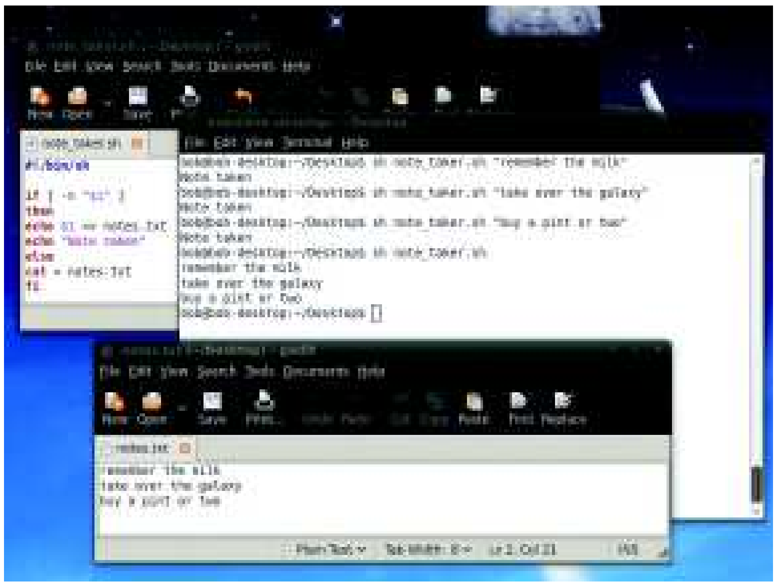
Ifs and loops

You can also automate decisions in your shell script using the **if**, **elif** and **else** clauses. Try using the code in Listing 1 to replace the contents of your current shell script:

```
Listing 1: shellscript_tutorial.sh
1  #!/bin/sh
2
3  if [ -n "$1" -a "$1" = "Hello world" ]
4  then
5  echo $1 >> sample.txt
6  cat < sample.txt
7  rm -rf sample.txt
8  elif [ -n "$1" ]
9  then
10 echo "Try Harder!"
11 else
12 echo "FAIL!"
13 fi
14
15 i=0
16 case $i in
17          0) echo "zero";;
18          1) echo "one";;
19          2) echo "two";;
20 esac
```

In line three you'll notice that we haven't included an exclamation mark in our **"Hello world"** string. It makes sense that this doesn't work by default, because if we were handling a file address that uses \ as an escape character somewhere in the path, we would want it to remain intact, so it can be used later in the script.

We use **-n** to make sure that the user has actually entered some text as a parameter by checking the value is not **null**, then we check that the parameter contains **"Hello world"** (notice we use **-a** to specify 'and'). If these conditions are met we execute the same commands we did in the previous example. Otherwise we check that the user has still managed to enter something, and if so, the script will heckle them a little for not entering the correct text! But if all else fails we use line 12 to output **FAIL!** in big capital letters in the terminal. We could also extend this by adding nested if statements for situations where this is necessary, and this will work so long as each **if** has a closing **fi**. Just remember that they always take the following form:



```
#!/bin/sh

if [ -n "$1" ]
then
echo $1 >> notes.txt
echo "Note taken"
else
cat < notes.txt
fi
```

```
bob@bob-desktop:~/Desktop$ sh note_taker.sh "remember the milk"
Note taken
bob@bob-desktop:~/Desktop$ sh note_taker.sh "take over the galaxy"
Note taken
bob@bob-desktop:~/Desktop$ sh note_taker.sh "buy a pint or two"
Note taken
bob@bob-desktop:~/Desktop$ sh note_taker.sh
remember the milk
take over the galaxy
buy a pint or two
bob@bob-desktop:~/Desktop$
```

```
remember the milk
take over the galaxy
buy a pint or two
```

› Shell scripting is so simple that you too could be taking quick notes with under 10 lines of code.

```
if [condition]
then
<command>
elif [condition]
then
<command>
else
<command>
fi
```

But you aren't restricted to just this type of conditional statement. As demonstrated in lines 15–20 you can also use a switch to accomplish much the same end. You will also notice that for the first time you need to suffix the end of each case with two semicolons in order to for execution to flow correctly. In the example the variable **i** stores a value of zero. This is then evaluated by the switch, and it produces the output that we specify. As a result, if you enter **"Hello world"** as an argument with this script example you should see the following output:

```
Hello world
zero
```

Using our new-found knowledge we could also include a loop to append our text several times to the same file when you run the script. Thankfully this is a relatively trivial exercise, as this example shows:

```
if [ -n "$1" ]
then
for x in 1 2 3 4 5
do
            echo $1 >> sample.txt
done
cat < sample.txt
rm -rf sample.txt
else
echo "No text for me to act on!"
fi
```

Here we check that the user has added text as a parameter, and if so append it five times to **sample.txt** (this is the number of iterations in our loop). We then read from the file and print to the contents to the terminal window (the output should be the argument text repeated five times!) before destroying the text file. If you didn't send any text to be appended, the script generates an error message that will appear so the user can see that they need to try again. »

Quick tip

You could also use -o or -x for 'or' or 'xor' when you define conditions within your if statements. Also, if you need < and >, simply use -l and -g.

Script a backup solution

Put your new-found knowledge to good use and secure your data.

Quick tip

For a quick way to verify that your backups haven't been tampered with you should also see the MD5 feature in **LXF123** Subscribers will find a copy in PDF format on the *Linux Format* website.

To give a practical example of scripting in action, we'll wrote a simple backup script that copies a source folder to another location. Take a look at the code in Listing 2 and you'll see that in line three that we use **-d** as our criterion in the conditional statement. This checks whether both of the directories you specify in the script actually exist. If so, you simply copy from source to destination. If not, we use line seven to check if the source directory exists (in which case we simply create the destination directory to copy to).

However, if the directory of important files you want to backup doesn't exist, line 13 prints a helpful error message to the terminal window. After all, we need the files to backup in the first place to make this a useful script!

```
Listing 2: backup.sh
01 #!/bin/sh
02
03 if [ -d $1 -a -d $2 ]
04 then
05 cp $1/* $2
06 echo "Backup complete"
07 elif [ -d $1 ]
08 then
09 mkdir $2
10 cp $1/* $2
11 echo "Backup complete"
12 else
13 echo "Unable to locate source directory"
14 fi
```

The code snippet you see above is not necessarily the best way of keeping repeated backups of a particular directory. For example, any backup files that have been automatically generated in the folder will still be retained (use **ls** in your documents folder to see how many filenames you have with a tilde on the end) and **cp** doesn't transfer changes to existing files, so you would have to remove the destination folder and copy everything over each time. Though we could use an **rm -rf** command in listing 2 to make this a more incremental backup, it still wouldn't make the backup any more efficient, as it simply adds an additional delete operation that uses more resources and slows the script as a result. But as a one-time backup the above script is sufficient for most purposes.

Another method we could use to make this script something we could use more often would be to use *rsync* instead. Suddenly we have a basic incremental backup system. Simply replace every line beginning with **cp** in Listing 2 with the following:

```
rsync --recursive --times --perms --exclude "~*" --exclude
"*bak" $1 $2
```

We use **recursive** to loop through each subdirectory, then **times** and **perms** to maintain file permissions and time stamps (which is useful information for subsequent *rsyncs*). Those two parameters are then followed by two **exclude** methods to remove those automatically generated backup files mentioned earlier, before specifying the source and destination locations. You will now find that file changes are transferred, and after the first run your backups should take considerably less time to complete, as you're transferring file changes rather than whole folders.

Shrink it down

The next step is to compress our backup into a tarball (also referred to as a gzip archive on some Linux systems) by using the script you can see in Listing 3.

```
Listing 3: compress.sh
01 #!/bin/sh
02
03 if [ -f $1/../backup.tar.gz ]
04 then
05 tar -uz --file=$1/../backup.tar.gz $1
06 elif [ -d $1 ]
07 then
08 tar -cz --file=$1/../backup.tar.gz $1
09 echo "Backup packaged"
10 else
11 echo "Backup packaging failed"
12 fi
```

Here we use the **tar** command to create an archive or update an existing one. You'll notice in line three that we use **-f** in our conditional statement to check whether our backup tarball already exists. If so, we can simply update the tarball. As we saw from Listing 2, we could just delete the existing tarball and create a new one, but it's much simpler to use the **-u** argument in the **tar** command as this will update the existing archive without needing to extract or repackage the file.

If we don't have a tarball but the backup directory exists, then we use line eight to create a new archive from that folder. We could optionally use an **rm -rf** line to tidy up afterwards. If all else fails, we print an error message people.

Note that as things stand this script only needs you to specify the backup directory and not the source. However, you can convert this routine into a standalone backup by changing all instances of **$1** to **$2**, excluding lines five and eight, and from then on you'll need to specify a source directory to back up.

› **Insure your favourite pictures and holiday snaps with our backup scripts.**

Now we have a backup of our files, it makes sense for us to have a script to quickly restore the files in that directory. For this we can use the code in Listing 4. Note that when you run this sample on the command line you will need to specify the backup directory first before the destination.

Listing 4: restore.sh

```
01 #!/bin/sh
02
03 if [ -f $1/../backup.tar.gz ]
04 then
05 tar -xz --file=$1/../backup.tar.gz
06 rsync --recursive --times --perms $1 $2
07 rm -rf $1
08 echo "Restore successful"
09 elif [ -d $1 ]
10 rsync --recursive --times --perms $1 $2
11 tar -cz --file=$2/../backup.tar.gz $1
12 rm -rf $1
13 echo "Restore successful"
14 then
15 else
16 echo "Restore failed"
17 fi
```

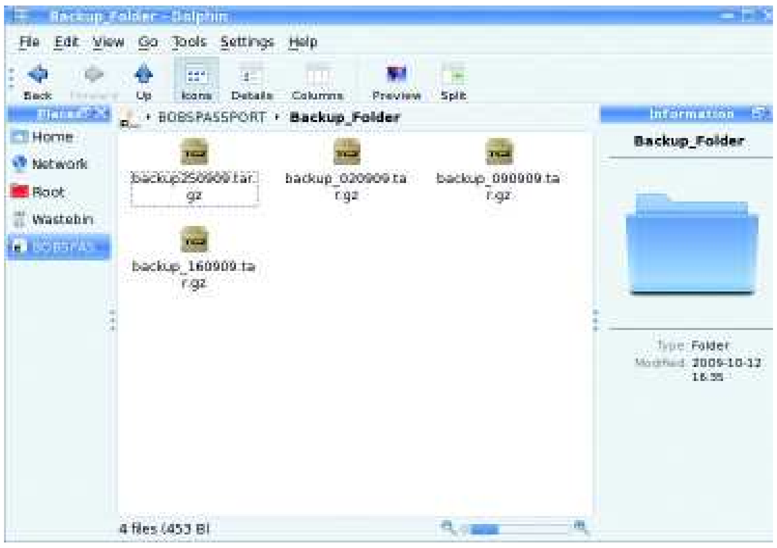
Once again we check whether the archive exists, in which case we extract it and use **rsync** to move files back to the relevant directory. We could have used **cp**, but we would lose our timestamps and permissions as a result. You will also notice that we don't need the **exclude** lines in this instance, as there will be no automatically generated backup files in your archive thanks to our previous backup scripts. We then clean up by deleting the directory where we have just extracted the tarball contents in order to save disk space.

If the archive doesn't exist, or you've specified a directory rather than an archive, then lines 10–13 restore your files, produce an archive of your backup then remove the backup folder you initially specified to save disk space once again.

These conditions ensure that we always produce the same result each time we use the restore script if the user enters valid input. If neither the backup directory nor backup archive can be located, then we output a simple human-readable error message.

There are endless ways you could extend this project. You could schedule your backup to run automatically using crontab (see box, below), have your computer tell you when the job is done using a speech synthesiser such as *Festival* or *eSpeak*, or even rewrite the scripts to keep archives for several dates or times. And if backups aren't your thing, you could write your own startup script to run at login that automatically opens up your favourite applications (which is quicker to manage than Gnome and KDE 4's GUI methods). You could convert your entire music collection from MP3 to Ogg Vorbis format. Any task that you find repetitive or needs to be executed a large number of times can usually be automated in a shell script, so the possibilities are limitless. If you produce anything particularly special or interesting, do let us know at the usual address and share your hard work. LXF

"You could write a script that automatically opens your favourite apps."



› **You could extend the backup scripts to retain archived snapshots, so you can easily roll back to any moment in time.**

Scheduling workshop

By far the easiest way to schedule a script is to make it run at startup thanks to the GUI methods provided by most modern Linux desktops. Within Gnome, head to System > Preferences > Startup Applications and simply add your shell script as an entry. KDE 4 users can accomplish much the same thing by copying the script into **~/.kde/Autostart**, and then by typing the following into a root terminal:

```
chmod +x script.sh
```

If you want finer control of when your script runs, use crontab, a native task scheduler. You'll need to ensure from the outset that you are able to access crontab, and to do this you'll need to check your system for files named **crontab.allow** or **crontab.deny**. If neither exist then you can only access crontab as a root user, but if they do exist, ensure your user is in the first file and not in the latter. You can now fire up a terminal and type the following commands:

```
export EDITOR=nano
crontab -e
```

You could now add something similar to the following line to the file:

```
30 18 * * * sh ~/script.sh
```

Once you have pressed Ctrl+X, follow the instructions to save and exit. Crontab will now run **script.sh** in your home directory at 6.30 pm every day. To make sense of this line, take a look at the following table:

In true American fashion, crontab's week starts on a Sunday (day 0).

Timescale	Minutes	Hours	Days	Month	Day
Range	0–59	0–23	1–31	1–12	0–6

You will also notice from the command we added before that simply using a wildcard character * as a field value will run the script each time the value increases, meaning you could for example run hourly and daily tasks if you so choose.

Every time your script is run crontab will try to send your user an email to say how things went, which is useful for server administrators but not necessarily useful for those of you who simply want to schedule actions on your home machines. We can optionally suppress the email by editing our previous line as follows:

```
30 18 * * * sh ~/script9.sh > /dev/null 2>&1
```