stdin, stdout, stderr

Linux is built being able to run instructions from the command line using switches to create the output.

The question of course is how do we make use of that?

One of the ways to make use of this is by using the three special file descriptors - stdin, stdout and stderr.

> Under normal circumstances every Linux program has three streams opened when it starts; one for input; one for output; and one for printing diagnostic or error messages. These are typically attached to the user's terminal (see man tty(4)) but might instead refer to files or other devices, depending on what the parent process chose to set up.
>
> --Taken from the BSD Library functions manual - STDIN(3)

Table 1.2. Standard Input, Standard Output and Standard Error

Type	Symbol
standard input	0<
standard output	1>
standard error	2>

stdin

Generally standard input, referred to as "stdin", comes from the keyboard.

When you type stuff, you're typing it on stdin (a standard input terminal). A standard input device, which is usually the keyboard, but Linux also allows you take standard input from a file.

For example:

```
cat < myfirstscript
```

This would tell cat to take input from the file myfirstscript instead of from the keyboard (This is of course the same as: **cat myfirstscript**).

Essentially what this boils down to is that the input for this command (cat) is no longer coming from where it was expecting to come from (the keyboard), but is now coming from a file.

Linux associates the file descriptor 0 with standard input. So, we could have said:

```
cat 0< myfirstscript
```

which would have produced the same as the previous command.

Why could we leave out the 0?

Since, at the time of creating a process, one standard input is associated with the process.

stdout

Standard output, as created at process creating time, goes to the console, your terminal or an X terminal. Exactly where output is sent clearly depends on where the process originated.

Our console or terminal should be the device that is accepting the output. Running the command:

```
cat file
```

would [con]catenate the file, by default, to our standard output i.e. our console or terminal screen. (Where the process originated.)

We can change this and redirect the output to a file.

Try the following:

```
ls -al myfirstscript > longlisting
```

This redirects the output not to the screen, but rather to a new file 'longlisting'. The process of redirection using the '>' will create the file 'longlisting' if it was not there. Alternately, if the file 'longlisting' existed, it would remove it, (removing the contents too of course) and put a new file there with the directory listing of "myfirstscript" within it.

How would we see the contents of the file?

```
cat longlisting
```

This will show the size, the owner, the group and the name of the file myfirstscript inside the file 'longlisting'.

In this example, the output of the **ls** command has not gone to the standard output (the screen by default), but rather into a file called 'longlisting'.

Linux has got a file descriptor for standard output, which is 1 (similar to the 0 for standard input file descriptor).

The above **ls -la** example can be rewritten as:

```
ls -al myfirstscript 1> longlisting
```

which, would do the same thing as leaving out the file descriptor identifier and just using the greater than sign.

In the same way we use our standard input as < (or a 0<), we use a > (or a 1>) to mean standard output.

Returning to the cat example above, we could type data on the command line that gets sent directly to a file. If the file is not there it will create it and will insert the content we typed on the command line, into the file. This is illustrated below:

```
$ cat > test
This is the first line.
This is the second line.
This is the final line. < Ctrl-d pressed here >
```

```
$ cat test
This is the first line.
This is the second line.
This is the final line.
```

Doing this does not return us to a prompt. Why? What is it waiting for?

It's waiting for us to actually enter our string into a buffer. You should start typing a sentence then another sentence, and another, etc. Each time you type a character, it's getting appended to the file 'newfile'.

When you have finished typing in what you want, press Ctrl-d. The Ctrl-d (^D) character will send an end of file signal to cat thereby returning you to your prompt.

If you list your directory contents using:

```
ls -al
```

you should see the file 'newfile'. This file is the one that you've just created on the command line.

```
cat newfile
```

will show you the contents of 'newfile' displayed onto the standard output.

Now why did all of this work? It worked because cat was taking its input from standard input and was putting its output not to standard out as normal, but was rather redirecting output to the file 'newfile'.

On typing the command and hitting enter, you are not returned to your prompt since the console is expecting input to come from stdin; you type line after line of standard input followed by ^D. The ^D stopped the input, by sending an end-of-file character to the file, hence the file 'newfile' is created.

Question: What do you think tac > newFile will do?

Using stdin and stdout simultaneously

If you decide you want to copy the contents of two files to another file (instead of using the **cp** command - there is more than one way to skin a cat in Linux) you could do the following:

```
cat < myfirstscript > mynewscript
```

Incidentally, this is equivalent to

```
cp myfirstscript mynewscript
```

Appending to a file

Well that's fine and dandy, but what happens if we don't want to delete our longlisting script and want to rather append it to a file that's already there.

Initially we typed:

```
ls -al myfirstscript > longlisting
```

If you did this a second time, it would overwrite the first file longlisting. How could you append to it? Simply adding two greater than signs, immediately following one another as in the example below, would append output to the file 'longlisting'

```
ls -al myfirstscript >> longlisting
```

Each time you ran this command, it would not clobber (remove the contents of) the longlisting file but would rather append to it and as a result, the file 'longlisting' would grow in size.

A final note on standard output and standard input is that redirection must be the final command that you execute on the command line. In other words, you can't do any other command after the redirection. We will talk about this in a later section on pipes.

stderr

The final component in this dialog of file descriptors is standard error.

Every command could send it's output to one of two places: a) it could be valid output or b) it could be an error message.

It does the same with the errors as it does with the standard output; it sends them directly to your terminal screen.

If you run the command (as your user):

```
find / -name "*" -print
```

you would find that find would find a load of things, but it would also report a lot of errors in the form of 'Permissions denied. . .'.

Perhaps we're not interested in 'Permission denied...' - we may wish to discard these messages (as root, no error messages would be returned).

If we ran the command, we could put standard error into a file (remember standard error by default is going to the console; the same place as stdout). In this case I'm going to put it into a special file on the Linux system called **`/dev/null`**.

`/dev/null` is similar to the "Recycle Bin" under Windows except it's a waste paper basket with a point of no return - the Linux black hole! Once information has gone into **`/dev/null`**, it's gone forever.

Initially, I'm going to put any errors that the find command generates into **`/dev/null`**, because I'm not interested in them.

We saw that standard input was file descriptor 0, the standard output was file descriptor was 1, so no prizes for guessing that standard error has file descriptor 2.

Thus, the command

```
find / -name "*" -print 2> /dev/null
```

discards any errors that are generated by the find command. They're not going to pollute our console with all sorts of stuff that we're not interested in.

Note

Notice there is no space between the 2 and the >

We could do this with any command, we could for example say:

```
ls -al 2> myerror
```

Which would redirect all the error messages into a file called "myerror".

To recap we can use either:

```
<       OR      0<      for standard input
>       OR      1>       for standard output
but for standard error I have to use 2>
```

It's not optional to leave off the number two (2). Leaving it off would mean that the standard output would go to "myerror", including a 2 means standard error.

In the listing case of:

```
ls -al  2> myerror
```

puts any errors into a file called 'myerror'.

If we wanted to keep all those error messages instead of using a single greater than sign, we would use double greater than signs.

By using a single greater than sign, it would clobber the file 'myerror' if it exists, no different to standard output. By using a double greater than sign, it will append to the contents of the file called myerror.

```
ls -al 2>> myerror
```

Thus the contents of 'myerror' would not be lost.

stdout, stderr and using the ampersand (&)

With our new found knowledge, let's try and do a couple of things with the find command. Using the find command, I want to completely ignore all the error messages and I want to keep any valid output in a file. This could be done with:

```
find / -name "*" -print 2> /dev/null > MyValidOutput
```

This discards any errors, and retains the good output in the file "MyValidOutput". The order of the redirection signs is unimportant. Irrespective of whether standard output or standard error is written first, the command will produce the correct results.

Thus, we could've rewritten that command as:

```
find / -name "*" -print >MyValidOutput 2>/dev/null
```

Finally I could've appended the output to existing files with:

```
find / -name "*" -print >> output 2>> /dev/null
```

Clearly appending to **/dev/null** makes no sense, but this serves to illustrate the point that output and errors can both be appended to a file.

What happens if we want to take our standard output and put it into the same file as standard error? What we can do is this:

```
find / -name "*" -print 2> samefile 1>&#38;2
```

This means that standard error goes into a file called samefile and standard output goes to the same place as file descriptor 2 which is the file called samefile.

Similarly we could've combined the output by doing:

```
find / -name "*" -print 1> file 2>&#38;1
```

This captures the output of both standard output and standard error into the same file.

Clearly, we could've appended to a particular file instead of overwriting it.

```
find / -name "*" -print 1>> file 2>>&#38;1
```

Exercises:

1. Using the simple scripts from the previous exercises, ensure that all output from the df, du, vmstat, iostat commands are written to a file for later use.
2. Write a script to run the vmstat command every 10 seconds, writing output to a file /tmp/vmoutput. Ensure that the existing file is never clobbered.
3. Propend the date to the beginning of the output file created by the script in question 2 above. (put the date on the front - or top - of the file)

Unamed Pipes

Up to now, we've seen that we can run any command and we can redirect its output into a particular file using our redirection characters (>, <, >>, 2> or 2>>).

It would be good if we could redirect the output of one command into the input of another. Potentially we may want the output of the next command to become the input of yet another command and so forth. We could repeat this process over and over until we achieve the desired output.

In Linux, this is affected using the pipe character, (which is a vertical bar '|'). An example is:

```
ls -la | less
```

This would pass the output of the **ls -al** command to the input of the less command.

The effect would be to page your output one page at a time, rather than scrolling it to the standard output all in one go - too fast for you to be able to read, unless of course you are Steve Austin!.

What makes the pipe command powerful in Linux, is that you can use it over and over again.

We could type:

```
ls -l | grep myfirstfile | less
```

Instead of **grep**'s standard input coming from a file (or keyboard) it now comes from the **ls** command. The **grep** command is searching for a pattern (not a string) that matches myfirstfile. The output of the **grep** command becomes the input of the less command. Less could take its input from a keyboard, or from a file, but in this case it's taken its input from the command **grep**.

How many of these pipes can we have on a single line? As many as we want! We will see this and how it's used to good effect in our shell scripts from the next chapter onwards.

If pipes were not available, we may have to archive the above command in two or more steps. However, with pipes, this task is simplified (speeded up).

If we take a step back to run the **who** command:

```
who | grep <your user name>
```

Note

The < and > here don't mean redirection!

We will see only the people that are logged on as your user (hopefully that is only you!!). Perhaps you want to only see people who are logged on to pseudo terminals, then:

```
who | grep pts
```

which would tell us only the usernames of the people logged on to pseudo terminals.

In these examples we are using the output of the who command as the input to the **grep** command.

Additionally we could redirect the output of this command to a file called outfile:

```
who | grep pts > outfile
```

This would produce the file 'outfile' containing only those users who have logged onto pseudo terminals. That's nifty.

We will see this put to good effect as we start scripting, building very complex filters where we use the output of one command to be the input of the next, and the output of that to be the input of the next.