The `#spawn` parameter specifies which commands are to be automated.

The `#expect` parameter provides the expected message.

`#send` is the message to be sent. `# expect eof` defines the end of the command interaction.

Making commands quicker by running parallel processes

Computing power has increased a lot over the last couple of years. However, this is not just because of having processors with higher clock cycles; the thing that makes modern processors faster is multiple cores. What this means to the user is in a single hardware processor there are multiple logical processors.

However, the multiple cores are useless unless the software makes use of them. For example, if you have a program that does huge calculations, it will only run on one of the cores, the others will sit idle. The software has to be aware and take advantage of the multiple cores if we want it to be faster.

In this recipe we will see how we can make our commands run faster.

How to do it...

Let us take an example of the `md5sum` command that we discussed in the previous recipes. This command is CPU-intensive as it has to perform the calculation. Now, if we have more than one file that we want to generate a checksum of, we can run multiple instances of `md5sum` using a script like this:

```
#/bin/bash
#filename: generate_checksums.sh
PIDARRAY=()
for file in File1.iso File2.iso
do
  md5sum $file &
  PIDARRAY+=("$!")
done
wait ${PIDARRAY[@]}
```

When we run this, we get the following output:

```
$ ./generate_checksums.sh
330dcb53f253acdf76431cecca0fefe7  File1.iso
bd1694a6fe6df12c3b8141dcffaf06e6  File2.iso
```

The output will be the same as running the following command:

```
md5sum File1.iso File2.iso
```

However, as the `md5sum` commands ran simultaneously, you'll get the results quicker if you have a multi-core processor (you can verify this using the `time` command).

How it works...

We exploit the Bash operand `&`, which instructs the shell to send the command to the background and continue with the script. However, this means that our script will exit as soon as the loop completes while the `md5sum` processes are still running in the background. To prevent this, we get the PIDs of the processes using `$!`, which in Bash holds the PID of the last background process. We append these PIDs to an array and then use the `wait` command to wait for these processes to finish.