SoC_Week1

A Unix Journey: Ctrl+Alt+Delight

May 2024

1 Installation guide

Just do something and be able to run the terminal/shell. Ask in the group, in case of any trouble.

2 Unix command line

Unix dates back to the mid-1960s when the Massachusetts Institute of Technology, AT&T Bell Labs, and General Electric jointly developed an experimental time-sharing operating system called Multics for the GE-645 mainframe1. Unix was developed by Ken Thompson, Dennis Ritchie, and others in the AT&T Laboratories. The essence of Unix was not just to provide a good environment for programming but a system around which a fellowship could form. Unix is known for its simplicity, portability, multitasking, and multi-user capabilities. And most OS, nowadays we see, are Unix based, e.g., Linux(open source), FreeBSD, and Mac OS X, and here is just a nice and short article on this.

In short, *nix comes in handy for doing automation tasks, with an easy way for scripting, and there are a lot of scientific libraries and programs for the *nix system because there have been many contributions, thanks to the fact that its mainly open source(Linux).

Well, the first question that comes to mind is, what's the Command line? Well, the answer is simple, and by command line, we mean a way to interact with a computer's operating system, where the interface is in the form of text. Compared to our familiar GUI(Graphical User Interface), it doesn't have graphics like icons and windows to allow users to interact with the system; instead, it has CLI(Command Line Interface), which is text-based and requires knowledge of specific commands. The main advantage of using a CLI over a GUI is that a GUI uses pre-programmed interfaces. Thus, there is a restricted set of possible actions, but in this case, CLI uses pre-written programs and composes new scripts.

Shell: Next thing that comes is shell, which, in basic terms, helps us to interact with OS, using either CLI or GUI. Here are more standard descriptions:

A shell is a computer program that provides an interface for users to interact with the operating system. It interprets human-readable commands and converts them into something the operating system can understand. Shells can be either command-line (CLI) or graphical (GUI), depending on the computer's role and operation of the keystroke needed.

Now, Terminal is software for interacting with OS and uses/opens a shell. Terminals are based on text since the text is light on resources, so the shell uses CLI, and hence, commands are kept very terse to reduce the number.

Commands in Terminal: Before, from now on, ensure you've gone through the installation process, as we assume you know how to open the terminal. When we open a terminal, it provides us a shell where we write a command, run it(pressing "Enter"), and then wait for the following command after completing the shell again. When we type a command, it gets written after "$" in the same line; this line is known as a prompt before we write anything. Also, the commands are case-sensitive, and the general form of the command is like this:

```
$ command -options aguments
```

Before we rush to list down commands(although the list is not significant for our case), we first try to understand some essential topics.

2.1 Linux Filesystem

This is the most essential topic and the beginning of almost all commands because you don't run commands in arbitrary places in the filesystem. You first decide a place where you run your command, and for that, you must know how to filesystem first.

Filesystem is a tree-like structure that organizes your memory space's files and directories(folders). Every filesystem(Linux-based) starts from a root directory denoted or symbolized as "/." Now this main **root** directory had sub-directories of home directories of different users, **bin** that contains binaries(some programs/applications that we can run), **boot** which contains files required at the time of booting(i.e., to start our system), etc. The main point is the tree-like structure like the **home** directories of different users containing directories like Desktop, Downloads, etc. The "~" refers to the current User's home directory. Now, to navigate or refer within directories, we need to specify a path from the root to that file/directory in the format:

```
/User/Desktop/DemoDirec/Demofile
```

Where the parent and child directories are separated by "/." Two virtual files are also maintained in every directory by OS:

- "**..**": This is convenient, as writing the address of the parent directory of the current directory can be unnecessary work, so instead, we can use "**..**" to refer to the parent directory. For example, suppose our current working directory is DemoDirec, and we want to refer to Desktop(parent of DemoDirec), then either we can say " **/User/Desktop**" or "**..**".
- "**.** ": This is a reference for the current directory. While it might seem confusing at first, as we can also refer to stuff inside the current directory directly by their name, we can't run

an executable inside the current directory that way due to some technical security issues. For example, suppose our current working directory is DemoDirec, and we want to refer to Demofile. Either we can say " **/User/Desktop/DemoDirec/Demofile**" or "**./Demofile**" or "**Demofile**".

When we open the terminal, the default shell opens home(which is /root for the root user, /home/username for regular users) as the current working directory. We can change it using the command **cd**, which takes the path of the target directory as argument:

```
$ cd /Users/Desktop/DemoDirec/
```

Now we can check the current working directory using the command:

```
$ pwd
```

Also, almost every shell has an in-built feature for auto-completion, using tap, which can be handy in writing commands, referring to the path of files, etc. For example, Sup DemoDirec has only Demofile; then we can type

```
$ c1 /Users/Desktop/DemoDirec/D
```

Where c1 is a command that takes a path as an argument, now if we hit the "TAP" button, the shell autocompletes it as: type

```
$ c1 /Users/Desktop/DemoDirec/Demofile
```

Note this auto-completion can be used anywhere, not just to autocomplete a path; we can even use it to auto-complete the command name. If auto-completion doesn't work, there was an ambiguity, as there was more than one possibility with the so far typed, so we can try typing more letters.

2.2 Commands related to filesystem

- ls(list): This Displays the contents of the current directory, but it can take a path(of a target directory) as an argument. There are many useful options, some of which are:
  - -a(shows hidden files also),-l(print files in a long listing format).
  - Here's the explanation of access mode, displayed in `ls -l`.

  E.g.:

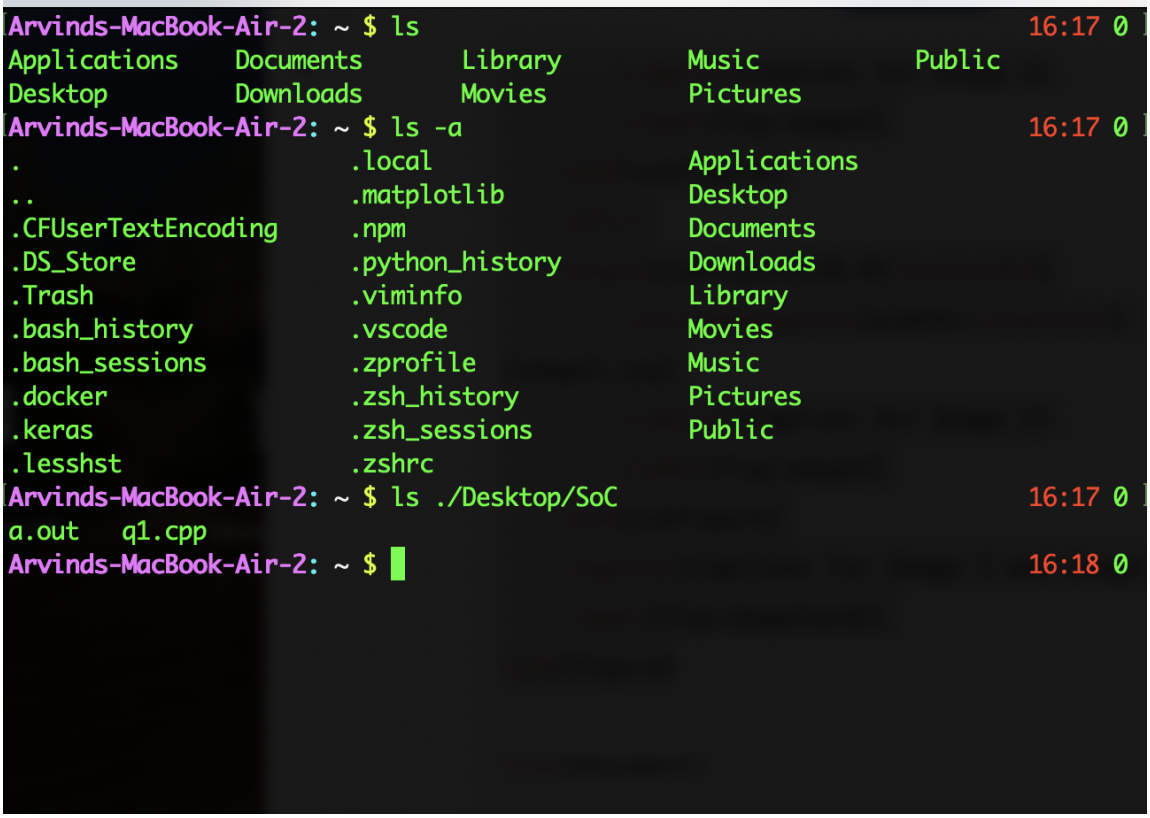


Figure 1: The first two commands optionally skip an argument(in which case the default current directory is considered an argument), and the third takes an argument "./Desktop/Soc"

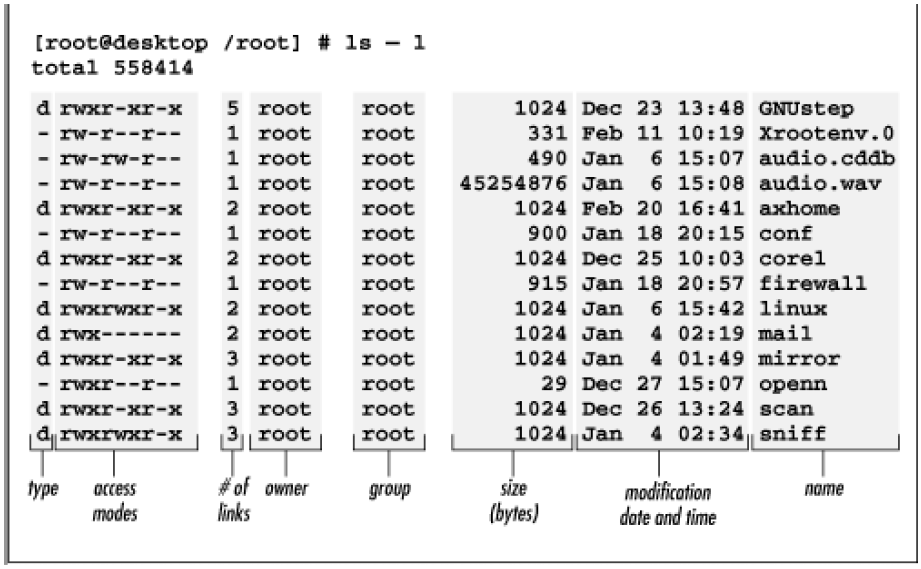


Figure 2: Different fields displayed by "ls -l"

- mkdir(make directory): This takes one or more directories (can be mentioned as a path) as arguments to create.
  - Options: -p(creates the directory only if it doesn't Exist).

  E.g.:

  ```
  $ mkdir ./Desktop/SoC ./Downloads/Yes
  ```

  This creates two directories, SoC in Desktop and Yes in Downloads.

- touch: This takes arguments as paths and creates a file with those paths.
  E.g.:

  ```
  $ touch ./Desktop/DemoDir/Demofile.txt
  ```

  This creates Demofile.txt inside DemoDir of Desktop.

- mv: This takes arguments as SOURCE and DESTINATION, where SOURCE is one or more files or directories, and moves them to DESTINATION, which must be a single file or directory. The DESTINATION must be a directory when multiple files or directories are given as a SOURCE. If you specify a single file as SOURCE, and the DESTINATION target is an existing directory, the file is moved to the specified directory. If you specify a single file as SOURCE and a single file as DESTINATION target, then you're renaming the file. When the SOURCE is a directory, and DESTINATION doesn't exist, SOURCE will be renamed DESTINATION. Otherwise, if DESTINATION exists, it is moved inside the DESTINATION directory.

- Options: -n(avoid overwriting existing files.),-i(prompt the user before overwriting an existing file),etc.

E.g.:

```
mv ./Desktop/SoC ./Desktop/DemoDire/Demofile ./Downloads
```

Note: The last argument represents the Destination. So this moves SoC of Desktop and Demofile to Downloads(verify using ls).

```
mv ./Dowloads/Demofile ./Dowloads/new.txt
```

Note: There is no command for renaming the file, but the same thing can be done using the mv command, like in the above command. Demofile is renamed to new.txt(again verify using ls).

- cp: This is similar to mv but copies the contents.
- rm: This is used to remove files, but be careful, as it doesn't move stuff to the recycle bin; files are permanently deleted. This takes arguments as a path to files or directories.
  - Options: -r(to delete multiple files or a directory).

E.g.:

```
rm -r ./Desktop/DemoDire ./Downloads/SoC
```

2.3 Regulr Expression(regex)

Regex is a pattern that matches a set of strings. It is often used in text editors, programming languages, and command-line tools. This pattern is mainly used in pattern matching with strings or string matching, i.e., "find and replace"-like operations. E.g., if we want to refer to all strings of the form `"THI...`(then something)`"`, then we can use `THIt*` where `*` means `THI` should be present then something is there(can be empty), note `'t'` can be any character, because as you will learn `THIt*` searches for `THI, THIt, THTtt, THTttt,...,` likewise. Next, we list regex, but a slight warning: "May the patience be with you, for what you face can be a long list" (don't try to mug it up; go through once and try to use it as a reference, for practice/assignments):

- `*`:
  - → This matches the preceding item zero or more times.
  - → As example, `at*` searches or matches for all the string containing `a,at,att,attt,...` likewise. Note here, by saying string A contains string B; we mean that string B is present in a string as a contiguous string in A; for example, in the latter, e.g., the strings that will be accepted are `"at", "hatt", "sadhkatt","` `jdSJaaskdjatdsgh"` etc. Now think about

the case for `"dasdkj*asdjk"`. And verify your answer later in the end when we learn to apply this stuff in the terminal.

- `+`:
  - → This matches the preceding item one or more times.
  - → So for `at+` we will look for `at,att,attt,...` like-wise. And hence the strings that are accepted here will be `"dkajatsa"`, but not `"asad"`.

- `?`:
  - → This matches the preceding item zero or one times.
  - → So for `at?` we will just look for `a` and `at`. Hence `"ffsasdds"` will be accepted, but not `"attsnjkd"`.

- {`n`}:This Matches the preceding item exactly n times.
  - → So for `at{3}`, we will just look for `attt`.

- {`n,`}:This matches the preceding item at least n times.
  - → Hence we can see that `at*` is same as `at{0,}` and `at+` is same as `at{1,}`. Therefore we can either use `*` or `+` for {`0,`} or {`1,`} respectively.

- {`,m`}:This matches the preceding item at most m times.
  - → Hence `?` can be used in place of {`,1`}.

- {`n,m`}: This matches the preceding item from n to m times.
- `( )`: This is used to group the patterns; for example, suppose we want to search for strings, like `UNIX`, `UNINIX` and `UNINIX`, then we can use {1,3} for this: Search for `U(NI){1,3}X`.

- `[ ]`: This is used to say that we can replace thing from `[ ]`, at the place of `[ ]`, so for example, `ab[xyz]c`, will search for `abxc`,`abyc` and `abzc`.

- `[^ ]`: It does opposite to that of `[ ]`, it says, that we can replace anything other than things in [^], for example, `ab[^xyz]c`, will search for anything other than `abxc`,`abyc` and `abzc`, but of the form `ab_c` where _ is a character.

- `[a-z]`: This says we can replace a letter from the small case alphabet in place of `[a-z]`, or in other words, it's the same as `[abcdefghijklmnopqrstuvwxyz]`.

- `[A-Z]`: This says we can replace a letter from the upper case alphabet in place of `[A-Z]`.

- `[0-9]`: This says we can replace a single digit in place of `[0-9]`.

- `^`: This had two meaning one if used inside `[ ]`, but we discussed it already, other is like `^dasd`, then looks for first word of every line having `dasd` at the beginning, that in other words `^` represents the beginning of a line.

- `$`: Like the previous, it searches for the end of every line with a particular ending. For example, `asda$` searches for words at the end of a line ending in `asda`.

- `.` : This also works like `[ ]`, but it says we can replace any character in place of `.` .

- \: This is a very important one; in fact, infact it is used beyond regex. Here is the complete story; sometimes, you might doubt how terminal, differentiate symbols having multiple meanings, like `$`, can be interpreted as the character `$` or the regex; for that, we use this to denote the special characters(like `$,_,#, ,...etc.` the 4th entry the space). We use this to refer to this special character as a usual character. This is called escaping a special character. For example to escape, `$`, you can refer **`\$`**.

- `|`: It will work like a or operator; that is, it searches for strings containing either the left or right side or both. For example, `unix|cmd` searches for `unix` or `cmd`.

Now, with this much regex, you do almost all of the work that is usually required. We can apply regex anywhere where searching or matching is required in strings. Now, we will discuss some of their uses and learn more commands.

2.4 echo, cat, grep, wc, tr, cut, paste, sort

In this part, we will mainly see how to use regex but not put it to real use, but the step that will be missing is small; we need to apply those changes to real things; once we are done with searching and modifying, here we discuss the tools to modify and search, latter we will see how to apply this changes mainly.
First, some general helpful commands also remember that some commands' output or working may slightly change due to internal implementation of commands by OS, but at the core, these commands a similar work:

- `man`: Did you forget any syntax or uses of command? Then type `man cmd`, which gives detailed info about the command `cmd`. These are often called man(manual) pages of that command.

- `clear, ctrl+K,ctrl+C`: You can type `clear` to clean up the shell; that is, it erases everything you've written so far; you can achieve the same effect using ctrl+K. Whenever you think or get bored because a command/program is taking too long, which you run using a terminal,

you can always hit ctrl+C to kill that program or command; we will learn more about this when we discuss Process.

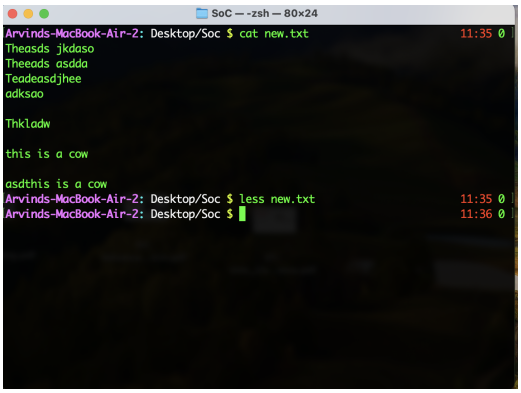
- `cat`: This is named as such because its indented purpose is to display the content of a file, or files, and concatenate to display as a single output. E.g.:

  ```
  $ cat file1.txt file2.txt file3.txt
  ```

  Options:

  - -n: to display contents of a file with line numbers.
  - -s: to omit the repeated empty output lines.

- `less`: But when the file is big, `cat` is of no good use, then you can use `less`; this command displays the content interactively by displaying some portion of the content, then you can use arrows keys up and down, page up and down, to move through your file. When you are done, type `:q`.



(a) On running `cat`, and `less`(after typing `:q`)

(b) On running `less`, now type `:q` to close it.

Figure 3: Difference between `less` and `cat`

- `wc`: This can be considered a word count command, which prints the number of lines, words, and characters in a file or files. E.g.:

  ```
  $ wc file1.txt file2.txt
  ```

  And the output is like this:

  ```
    10      15      92 file1.txt
     1       1       6 file2.txt
    11      16      98 total
  ```

  Any row's first, second, and third entry represents the number of lines, words, and characters. There are a few options like `-w` prints the word count; similarly, `-l` and `-c` are also options.

- `echo`: This displays text in the terminal. It has some options like `-n` suppresses the newline, as every echo print ends with a newline, or you can say `\n`. E.g.:

  ```
  $ echo -n aksjd
  ```

  Output:

  ```
  aksjd%
  ```

  Now try this:

  ```
  $ echo aksjd
  ```

  Output:

  ```
  aksjd
  ```

  There is also another important option `-e`, and I would suggest always keeping this because this helps to use `\n, \t` for inserting newline and tabs in the output. E.g.:

  ```
  $ echo -e "I'm having a \n\t\tlot of fun in learning \n\t\t\t\t\t terminal"
  ```

  Output:

  ```
  I'm having a
                  lot of fun in learning
                                                terminal
  ```

- `grep`: This is where regex comes into the picture, as this command searches for character patterns in a file or files by taking the pattern and file name as arguments. E.g.:

  ```
  $ grep "UNIX" new.txt
  ```

  It searches the pattern we described earlier, i.e., it checks if a string contains that pattern, but when it gets a match, it prints the whole line. That's why I would suggest always to use the options `--color=auto`, E.g.:

  ```
  $ grep --color=auto "UNIX" new.txt
  ```

  It will again print the whole line, highlighting the matched pattern with different colors. Now get ready, create a file with the name new.txt and fill some text in it, run and observe the following things:

  → To use regex, you need to escape the special character. Why? Because, see `grep` takes pattern as string argument and interprets it, we need to tell the terminal that this special character must treated as string and to be passed to grep(it's executable/program). Hence for example, if you want to search for `at+`, you need to write:

```
$ grep --color=auto "at\+" new.txt
```

OR

```
$ grep --color=auto "at\{1,\}" new.txt
```

Also, one common misconception is that `*` is a special character. No it's not, so for searching `at*`, you can write:

```
$ grep --color=auto "at*" new.txt
```

→ Options:

◇ `-w`: This matches only whole words, for example:

```
$ grep --color=auto -w "at" new.txt
```

This will search for word `at`, hence strings like `sadat`, `kkkatkk`, and `atkk` etc.
Question: Can you write an equivalent of this from basics regex?
Answer: Yes, first make cases so that a word can be at the beginning of the line, in the end, or the middle, then convert each of these cases in regex. The individual cases can be huge, but remember `-w` doesn't cover all cases, like `?well?asd` is accepted for `well` using `-w` option. For example, we are searching for `well`, let's take the case of when the word is in the middle; we use the regex `well` ; let us consider the case where the word in at the end, that is `well` will be either followed by one of this `? , . !`, then for this, we can use the regex `well[?,.!]`, but it's partially correct because `well?adsa` will get accepted, but this is also accepted by `-w`. However, still, there are more corner cases, let it be, so the final regex for the combined end and the middle case is: `well | well[?,.!]`, and the code will look like this (note be careful what to escape, or just hit and trial it.):

```
$  grep --color=auto " well \| well[?.\!]" new.txt
```

◇ `-i`: This ignores the upper and lower case difference.

◇ `-v`: This inverts the sense of matching to select non-matching lines.

◇ Others: There are many other options, but this is sufficient, and you can always reference online tools for getting options.

There is much to play with `grep`, but let's move to some modifying tools.

- `tr`: This changes a set of characters to another set of characters, but always remember that this only applies to characters. To use it, and because `tr` has only two arguments, there are two options to give input, like a text or a file, that can be achieved either by redirection or pipeline, which we will cover in the next subsection, so for example, we will say/use `tr [OPTIONS] SET1 [SET2] < file.txt` where `<`, stands for redirection, which for now, you can take it as if file.txt in being given input to `tr`. E.g.:

```
$ tr "[a-z]" "[A-Z]" < file.txt
```

In the above command, our first set is `[a-z]` and the second set is `[A-Z]`, since `tr` replaces/translates the character of the first set to the second set, therefore, so the output will contain the upper case version of file.txt. Again, we will emphasize the fact that `tr` goes character-wise translation, so if we try the following command:

```
$ tr "asd" "kaj" < file.txt
```

Then, contrary to what we expected, a replace of all occurrences of `asd` to `kaj` will not happen, instead `tr` draws a mapping, and it will translate/replace `a` with `k`, `s` with `a` and `d` with `j`. The main use of `tr` is with its options:

→ `-d`: When using this option, we don't need to mention the second set, as this option indicates to delete every occurrence of characters mentioned in the first. Suppose file.txt had just one line: `This aj nob duh.`

```
$ tr -d "[abd]" < file.txt
```

then output will be `This j no uh.`

→ `-s`: This most loved option, as it squeezes the continuous appearance of a character to one, like **aseee** on applying `tr -s "[e]" "[k]"` becomes **ask**. We can optionally skip the second set here if you don't want to change the character.

→ `-c`: This is just to complement the effect of the first set; suppose the set we want to be the first set is larger than its complement. Then, we can mention its complement as the first set and put this option. E.g.:

```
$ tr -c -d "[asd\n]" < file.txt
```

Or

```
$ tr -cd "[asd\n]" < file.txt
```

The output of the above commands is file.txt by deleting every character other than `asd\n`, note that newline `\n`, is a character to represent a newline. Try excluding `\n` and compare the output.

- `cut`: Uhmm We have reached so far, and interesting things are starting to come up and adding to the list. `cut` is here a useful and powerful tool, easily handy. This takes a file or files as arguments, and depending upon the options it returns, some sections from each line of a file or files. We will directly head towards the options because they are the core of this command:

  ◇ `-c`: Consider the following E.g.:

  ```
  $ cut -c 1,10 file.txt
  ```

  It will print the first and tenth characters in every line of file.txt. If we'd have used `1-10` instead of `1,10`, it would have printed all characters from the first to the tenth character in every line of file.txt.

- ◇ `-f,-d`: The `-f` specifies the fields numbers to be printed, much like `-c`, but fields are separated by what is called delimiters; these are some separators between different fields, and `-d` is used to set delimiter, which by default set to one `"TAB"`.

  ```
  $ cut -f 1-3 -d'' file.txt
  ```

  In the above example, the output is fields from first to third, separated by space, in each line of file.txt.

- ◇ `-c`: Much like `-c` of the `tr`, it has a complementing effect like `cut -f 1,2 -c -d'' file.txt`, will print every other field separated by spaces, other than first and second.

- `paste`: The general use to merge files into one output; however, the uses are particular, so we list some frequent ways in which we use `paste`:

  - ◇
    ```
    $ paste file1.txt file2.txt file3.txt
    ```

    This output is achieved by combining the lines of three files side by side.

  - ◇
    ```
    $ paste -s file1.txt file2.txt
    ```

    This output is achieved by combining the lines of two files vertically instead of side by side.

  - ◇
    ```
    $ paste -d ",;" file1.txt file2.txt file3.txt
    ```

    Here, it just does everything as such in the first example, except we change the default separator from `"TAB"` to `,` for the first separation(between file1.txt and file2.txt lines) and from `"TAB"` to `;` for second separation(between file2.txt and file3.txt lines) during the output.

  - ◇
    ```
    $ paste -d "," - - < file.txt
    ```

    This will combine each pair of lines(the first and second lines form one pair; similarly, the third and fourth lines form the second pair, which goes like this) from file.txt into a single line, separated by a comma. (ignore the redirection `<`)

- `sort`: This is an effortless and chill command; it just sorts the lines of files alphabetically by default, and with options, we can also sort numerically.E.g.:

  - ◇
    ```
    $ sort file1.txt file2.txt
    ```

    This sorts the lines alphabetically by considering the lines from both files.

  - ◇
    ```
    $ sort -n file1.txt file2.txt
    ```

    It does the same thing as above, but just sort numerically.

  - ◇
    ```
    $ sort -r file1.txt file2.txt
    ```

    Sort the lines alphabetically in reverse order by considering the lines from both files.

◇ `$ sort -t " " -k 1,2 file.txt`

It sorts the alphabetically, using the field first and then second(if there is a tie on the first field) of the lines, where the delimiter for the field is set from `"TAB"` to `"SPACE"`.

2.5 Sudo, Redirection, Pipes, Concurrent and Sequential Executions

`sudo` It was introduced for security reasons. It allows us to switch to another user and run a command, and after that, we are back again in the earlier User. Although we don't have to worry about `sudo` being used to switch to any use(`-u` options is used for this)r, we almost always use `sudo`, to change to superuser(who can do anything, has all rights on files). The syntax is:

```
$ sudo ls   (replace ls with your command)
```

Redirection First, we must understand what a stream is. Stream is a special file that continuously receives or pushes text in. Whenever a process starts, that Process is given access to three "standard" streams

- Standard input (stdin; fd is 0)
- Standard output (stdout, fd is 1)
- Standard error (stderr, fd is 2); used when an error has occurred

In the context of command-line interfaces, fd stands for "file descriptor." A file descriptor is a non-negative integer representing an open file or stream within a process. File descriptors are used to perform input/output operations on files and streams. In Linux and other Unix-like operating systems, there are three standard file descriptors: stdin (standard input), stdout (standard output), and stderr (standard error). These file descriptors have 0, 1, and 2 values, respectively. So when you see fd=1 in a command line, it refers to the standard output stream (stdout)(Note: this is the only default behavior; we can change this, and roughly this is what redirection means).

- `>`: Output of a command redirected to a file, and for this we can use either of these:

  ```
  $ cmd > file.txt
  ```

  Or

  ```
  $ cmd 1>file.txt   (1 if fd for stdout)
  ```

  But this always overwrites the existing files.

- `>>`: same as above but append instead of replace.
- `<`: This makes the stdin of a command read from a file.

Pipe Pipe(denoted by `|`) does nothing more, but it takes output from one command and feeds it as input to another command. E.g.:

```
$ echo -e "this is a demo" | grep --color=auto" is"
```

Output:

```
this is a demo              (is will be colored differently)
```

Concurrent and Sequential Executions It is very clear from the naming that concurrent execution means that we can run multiple commands simultaneously on a single enter. In contrast, in sequential execution, they run one by one in a single enter hit. E.g.:

```
$ cmd1 && cmd2 && cmd3
```

This runs `cmd1,cmd2,cmd3` sequentially, so to run sequentially, we need to add `&&` in between the commands.

```
$ cmd1 & cmd2 & cmd3
```

This runs `cmd1,cmd2,cmd3` concurrently, so to run concurrently, we need to add `&` in between the commands.

2.6 Processes

This will be discussed in more detail when we study the theory for OS, but here, we will discuss a little to see the tools that can be used when we study the theory part. We begin with some concepts and then finish with useful commands to get the properties of those concepts.

First comes, is what we mean by Process. Although the answer can be long, in short, we can say, "A process is a program in execution." An application runs multiple processes on CPU(or CPUs) for different tasks, like when we run the terminal, it creates a process for the shell that we use to type and run the commands. Still, when we run a command after hitting `"ENTER"`, the shell creates another process that runs the command. The shell is called the parent of the Process that it spawns, and this Process is called a child of the shell process. Likewise, if we trace the parent-child relations, we get a hierarchy of processes, a tree-like structure, where every node represents a process, and the parent of a node is the parent of the Process represented by that node. Like, if think, who was the parent in the case of the shell when you opened the terminal? Well, there is someone who is a parent, but till the OS gives us the GUI, it has spawned many processes, and the story is too long to trace back, but we can trace it from the root.

Every Process has a unique ID called a process ID. The first Process that runs after boot up is called the Init process, and it is the ancestor of all processes with process ID 1, and it's the root of the tree structure. It then spawns a process called a shell, which the User uses to interact with the OS; now, the shell process(and its children) will be used to get the usual GUI. Now, every Process also has what is called the state of the Process, which represents whether the Process is running(Note: A process means a program that is executed, so, despite it being executed, it might not be running because CPU, schedules the Process to manage its resources better), sleeping(when

CPU kicks a process to sleep, that is it will ask the CPU, again for permission to run again only after a while), runnable(when the Process is waiting to be scheduled by CPU), zombie(when the Process had exited, now wait for its parent to clean it, we will learn why it needs to wait for the parent to clean it). Note: Different OSs use different terminologies, but these are the central states. Now, processes have some memory occupied on RAM(called physical memory) and some virtual memory on a Hard drive. However you will learn that the physical memory is also virtual because the OS gives some illusion to every Process. It's okay if you don't understand the previous line. Eventually, you will know as we move forward. So, okay, let's finish this chapter with some more commands:

- `ps`: This is used to get the info about the Process, like process ID(`PID`), parent ID(`PPID`), physical memory(`RSS`), virtual memory(`VSZ`), to get the percentage of the Process's resident set size(`%MEM`), state of the Process (`STAT`). This can be obtained using `ps aux` command, but it lists all the processes. Therefore we prefer to use `ps -f` or `ps -j` to get PID, then use `grep` with the pipeline to `ps aux`, like:

  ```
  $ ps aux | grep "1213\|RSS"
  ```

  Where the number you see is the PID of the desired Process, and we have used `|RSS` to get the first rows also, which tells which column is what(try to run `grep` just for PID, and you will set the first row is missing to guide us). To the physical memory on the machine.

Next, we discuss the concepts of foreground and background processes, and there is a simple way to describe them. Did you recall that whenever you run/hit `"ENTER"`, you can't write another command until the previous one finishes? To get a better view of this, use a command that creates a process(every command runs on the child process of shell), which sleeps for 10s:

```
$ sleep 10
```

The shell doesn't prompt us to write another command for 10s; this is called foreground running of a command(or foreground process), where the shell waits for the command to finish, whereas, in background running of a command, the shell doesn't wait to finish the command, it immediately finishes prompt use for another command and we can write and run another command, background finishes in the background, and its output is displayed(if any), whenever it's ready(this behavior may differ for different OS). You can run any command in the background using `&` after the command, but note that this also mentions concurrent runs; hence, we can't run multiple commands in the background at once. We run just one command in the background, which has `&` in the end and nothing else, like:

```
$ cmd1 & cmd2 & cmd3 &
```

Then only **cmd3** will run in the background, but they all will run concurrently. Try using:

```
$ sleep 10 &
```

You will be able to write the next command immediately, and after 10s, you will see:

```
[1]  + done       sleep 10
```

- `jobs`: jobs is another command, a shell builtin that tells you about the jobs the current shell manages. The important part is that it tells only about the current shell; you can check using the following:

  ```
  $ sleep 15 &
  ```

  Then open a new tap/shell, and run `jobs`, you will see nothing, but when you will run `jobs` in original shell, you will see:

  ```
  [1]  + running    sleep 10
  ```

  But the output is the same when you use `ps`.

- `kill`: This can be used to terminate a process if it hangs and not responding, using

  ```
  kill -9 32212
  ```

  where `32212` is the PID of the Process we want to kill. This is a more general method than `ctrl+C`, which can only terminate the foreground process.

- `Cron Jobs`: These schedule tasks to run automatically on Unix-like operating systems at specified intervals. A process called cron daemon(crond) runs in the background, and the task is to check and execute the schedule(which is called a crontab) for tasks. We can open the crontab and add or remove an entry using the following commands:

  - `$ crontab -l`

    This lists all the current User's cron jobs.

  - `$ crontab -r`

    This removes all cron jobs for the current User.

  - `$ crontab -e`

    This opens the crontab in an editor(which you can set, using this video). By default, the editor will be Vim. For VVim, here is a quick guide: to edit the file, first hit **i** to enter the editing mode, then once done, hit **esc** to go back to read mode, then type **:wq** to write and quite. Or you run `export EDITOR=nano` before `crontab -e`. Then, you can add or remove entries. But first, understand how the entries are defined:

    ```
    # |------------- Minute (0 - 59)
    # |  |---------- Hour (0 - 23)
    # |  |  |-------- Day of the month (1 - 31)
    # |  |  |  |---- Month (1 - 12)
    # |  |  |  |  |- Day of the week (0 - 7) (Sunday is both 0 and 7)
    # |  |  |  |  |
    # *  *  *  *  *  command_to_be_executed
    ```

For example, some you want to print, `hello demons`, every 2 minutes, then the entry will look like:

```
*/2 * * * * echo -e "hello\! demons"
```

Note: `*/2` specifies to run every 2 minutes; if we wrote just `2` instead, it will write at the 2nd minute of every hour, not every 2 minutes. (in case of mac). You won't see the changes, but when you open the terminal, you see

```
you have 1 mail.
```

Then, you can use the command `mail` to see it, in interactive mode, and type `p,d` to read the next mail and delete the current mail, respectively, when you type `p` to read the mail you will see `hello demons` as the message in end, then type `quit` to exit. You use this command to check/verify its correctness:

```
log show --predicate 'process == "cron"' --info
```

You can run any command, but you need to set the permission and handle the errors carefully, like printing something to file in User's files using `crontab` is difficult for Mac users due to security issues.Here is cool site for those who are interested more in this.

3 Assignment

There are more Command-line tools, but we stop here only; before the assignment is released for this week, you can try playing with these commands or learning more about them(like crontab). The assignment is expected to be released by Friday, with enough time for submission. Till then,

```
$ waste your time | --in -a good summer > chill
```