Shell

`shell` refers to a **command-line interpreter** (like a program) that provides a user interface for accessing an operating system's services.

In Unix and Linux, shells are the primary way for users to interact with the system via commands and scripts.

There are several types of shells, such as the **Bourne shell (sh)**, C shell (csh), Korn shell (ksh), and others.

Operating Systems

An operating system (OS) is a software program that acts as an intermediary between computer hardware and the user, as well as between other software applications.

`Hardware` is the physical component of a computer system, including the CPU, GPU, RAM, storage devices, etc.

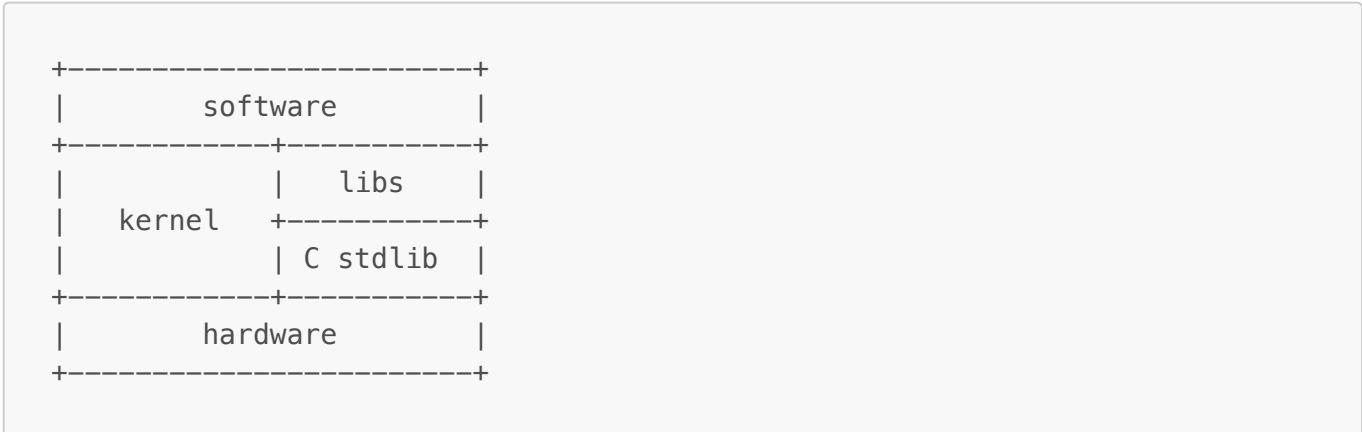
`Kernel` is the core component of the OS, acting as an intermediary between the harware and the software applications that run on it. It provides low-level services to the parts of the system and applications, and it controls all hardware interactions.

`Operating System` built around the kernel, includes additional components like the user interface (UI) and utility programs, providing a user-friendly environment and higher-level functionalities.

`Software` includes application software and system software. Application software consists of programs that perform specific tasks for users (e.g. word processors, web browsers). The operating system itself is considered system software, which provides a platform for application software to run.

- Ubuntu is an example of a Linux distro (distribution).
- Debian is the upstream distro for Ubuntu from which it inherits thousands of *packages*. You can can check the packages with `dpkg -l`.
- Linux is an OS **kernel**.

Software levels of abstraction



sh and Emacs are applications in themselves, one of many **applications** that sit on top of the operating system.

Process ID (PID)

`PID` is a unique numerical identifier assigned by the OS to each process when it is created. This identifier allows the OS and users to manage and interact with specific processes, such as terminating a process, monitoring its resource usage, or tracing its activity.

Access and Trace a PID

View Processes: `ps` reports a snapshot(简况) of current processes.
To see all running processes, use `ps -e`.
To display with a full-format listing (includes UID PID PPID(Parent PID)), use `ps -f`
To display the process hierarchy like a tree-like format, use `ps -H`

Find a Process by Name: `pgrep` finds the PID of a process by name. For example, `pgrep nginx` finds the PID(s) of nginx processes.

Kill a Process: `kill PID` terminates a process. If the process does not terminate, `kill -9 PID` forces termination.

Superuser and Concept of Privilege

- **Superuser** (root) are the only ones with permission to `kill` PID 1, `system`.
- The `sudo` command lets you run a command as "root":

```
sudo sh
```

> **Why have multiple users instead of just root?**
>
> The concept of **minimization of privileges** aka "principle of least privilege". If a program breaks or a user makes a mistake, the damage is limited.

> **Use only the `ps` command to find your own login shell's process, all that process's ancestors, and all its descendants.**
>
> **1. Find Shell's Process ID**: First, we need to find the PID of current shell session, using the command of `echo $$` or `ps $$`.
>
> **2. Get Ancestors**: To get all ancestors of the shell process, we need to trace the parent process IDs (PPIDs) recursively until we reach the initial process (PID 1). We can do this by repeatedly using `ps -f PID#` to find the PPID of each process, starting with the shell's PID. (Or scroll the screen to find)
>
> **3. Get Descendants**: We need to list all processes, let the PID of this process be the PPID of its descendant, and recursively until we reach the numerically largest PID.

The `(ba)sh` Program

A shell is itself a program, and programs themselves are just files that can be executed by the operating system.

sh is the predecessor to **bash** (Bourne Again SH). sh was designed to work on 16-bit machines so it's a very little language. Bash adds some features in addition to the original sh.

There is also a lot of other shell languages ending in sh. Having so many distinct shell languages becomes a problem, so the **POSIX standard** was created as a spec for shells.

Creating another instance of the shell from within the shell itself to execute a one-off command:

```
sh -c <command>
```

This is what you call a **subprocess**, or a **child process**. Within your running instance of `sh` (the CLI you're typing into), another *instance* of `sh` is spawned as a child of that `sh`. In fact, every time you run a command, an instance of their little program is attached to your shell as a child process. You can see this for yourself when using commands like `ps` to show processes and their parentage.

ASIDE: Some Mischievous Things

Things you can do to annoy your system administrator:

Telling the shell to go into an infinite loop:

```
sh -c 'while true; do true; done'
```

A no-op instruction(无操作指令)(休眠) for a number of seconds, doesn't use CPU

```
sleep 10
```

The `truncate` command sets a certain file to a certain size, ending at the size you specify (filling with null data if the size expands I presume):

```
truncate --size=10TB bigfile
```

If you inspect the filesystem, you'll find that you didn't actually use up that much space, just convince the directory listings that much space had been allocated for *something*. This is still annoying though because it will trip up the sysadmins when they perform **backups**.

Baisc Commands in Shell

0. Find help and Manual

- `$ command_name --help` - Displays the help documentation (for a quick overview)
- `$ man command_name` - Display the manual pages (for a comprehensive understanding) (including all options, nuances, and examples)

- `$ info command_name` - Similar to `$ man` above, whereas it is particularly useful for **GNU tools and programs**

1. Navigating and Manipulating the File System

- `~` - Home directory
- `$ cd` - Changes the current directory
    - `$ cd ..` - moves one directory up in the hierarchy
- `^D` - Exit the current shell session
- `$ pwd` - Print the current working directory path (absolute path)
- `$ ls` - List **files** and **directories** in the current directory
    - `-a` (show all files, including hidden)
    - `-l` (show files in long listing format) (more details)
    - `-i` (display the inode number of each file)
    - `-d` (show the current directory) (only display directory name without listing the contents)
    - `-t` (sort by modification time) (the newest(lastest modified) files appear at the top of the list)
    - `-h` (show human-readable sizes)
    - `-r` (reverse order)
    - `-R` (recursively list) (list the content inside all directories and their subdirectories)
- `$ mv` - Move files or directories from current location to another **OR** Rename a file (cheap)
    - `$ mv file1 file2... PATH` (Moving)
    - `$ mv oldname.txt newname.txt` (Renaming)
- `$ cp` - Copy files and directories (expensive)
    - `-r` (recursive copying when dealing with directories) (copy every item within the directory, including its sub-directories and all their contents)
- `$ rm` - Remove(delete) files
    - `-r` (Remove **recursively**) (Delete a directory and **all of its contents**, including all files and sub-directories)
    - `$ rmdir` - Remove directories (Safer than `$ rm -r`, will not accidentally delete files)
- `$ mkdir directory_name` - Create a new directory

- `$ touch file_name` - Create a new file if it does not exist **OR** Updates the timestamp of an existing file
- `$ make target_name` - Used to build and compile a software program. It reads a file named `Makefile`, which the `target_name` points to, contains a set of directives(not code itself) about how to compile and link the program(code).

- `MakeFile` is the file pointed by `target` containing instructions for `make` on how to build the target program.

> **Consider a directory containing only 'Makefile' and 'prog.c'. The Makefile contains a target 'prog' that compiles 'prog.c' to 'prog'.**
> **We run the following sequence of shell commands:**
>
> ```
> $ make prog
> $ touch prog.c
> $ touch prog
> $ make prog
> ```
>
> **How many times is 'prog.c' compiled?**
>
> Once.
> `make` checks if the modification time of 'prog' newer than 'prog.c' to determine whether the file needs compilation.
> `touch` updates the timestamps in a way such that 'prog' is newer than 'prog.c'.

> `prog.c` is the **C Source Code File** where the programmer writes the code.
> `prog` is the **Executable File** generated from compiling prog.c. It is also a **target** for `make` in this case.
> The **Executable File** contains machine code that the computer can execute directly to perform the tasks defined in the source code.

> `$ make prog` compiles 'prog.c' into the executable 'prog'. At this point, 'prog.c' has been compiled once.
> `$ touch prog.c` updates the modification time of 'prog.c' to the current time, making it newer than the previously compiled 'prog' executable.
> `$ touch prog` updates the modification time of the 'prog' executable to the current time, making it newer than 'prog.c' again.
> `$ make prog` Since 'prog' already exists, `make` checks its timestamps first. Since 'prog' was `touch`ed after 'prog.c', making its modification time newer than that of 'prog.c', `make` might normally decide that 'prog' does not need to be recompiled because the **target(prog)** is newer than its dependency(prog.c).

2. Working with File Contents

- `$ cat file_name` - Display the content of a file
- `$ cat file1 >> file2` - **Append** the content of file1 to file2
- `$ cat file1 file2 > file3` - Concatenate file1 and file2, storing the result into file3 (if file 3 already exists, then **overwrite** it with new contents)
- `$ diff file1 file2` - Compare files line by line, and output their difference in the "normal" format
  - `$ diff -u file1 file2` (use the **unified format** for its output)

> `$ diff -u file1.txt file2.txt >result.diff`
> compares the difference between two file in unified format and as a stdout redirect to a diff file

- `$ sed [option] 'PATTERN'` (Stream Editor) - Modify files automatically or extracting data based on patterns. Combine with **REGEX**(regular expression) is helpful.

  Specific Operations in 'PATTERN':
  `s` - substitute
  `d` - delete
  `p` - print

  `-n` - Suppress automatic printing of pattern, used with `-p` to print specific patterns
  `sed` processes each line of input, applies all the specified commands to it, and then prints the pattern by default. `-n` changes this behavior so that it does not automatically print each line after processing. Instead, lines will only be printed if explicitly instructed to do so, typically using the `p`.

```
$ sed 's/hello/world/' file.txt
# `s` specifies substitution
# Replace "hello" with "world" in a file

$ sed '/^$/d' file.txt
# Delete all empty lines from a file
# `^$` matches start and end of the line with nothing between them
# `g` deletes all lines match with pattern

$ sed -n '/pattern/p' file.txt
# Only lines matching pattern are printed to stdout. All other lines
are suppressed and not shown.
# `-n` with `p` prints lines matching "pattern"
$ sed '/pattern/' file.txt
# Every line of the input file is printed to stdout. This command
behaves similar to ($ sed '/pattern/' file.txt) because there's no
Specific Operation in the PATTERN

$ sed '/pattern/a\New line here' file.txt
# Append a line after matching "pattern"
# `a\` stands for append

$ sed '/pattern/i\New line before' file.txt
# Insert a line before matching "pattern"
# `i\` stands for insert

$ sed '3,5d' file.txt
# Remove lines from 3 to 5
```

3. Output Manipulation

Commands below often work with `ls`, `grep`, `find`, and **pipe** `|`, which takes the output of previous command, and pipe it as the input to the latter command.

- `$ wc` - word counts
  - `-c` (count bytes)
  - `-m` (count characters)
  - `-l` (count lines)
  - `-w` (count words)
- `head -n number` - Output only the first `number` lines of the input text
- `tail -n number`: Output only the last `number` lines of the input text

```
$ ls -lt /path/to/directory | head -n 10
```

> This command lists the top 10 most recently modified files or directories in /path/to/directory, sorted by modification time (the newest first)

4. Searching Files

- `$ grep [option] PATTERN file_name` - Search for patterns within files
  - `-E '^[regular expression]$'` (enable the use of **extended Regular Expressions**) (Regex starts with `^`, and ends with `$`)
  - `-v` (give the complement of search) (displays all lines that **NOT** contain the specified pattern)

    > `$ grep -v 'banana' example.txt`
    > filters out the line containing "banana", and display every other lines

  - `-r` (recursively read all files under all directories and their sub-directories)
  - `-i` (ignore case) (make the search case-insensitive)
- `$ find PATH [option1] [option2] [...]` - Search for files or directories in a directory hierarchy
  - `-type f` (search only for files)
  - `-type d` (search only for directories)
  - `-inum inode#` (search by inode #)
  - `-name file_name` (search for files by name)
    - `-iname` (case-insensitively search for files by name)
  - `-mtime number` - Find files in the current directory via **last modification time** in *days* (modified **`number` days ago**)
    - `-mmin number` - Find files in the current directory via **last modification time** in *minutes*
    - `find PATH -mtime -7` (find files accessed less than 7 days ago)
    - `find PATH -mtime +7` (find files accessed more than 7 days ago)
  - `-atime number` - Find files in the current directory via **last access time** in *days*

        - `-amin number` - Find files in the current directory via **last access time** in *minutes*

    - `-ctime number` - Find files based on the time their **metadata(or inode information)** was changed in *days*

        - `-cmin number` - Find files based on the time their **metadata(or inode information)** was changed in *minutes*

    - `-size` (search for files by size)
        - `-size -10K` (search for files under 10 KB)
        - `-size +50M` (search for files over 50 MB)
    - `-exec command_name` (execute a command on the files found)
    - `-maxdepth / -mindepth number` (limit the search to a specific depth in a directory hierarchy)
    - `-delete` (delete the files found (use with caution))
    - `-print` (print all outputs the `find` found) (this is a default behavior of `find`, so you don't need to tpye out while using)

```
$ grep -E '^[^e]+$' /usr/share/dict/linux.words | wc -l
```

> This command searches through the file /usr/share/dict/linux.words, which typically contains a list of English words, to find and count all lines (words) that do not contain the letter 'e'.

> `grep` is the command used to search for patterns within files
>
> `-E` enables **extended regular expression** syntax, allowing more complex patterns
>
> `'^[^e]+$'` is an **extended regular expression**:
>
> - `^` asserts the start of a line
> - `[^e]+` matches one or more characters (+) that are not (^ when used inside []) the letter 'e'
> - `$` asserts the end of a line
>
> `|` pipe(pass) the output of the previous command as input to the next command
>
> `wc -l` stands for **word count**, and the `-l` option tells it to count lines

```
$ find . -name "*.txt" -mtime -7 -exec cat {} \;
```

> This command searches for and displays the contents of all .txt files in the current directory and its subdirectories that were modified in the last 7 days.

> `find` is the command used to search for files in a directory hierarchy

> `.` specifies the starting PATH for the search, which is the current directory (denoted by .)
>
> `-name "*.txt"` tells `find` to match all files whose names end with `.txt`.
>
> `-mtime -7`
>
> - `mtime` is the **modification time** of the files
> - `-7` specifies files modified **less than 7 days ago**
>
> `-exec cat {} \;`
>
> - `-exec command_name` tells `find` to execute a command on each file found
> - `cat` displays the contents of files (should be used with `{}`)
> - `{}` a placeholder for the file name found by `find` (should be used with `cat`)
> - `\;` marks the end of the command to be executed (where `\` is the **escape character**, which can also be achieved by `''` or `""` like `';'` or `";"`)

> **What's the main difference between `ls` and `find`?**
>
> `ls` lists the files only in the **current directory** by default (unless with the recursive option `-R`)
>
> `find` searches for files in a **directory hierarchy** (which includes all direcotries and their subdirectories)

5. File Compression and Archiving

- `$ tar`
  - `-c` (create a new archive)
  - `-z` (**compress or decompress** the archive using 'gzip')
  - `-f` (specify the filename of the archive)
  - `-x` (extract files from an archive)
  - `-t` (list all contents of an archive)
  - `-v` (Show details about the operation(`tar` and its options) being performed) (v stands for Verbose mode)
- `$ tar -czf tgz_file_name file1 file2...` - Create a compressed archive named `tgz_file_name` using gzip and put `file1 file2...` into it
- `$ tar -tvf tgz_file_name` - Give a verbose listing of the files contained in the archive named tgz_file_name by showing detailed information about each file without actually extracting them
- `$ tar -xzf tgz_file_name` - Extract files from a compressed archive using gzip

```
tar -czv -f filename a b c
tar -czv -f=filename a b c
tar -czv --file=filename a b c
# these three commands do the same thing
```

6. Miscellaneous Commands (some others)

- `wget2 [option] [URL]` - Download content from a **URL**
  - `-0 file_name` (specify the output filename where the downloaded content stored)

    > `wget2 -O index.html [URL]`
    > downloads the content from [URL] and stores it to a file called index.html

- `$ tree` - Display the directory structure in a tree-like format
- `$ seq` - Generate a sequence of numbers (sequence)
  - `$ seq 5` (generate numbers from 1 to 5)
  - `$ seq 2 10` (generate numbers from 2 to 10)
  - `$ seq 0 2 10` (generate numbers from 0 to 10, increment by 2)
- `$ which command_name` - Show the full path of the command's executable (find where it is located)
- `$ ln source_file link_name` - Create hard links between files
  - `-s` (use -s if you want to create a symlink)
- `$ shred file_name` - overwrite a file to erase its contents (wear out the actual data on the drive)

- `$ ps` - displays only the processes running in the current shell (associated with the terminal session currently using)
  - `-e` (display every process running around the whole system)
  - `-f` (a full-format listing) (includes UID PID PPID)
    - `ps -f PID#` (show information of a particular process via its PID#)
  - `-H` (displays the process hierarchy like a tree-like format)
- `$ kill PID#` - terminate a process

- `$ shuf` - Generate random permutations of input lines (shuffle)
  - `-e arg1 arg2... --echo` (treat each command-line argument as an input line, and output with a random permutation)
  - `-n number --head-count` (display any n random lines **from the input line or file**)
  - `-i LO-HI --input-range` (output a random permutation of the numbers from LO to HI)
  - `-r --repeat`

    ```
    $ shuf -n 2 -e apple banana cherry

    $ shuf -i 1-100 -n 5
    ```

```
$ shuf -r -n 5 -e apple banana cherry
```

- `$ chmod` - change mode(permission)
  - `$ chmod [who][+/-][permission flag] file_name` - Give permission for **ugo** to a file (allowing it to be **rwxst**)
    - `who` (ugo): user, group, others
    - `permission flag` (rwxst): read, write, execute, setuid/setgid, sticky(restrict deletion)
  - `$ chmod [permission number] file_name` - The **Permission Number** consists of *three digits*, representing *ugo* respectively; The **Size** of each number represents their own *permission*.

p.s. check `Filesystem.md` to get more information about `chmod` and `permission flag`

File Expansion

1. Home Directory (‘~’)

When used alone, ~ expands to the current user's home directory

```
cd ~
# Changes the directory to the user's home directory.
```

2. Specific User’s Home Directory ('~ username')

```
cd ~ john
# Changes the directory to John's home directory.
```

3. Relative Paths (‘~/’)

When followed by a slash, ~ expands to the current user's home, directory and then appends the specified path

```
ls ~/Desktop
# Lists the contents of the Desktop directory in the user's home
directory.
```

Field Splitting

Field splitting is the process where the shell splits the results of expansions (like parameter expansion, arithmetic expansion, and command substitution) into words.
Field Spliiting is similar to str.split() in Python.

- This process is guided by the value of the IFS (Internal Field Separator) variable (内部字段分隔符)
- IFS determines the characters used as word delimiters (分隔符) during field splitting. (IFS决定字段分割时用作单词分隔符的字符)
- Its default value includes space, tab, and newline, which means that by default, words are separated by these characters
- We can change the value of IFS to control how the shell splits the results of expansions. For instance, setting IFS to a colon (😃 will make the shell use colons as delimiters (分隔符).

> Example 1: Changing IFS
> Here, if you echo $PATH after setting IFS to :, the shell will treat each entry in PATH as a separate word, splitting the string at every colon.

```
IFS=':'  # Setting IFS to colon
echo $PATH
```

> Example 2: Saving Original IFS
> It's a common and good practice to save the original value of IFS before changing it, and restore it afterward.

```
old_IFS=$IFS        # Save the original IFS
IFS=':'             # Change IFS
# ... operations with custom IFS ...
IFS=$old_IFS        # Restore original IFS
```

Pathname Expansion (Globbing)

Pathname expansion allows you to use wildcard (通配符) characters to match filenames and paths, which is extremely useful for operating on multiple files or directories with similar names or patterns.

Key Wildcards in Globbing:

- * - matching any string of characters

```
ls *.txt
# Lists all files ending with .txt in the current directory
```

- ? - matching any single character

```
ls file?.txt
# Matches file1.txt, file2.txt, etc., but not file10.txt

ls file??.txt
# Matches file10.txt, file11.txt, etc., but not file1.txt, or
file.100.txt
```

- `[...]` - matching any one of the enclosed characters. A range of characters can be specified with a hyphen(-) (连字符)

```
ls file[1-3].txt
# Matches file1.txt, file2.txt, and file3.txt

ls file[!1-3].txt # ! means 'NOT'
# Matches file4.txt, but not file1-3.txt
```

Redirection

Redirection in the shell allows us to control where the input and output of commands go. You can redirect the output of a command to a file, or use a file as the input for a command.

File Descriptors inlcudes `0`(stdin), `1`(stdout), `2`(stderr), which are small, system-allocated integers used by the shell to handle inputs and outputs of commands.

Common Redirection Operators:

- `0<` or `<` Redirects standard input from a file

```
grep "Hello" < hello.txt
# Uses hello.txt as input for the `grep` command
# uses the ‘grep’ to search for the string "Hello" in the file
hello.txt
```

- `1>` or `>` Redirects standard output to a file (overwrites the file)

```
echo "Hello, World!" > hello.txt
# Uses the output generated by the command `echo` as the input to
hello.txt
# Writes to hello.txt (overwrites)
```

- `2>` Redirects standard error to a file

```
ls non_existing_dir 2> error.log
# Uses the error message generated by the command `ls` as the input to
error.log
# Redirects error message to error.log
```

- `&>` Redirects both standard output and standard error to a file

```
ls good_dir &> output.log
# Uses both the error message and output generated by the command `ls`
as the input to error.log
# Redirects both output and error to output.log
```

- `>>` Redirects standard output to a file (appends to the file)

```
echo "Welcome" >> hello.txt
# Appends to hello.txt
```

- `i>&j` Redirects (file descriptor) i to wherever (file descriptor) j is currently directed

```
$ command > file.log 2>&1

# `command > file.log` redirects the stdout(file descriptor 1) to
file.log (use the ouput generated by the `command` as the input to
file.log)
# `2>&1` redirects the stderr to wherever the stdout is currently
directed, where's the `file.log` in this case

# This line redirects both stdout and stderr of `command` to
`file.log` (where the stdout is directed)
```

- `i>&-` Closes (file descriptor) i, discard what i does
(effectively discarding command standrad output or error)

```
$ command 2>&-
# This line runs `command` while discarding any error messages it
generates.
```