Chapter 3 The Shell: I/O And Job Control

3.5 I/O Redirection

It's not always the case that you want to run a program and see it output some data. Often times, you'll want that data to be output specifically to a file, or possibly you'd like to use that data output as an input to another file. You could sit around and copy paste data, but there is a significantly easier way with I/O Redirection.

Standard I/O

When a process is created, the operating system assigns it 3 *file descriptors:* 0, 1, and 2.

File Descriptor	Job
0	Standard Input (stdin): Connected to the keyboard for input
1	Standard Output (stdout): Connected to the terminal window for output
2	Standard Error (stderr): Connected to the terminal for error output

We use these 3 I/O channels all the time, but what is extremely useful is that we can change how those streams flow!

Redirection

I/O Channels can be redirected with `>`, `<`, and `|`.

Operators	Description
`>`	Redirects the standard output of a command from the terminal screen to a given file `cmd > filename` **Note:** If the file exists already it will be wiped, unless you turn on `nonclobber`: `set –o nonclobber` : turns nonclobber on (can't overwrite files) `set +0 nonclobber`: turns nonclobber off (so you can overwrite files)
`>>`	Works just like `>` however, instead of overwriting, this operator redirects standard out and appends it to the file if it already exists. If it doesn't, then a new file is created. `cmd >> filename`
`<`	The `<` operator redirects what would have been standard in so that you can use text files as input for a given program's input. `vim someFile < vimCommands` where `vimCommands` is a file: dd x ZZ
`|`	Pipes allow for you to redirect output from one program to be treated as input for the next! `cmd1 | cmd2` ex: `history | grep touch`

Note: While the default is to redirect standard out, it is possible to redirect the standard error as well:

`cmd > filename 2>&1` combines both stdout and standard error to write to the same file

`cmd > filename 2>errordata` Writes the standard out to `filename` and the standard error to `errordata`

3.6 Job Control

Suppose you have a number of processes that you have running in the background or have suspended. Over time, these processes can build up! To see the jobs your process is running, you can use the `jobs` command:

```
1 jobs
```

If you have nothing running, the command will print nothing. However, navigate to the directory where `takeforever.sh` exists and try running `takeforever.sh &` again so we can put it in the background.

```
./takeforever.sh &
```

> [1] 3797 [mjlny2@delmar ~]$ starting to work!

```
jobs
```

> [1]+ Running ./2750Materials/ch3/takeforever.sh &

`jobs` will print out all the jobs started from the our shell's process. The general output of jobs is:

```
[jobNumber]±  jobID Status  commandName
```

Bringing a Job Back to the Foreground

To bring a specified job back to the foreground, you can use the `fg` command:

```
./takeforever.sh &
```

> [1] 4208 [mjlny2@delmar ~]$ starting to work!

```
fg %+
```

> ./2750Materials/ch3/takeforever.sh doing more work! finishing up
>
> done!

So, above, when we ran jobs, we got a jobNumber of `1` with a `+`, and it was `Running`. When running working with specific jobs, you can refer to them by either their job number, their job ID, their name prefix, or by the `%+` or `%-`.

Operators	Description
`%+`	Refers to the most recently suspended job
`%-`	Refers to the second most recently suspended job

Note: We've seen that background jobs still print stdout to the console. In the event that the background job is waiting for input from a user, however, then the job itself will wait until it is in the foreground so that it can have access to the stdin.

If a Background job goes through its processes and finishes without an error it will print:

> [1]+ Done ./2750Materials/ch3/takeforever.sh

However, let's try to force a failure to see what happens. Copy `takeforever.sh` to `failFast.sh`. In there, replace the code with:

```
#!/bin/bash

echo starting to work, but gonna fail real quick!
sleep 2
echo get ready to fail
sleep x
```

Sleep does not take characters as input, and will fail the second it gets executed.

```
./failFast.sh &
```

> [1] 5107 [mjlny2@delmar ch3]$ starting to work, but gonna fail real quick! get ready to fail
> sleep: invalid time interval ‘x’ Try 'sleep --help' for more information.
>
> [1]+ Exit 1 ./failFast.sh

Quitting a Job

Many programs have their own methods for quitting:

Method of Quitting	Program
`ctrl + x`	Nano
`x+c`	Emacs
`:q`	vi/vim
`q`	mutt/man/etc
exit	shell
quit()	Python repl

Many programs have special ways to quit, and 99% of the time, quitting a program works just fine. However, for that other 1% where using the pre-defined quitting methods just don't work, there are ways to force a program to stop (even without bringing it to the foreground).

You can force any program to quit by using the `kill` command. This command sends a specific signal to the process that's running.

```
kill -9 jobNumber #or jobID or %±
```

The above command will send a signal to the process that is currently running the job. The `-9` is a specific signal (we'll discuss signals in greater detail in chapter 11). Let's take a look:

```
./takeforever.sh &
```

> [1] 5986 [mjlny2@delmar ch3]$ starting to work! doing more work!

```
jobs
```

> [1]+ Running ./takeforever.sh &

```
kill -9 %+
jobs
```

> [1]+ Killed ./takeforever.sh

Note: We haven't covered processes much, but it should be noted that JobID is not a process ID.

Finally, in the event that you have background jobs running, and you exit the shell, or kill the parent process of a background job, the jobs will all be terminated.