* Standard File Streams

-> when commands are executed by default there are three standard file streams always open for use: standard input (standard in or stdin), standard output (standard out or stdout) and standard error (or stderr).

Name	Symbolic Name	Value	Example
standard input	stdin	0	keyboard
standard output	stdout	1	terminal
standard error	stderr	2	log file

-> usually, stdin is your keyboard, and stout and stderr are printed on your terminal.

-> stderr is often redirected to an error logging file, while std in is supplied by ~~directory~~ directing input to come from a file or form the output of previous command through a pipe.

-> stdout is also often redirected into a file. Since stderr is where error messages (and warning) are written, usually nothing will go there.

-> In Linux, all open files are represented internally by what are called file descriptors.

-> simply put, these are represented by numbers starting at zero.

-> stdin is file descriptor 0, stdout is file descriptor 1, stderr is file descriptor 2.

-> Typically, if others files are opened in addition to these three, which are opened by default, they will start at file descriptor 3 and increase from there.

* I/o Redirection

-> Through the command shell, we can redirect the three standard file streams so that we can get input from either a file or another command

-> Instead of from our keyword and we can write output and errors to files or use them to provide input for subsequent commands.

→ For Example, if we have a Program called do_Something that reads from Stdin and write to Stdout and Stderr, we can change it's ~~input~~ Source by using the less-than sign (<) followed by name of the file to be consumed for input data

```
$ do_Something < input-file
```

→ If you want to send the output to a file, use the greater-than sign (>) as in:

```
$ do_Something > output-file
```

→ In fact, you can do both at same time as in

```
$ do_Something < input-file > output-file
```

→ Because stderr is not the same as stdout, error massage will still be Seen on the terminal windows in the above example.

-> If you want to redirect stderr to separate file, you use stderr's file descriptor number (2), the greater-than (>), followed by the name of the file you want to recive everything the running command writes to stderr:

```
$ do-something 2 > error-file
```

NOTE :- By the same logic, do-something 1 > output-file is the same as

```
do-something > output-file.
```

-> A special shorthand notation can send anything written to file descriptor 2 (stderr) to the same place as file descriptor 1 (stdout) : 2 > 1

```
$ do-something > all-output-file
   2>&1
```

bash permits an easier syntax for above

```
$ do-something >& all-output-file
```

* Pipes

-> In order to ~~d~~ use pipe, we can use the vertical-bar pipe symbol (|), between commands as in

```
$ command1 | command 2 | command3
```

-> The above represents what we often call a pipeline.

-> It allows Linux to combine the actions of several commands into one.

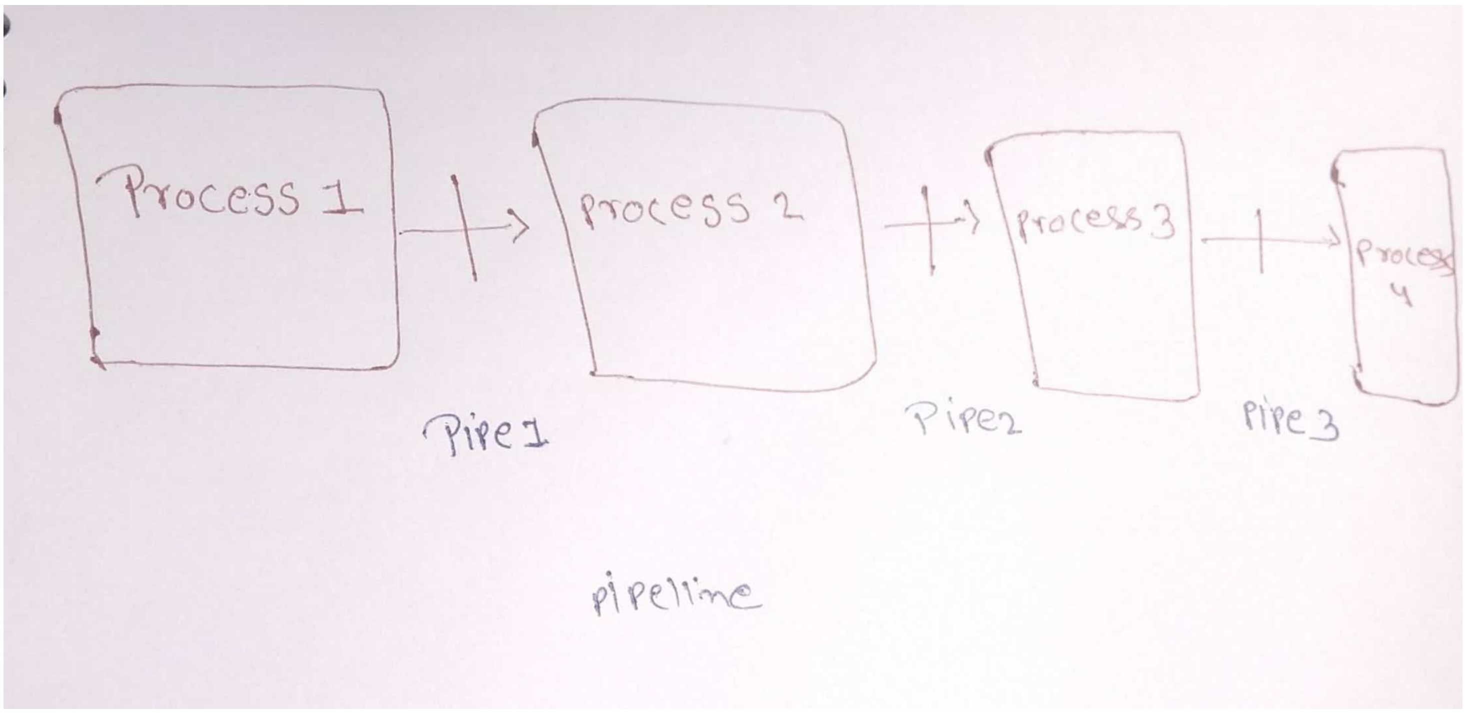
-> This is extraordinarily efficient because command2 and command3 do not have to wait for the previous pipeline commands to complate.

-> Computing power is much better utilized and things get done quicker.

-> Furthermore, there is no need to save output files between stages in the pipeline

-> which saves disk space and reduces reading and writing from disk

-> which often consitutes the slowest bottleneck in getting something done.


Process 1
Process 2
Process 3
Process 4
Pipe 1
Pipe 2
Pipe 3
pipeline