SysAdmin - Part 2 - Meta-characters & Process handling

Michel FACERIAS

15 octobre 2024

Polytech Montpellier

Université de Montpellier

Table des matières

1 Understanding meta-characters

A **command and its arguments** are separated by a whitespace : this is a **command line**. This whitespace is understood as a separator between the command line arguments. We have already seen some other when parsing paths, special paths (`/ ~ . ..`), and when writing unilines (;).

They are called **meta-characters**. Instead of being passed to the command, the **shell interprets the meta-characters**, changing the behavior of a command or replacing them with an interpretation. We call this **shell expansion**.

Remember to never use meta-characters in file names.

We will learn how to use the other characters mentioned bellow in a next part :
- `#` starts a comment, all that is writen after is unreadable by the shell ;
- `&` `&&` `||` are used to control process launch ;
- `|` `>` `>>` `<` are used to handle process flow ;
- `$` is used to handle variables ;
- `"` is used to group sub-chains, but allow shell expansion ;
- `'` is used to group sub-chains, but avoid shell expansion.

⚠ Beware, **do not confuse meta-characters and regular expressions** :
- **meta-characters** are used **by the shell**, and alter the command line ;
- **regular expressions** are used **by commands which are able to understand them**, they **should** be protected against the shell expansion.

1.1 How the shell expands meta-characters

1.1.1 Concept : Meta-characters in file names

Meta-characters in file names are used to replace characters :
- `?` is the single character wildcard and represents exactly one character ;
- `*` is the character sequence wildcard and stands for any sequence of characters (including no character) ;
- `[ ]` are used to give list of characters ;
- `!` is used to negate a statement.

Here are some examples, using `ls` command. First, we start by a full view of the files in the current directory. As we can see, 5 files are present :

```
$ ls
file.txt file1.txt file2.txt file3.txt file4.txt
```

Let's try to show only numbered files, replacing the digit by a meta-character. The pattern of the file name is :
- the begining of the filename : `file`
- one single character : `?`
- the end of the filename : `.txt`

```
$ ls file?.txt
file1.txt file2.txt file3.txt file4.txt
```

It is possible to ask for 5 characters, and the same filename end :

```
$ ls ?????.txt
file1.txt file2.txt file3.txt file4.txt
```

Remember that `.` (dot) is a normal character :

```
$ ls ??????txt
file1.txt file2.txt file3.txt file4.txt
```

You can also replace several characters with a single meta-character :

```
$ ls *.txt
file.txt file1.txt file2.txt file3.txt file4.txt
```

And here, it's equivalent to :

```
$ ls *txt
file.txt file1.txt file2.txt file3.txt file4.txt
```

We can be even more imprecise :

```
$ ls *
file.txt file1.txt file2.txt file3.txt file4.txt
```

We could use a list of character (digits and letters are characters), here 1, 2 or 3 :

```
$ ls file[123].txt
file1.txt file2.txt file3.txt
```

Here the character are contiguous, so we can make it simple :

```
$ ls file[1-3].txt
file1.txt file2.txt file3.txt
```

We can combine a list and a wildcard :

```
$ ls [a-z]ile*
file.txt file1.txt file2.txt file3.txt file4.txt
```

But, remember that filenames are case-sensitive :

```
$ ls [A-Z]ile*
```

We can use negation in the pattern, to find filenames without any digit :

```
$ ls file[!0-9]*
file.txt
```

1.1.2 To do : Reproduce the examples above

Make a folder named `test`, and move into it. Use `touch` to create files `file.txt file1.txt file2.txt file3.txt file4.txt`. Reproduce all the examples above, and try some variations.

1.1.3 Question : Dealing with meta-characters

What are the good expressions to list :

1. files beginning with `Ba`, following by 3 unknown characters and ending with `txt` ?
2. files numbered from `file000.pdf` to `file999.pdf` ?
3. all files in the current directory ?
4. any file which name starts with any character within `a-p` or any digit within `0-5` ?
5. any file which name starts with any of these characters `afgh` and containing the character `o` ?

1.1.4 To do : Implement the questions above

In the folder named `test` and created before, create needed sample files and test your answers.

1.2 How to avoid the shell expansion of meta-characters

1.2.1 Concept : Using an escape character

It is possible to use \ as an **escape character to force the shell** to understand **literally a meta-character**.

We are going to use the same sample folder as in 1.1.1 part :

```
$ ls *
file1.txt  file2.txt  file3.txt  file4.txt  file.txt

$ ls \*
ls: '*': cannot access '*': No such file or directory
```

Using \, the shell is forced to use * literally, and then, the `ls` command receives it. As there is no file using * in its name, `ls` shows an error.

1.2.2 Concept : Using a protection

It is possible to use a **protection to force the shell** to understand **literally a meta-character** too.

We are going to use the same sample folder as in 1.1.1 part :

```
$ ls *
file1.txt  file2.txt  file3.txt  file4.txt  file.txt

$ ls '*'
ls: cannot access '*': No such file or directory
```

By using **' before and after the chain** we want to protect, the shell is forced to use * **literally**, and then, the `ls` command receives it. Its behavior is identical to before.

But, as we can see, the error message is exactly the same. In both cases, `ls` warns that **'*'** is not present in any file name.

1.2.3 Concept : Sub-chain grouping, a definitive explanation

In part 1, we defined `'` and `"` as **sub-chain grouping meta-chararcter**. Let's have a closer look, by using an expansion of a variable :

```
$ echo *
file1.txt  file2.txt  file3.txt  file4.txt  file.txt

$ echo '*'
*

$ echo "*"
file1.txt  file2.txt  file3.txt  file4.txt  file.txt
```

As we can see, there is nothing to group. We just demonstrated the **protective and non-protective** effect.

So, `"` has no effect here. In fact, it's **used to group space-separated sub-chains** to avoid the shell to recognize the group as a unique argument in the CLI.

2 Process handling

2.1 Process birth, life and death

2.1.1 Concept : What is a process ?

A **process is a computer program under execution**. Linux **processes are isolated** and **do not interrupt each other's execution**.

Since **many processes are running at any given time** in Linux, they have to share the CPU. The **action of switching** between two executing processes on the CPU is called **process context switching**.

Process are identified by a unique number named **PID** (Process IDentifier). Whith Linux, the **first program** started has the **PID 1**.

The following processes **increment the number** up to **PID 4 294 967 295** (32 bit), **then start again from the beginning** avoiding the PIDs already used.

2.1.2 To do : How to identify a process ?

Open a console and type `ps aux`. You will see something like this :

```
$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START    TIME COMMAND
root           1  0.0  0.1   1272   196 ?        S    Aug19    0:06 init [2]
root           2  0.0  0.0      0     0 ?        SW   Aug19    0:01 [keventd]
root           3  0.0  0.0      0     0 ?        SW   Aug19    0:01 [kapmd]
root           4  0.0  0.0      0     0 ?        SWN  Aug19    0:01 [ksoftirqd_CPU0]
root           5  0.0  0.0      0     0 ?        SW   Aug19    7:00 [kswapd]
root           6  0.0  0.0      0     0 ?        SW   Aug19    0:00 [bdflush]
root           7  0.0  0.0      0     0 ?        SW   Aug19    0:18 [kupdated]
root           8  0.0  0.0      0     0 ?        SW   Aug19    1:23 [kjournald]
...
root         214  0.0  0.4   2784   600 ?        R    Aug19    0:07 /usr/sbin/sshd
root         217  0.0  1.5   1976  1968 ?        SL   Aug19    0:13 /usr/sbin/ntpd
daemon       250  0.0  0.1   1384   180 ?        S    Aug19    0:00 /usr/sbin/atd
root         253  0.0  0.1   1652   232 ?        S    Aug19    0:02 /usr/sbin/cron
root         256  0.0  0.1   1252   136 tty1     S    Aug19    0:00 /sbin/getty 38400 tty1
root         257  0.0  0.1   1252   136 tty2     R    Aug19    0:11 /bin/bash
root        5072  0.0  0.4   3840   564 ?        S    Aug27    0:00 /usr/sbin/squid -D -sYC
proxy       5075  1.2 61.7 124456 78352 ?        S    Aug27  119:16 (squid) -D -sYC
proxy       5086  0.0 61.7 124456 78352 ?        S    Aug27    0:01 (squid) -D -sYC
proxy       5087  0.0 61.7 124456 78352 ?        S    Aug27    0:15 (squid) -D -sYC
```

2.1.3 Question : Find some process information

Answer the questions below using the example above :

1. What is the oldest process, and what is his PID ?
2. What is the PID of the `sshd` process ?
3. What is the name of the user running process 1 ?
4. How much time did this process take ?
5. When did it start ?
6. Could you explain the difference between the execution time and time of life (now - start time) of the `sshd` process ?
7. Why isn't there any PID 215 ?

2.1.4 Concept : The birth of a process

Linux has **system calls** defined for basic OS functions like file management, network management, process management, and others. One of them allows **to launch a program and generates a process**.

In fact, **another process calls the system and asks it to run a program**. Then, **each process is the child of another process**. Normally, if a **parent process dies, all child processes die** (Some processes are built to detach themselves from a dead parent, but it's not the standard case). The **PID of a parent** is called **PPID** (Parent Process IDentifier).

The first, `init`, is the ancestor of all the others. **Only `init` has no ancestor**, except the Linux kernel itself.

2.1.5 Question : Luke, I am your father !

The `ps auxf` command adds a tree structure that shows the relationship between processes (when it exists).

```
theForce   4542  2.4  1.9  37252 15000 ?        Sl   18:59   0:00 anakin
theForce   4544  0.0  0.1   2932   868 ?        S    18:59   0:00  \_ luke
theForce   4545  0.0  0.2   4716  1960 pts/1    Rs   18:59   0:00  \_ leia
theForce   4553  0.0  0.1   3624   984 pts/1    R+   19:00   0:00      \_ ben
```

1. What is the parent of the *ben* process and its PID ?
2. Which processes have the same parent ? What are their PIDs ?
3. What happens after 4542 process dies ?

2.1.6 Concept : Killing a process

There are two commands to kill a process from the outside of it :
— `kill`, which takes one or more PIDs as arguments ;
— `killall`, which takes as argument the name of the process.
`kill` is therefore more precise, because it **only affects the PIDs explicitly** mentioned.

Each one **sends** to the concerned process the **SIGTERM signal**. This signal is **a shutdown request** that the process can do things before terminating, thus **possibly saving a work in progress**.

It is possible to send the **SIGKILL signal** instead of SIGTERM. This **method is more violent** because the signal is intercepted by the system and the **process concerned is purged without notice**, making it **impossible to save a work in progress**. This method should only be used with stubborn or uncontrollable processes.

2.1.7 To do : Licence to kill

Open a console and run `geany` (a graphic text editor) in background mode (using the ampersand as argument). Thanks to background mode, the CLI becomes usable immediately (we will explain this later).

```
$ geany &
[1] 212991
```

Type some text in `geany`. Don't save your work. Then, from the console invoke `kill` using `geany` PID :

```
$ kill 212991
```

The CLI becomes usable immediately, and `geany` suggests to save your work. Dismiss this action. Then, from the console invoke `kill -SIGKILL` using `geany` PID :

```
$ kill -SIGKILL 212991
```

`geany`'s window close immediately.

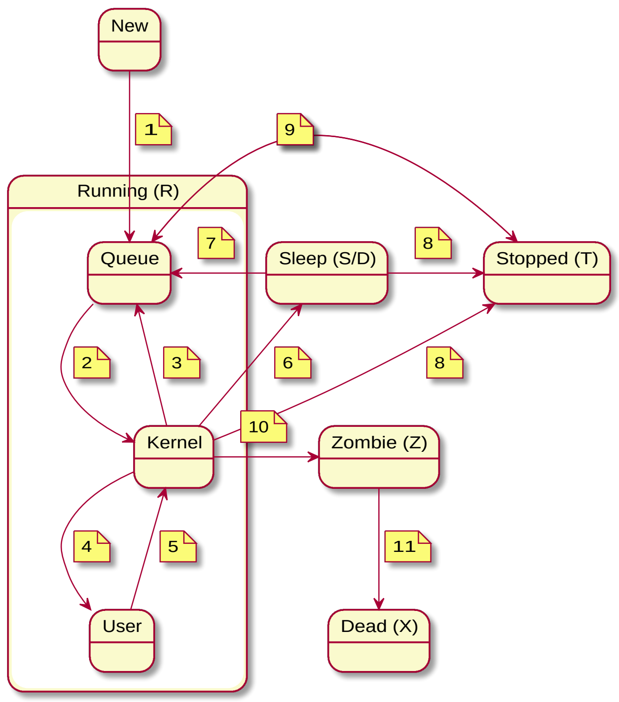
2.1.8 Question : `SIGTERM` vs `SIGKILL`

1. What happens when you type `kill 212991` ?
2. What happens when you type `kill -SIGKILL 212991` ?
3. Regarding to `man 7 signal`, what should happen if you type `kill -9 212991` ?
4. What should happen if you type `kill -15 212991` or `kill -SIGTERM 212991` ?

2.1.9 Concept : The lifecycle of a process

The figure below shows the lifecycle of a process :

1. The process has asked for the necessary resources and has everything to run properly. It the waits for a CPU core slot ;
2. The process is elected by the scheduler and begins processing using a CPU core instantly ;
3. The scheduling algorithm forces the running process to give up its execution right to ensure that each process can have a fair share of CPU resources ;
4. The process returns from a system call or an interrupt ;
5. The process makes a system call or suffers an interruption ;
6. The process requests external resources. It is mainly IO-based such as to read a file from the disk or make a network request ;
7. The process returns when the event has occurred ;
8. The process is stopped by a STOP signal ;
9. The process is woken up by a CONTINUE signal ;
10. The process is finished, but the resources it uses have not yet been released ;
11. All resources are released by the operating system.



The letter in brackets show the status (`STAT` column in `ps`) of the process :
— R : running (kernel-land or userland) or runnable (waiting on queue) ;
— T : stopped (suspended by CTRL+Z) or traced ;
— S : interruptible sleep (waiting for an event to complete) ;
— D : uninterruptible sleep (deep sleep, usually IO) ;
— Z : defunct (zombie, rarely seen on kernels $>=3$) ;
— X : dead (should never be seen) ;
— W : paging (transitional state rarely seen, not shown on the figure).

2.1.10 Question : Process states

Regarding to the `ps` command output at 2.1.2, answer the following questions :

1. Which processes are asleep ?
2. Which processes are active ?

2.1.11 To do : How to use the `ps` command ?

Search the options using the manual to view the processes :

1. In progress in the current console ;
2. Adding the name of their owner ;
3. As a whole (all) ;
4. As a whole (included without `tty`) ;
5. As a whole and with the ancestor tree.

2.1.12 Concept : Process priority

It is possible to run a **process** with **modified scheduling priority**, even if today the **computing power** makes the **manual control of the scheduler almost obsolete**. In fact, it is **only used for some system tasks**. However, it can be useful **when working on embedded systems** of lower power

Niceness values range is :
— from **-20** (most favorable to the process) ;
— to **19** (least favorable to the process) ;
— default value is **0**.

⚠ Only **superuser** can use **negative niceness** or can **decrease the value** (give more priority) to any process.

The `nice` command can be used :
— with no argument, it shows the niceness of the current process ;
— when giving a niceness value and a command as argument, it runs this command with the niceness value given.

The `renice` command can be used to alter priority of running processes. You must give the new niceness and the PID of the process as arguments.

To see more, read the manual : `man nice` or `man renice`.

2.1.13 To do : Play with niceness

`$$` variable is the PID of the parent process, and we are going to use it as an alias to retrieve it. Of course, the PID below will differ from those you will read on your computer. Open a console, type the following commands and try to understand the steps.

```
$ echo $$
336706

$ nice
0
```

The current console (the parent of `echo`) is running `bash` as 336706 PID, with the default niceness 0.

```
$ echo $$
336706

$ nice bash

$ echo $$
336736

$ nice
0

$ exit
exit

$ echo $$
336706
```

We are launching another `bash` (336736), with the default niceness 0, and then exiting to go back to its parent (336706).

```
$ echo $$
336706

$ nice -n 10 bash
$ echo $$
336744

$ nice
10

$
exit

$ echo $$
336706
```

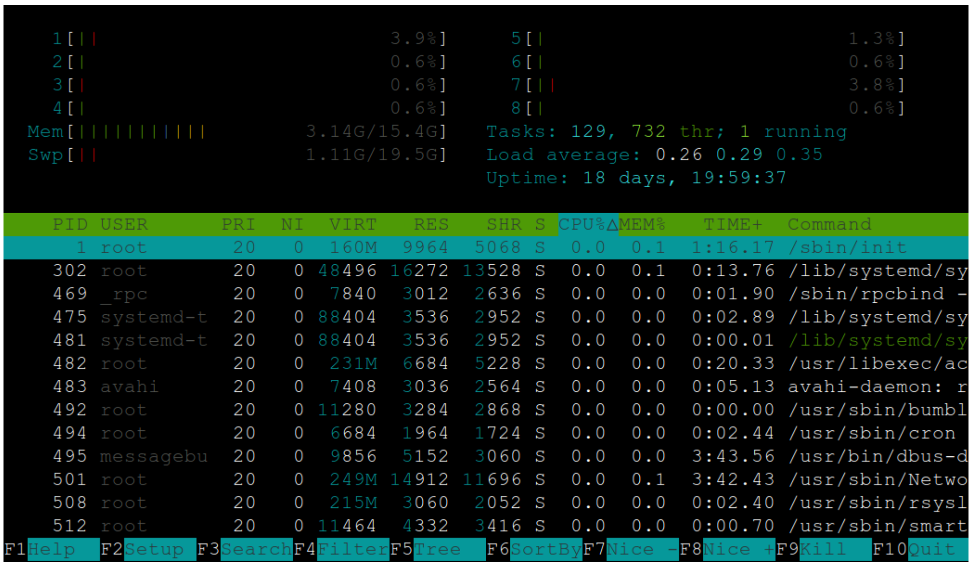
We are launching an other `bash` (336744), with the niceness +10 (less priority), and then exiting to go back to its parent (336706).

```
$ echo $$
336706

$ nice -n 10 bash

$ echo $$
337188

$ nice
10

$ renice -n 11 -p $$
337188 (process ID) old priority 10, new priority 11

$ nice
11

$ renice -n 1 -p $$
renice: failed to set priority for 337188 (process ID)

$ nice
11

$ renice -n 25 -p $$
337188 (process ID) old priority 11, new priority 19

$ nice
19

$ exit

$ echo $$
336706
```

We are launching a new `bash`, and playing with its niceness. We can only increase it, not more than 19.

2.1.14 Concept : Process dashboard

Linux offers a text dashboard to control the state of the processes :
— `top` is the historical version ;
— `htop` is the improved version, colorized and more user friendly.
The illustration below shows `htop` in operation.



```
    1[||                         3.9%]     5[|                          1.3%]
    2[|                          0.6%]     6[|                          0.6%]
    3[|                          0.6%]     7[||                         3.8%]
    4[|                          0.6%]     8[|                          0.6%]
  Mem[|||||||||||         3.14G/15.4G]   Tasks: 129, 732 thr; 1 running
  Swp[||                  1.11G/19.5G]   Load average: 0.26 0.29 0.35
                                         Uptime: 18 days, 19:59:37

    PID USER      PRI  NI  VIRT   RES   SHR S CPU%▵MEM%   TIME+  Command
      1 root       20   0  160M  9964  5068 S  0.0  0.1  1:16.17 /sbin/init
    302 root       20   0 48496 16272 13528 S  0.0  0.1  0:13.76 /lib/systemd/sy
    469 _rpc       20   0  7840  3012  2636 S  0.0  0.0  0:01.90 /sbin/rpcbind -
    475 systemd-t  20   0 88404  3536  2952 S  0.0  0.0  0:02.89 /lib/systemd/sy
    481 systemd-t  20   0 88404  3536  2952 S  0.0  0.0  0:00.01 /lib/systemd/sy
    482 root       20   0  231M  6684  5228 S  0.0  0.0  0:20.33 /usr/libexec/ac
    483 avahi      20   0  7408  3036  2564 S  0.0  0.0  0:05.13 avahi-daemon: r
    492 root       20   0 11280  3284  2868 S  0.0  0.0  0:00.00 /usr/sbin/bumbl
    494 root       20   0  6684  1964  1724 S  0.0  0.0  0:02.44 /usr/sbin/cron
    495 messagebu  20   0  9856  5152  3060 S  0.0  0.0  3:43.56 /usr/bin/dbus-d
    501 root       20   0  249M 14912 11696 S  0.0  0.1  3:42.43 /usr/sbin/Netwo
    508 root       20   0  215M  3060  2052 S  0.0  0.0  0:02.40 /usr/sbin/rsysl
    512 root       20   0 11464  4332  3416 S  0.0  0.0  0:00.70 /usr/sbin/smart
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Processes are sorted in descending order of CPU usage. `htop` presents a command help in the form of a menu bar located at the bottom of the window. You can use it to handle process (`kill`, `renice`) instead of using the CLI. Like `top`, it retains the use of the Q key to exit.

2.1.15 To do : Measuring execution time

Open a console and use the `time` command to measure the duration of another command. `dd` is used to make special copies. Here we are going to copy 100 000 blocs of 512 bytes from the pseudo-random generator to a standard file.

```
$ time dd if=/dev/urandom of=toto count=100000
100000+0 blocs readed
100000+0 blocs writed
51200000 bytes (51 MB, 49 MiB) writed, 1,15507 s, 44,3 MB/s

real   0m1,160s
user   0m0,081s
sys    0m0,967s
```

First, the screen shows the result of two commands :
- `dd` itself, the 3 first lines ;
- `time`, the 3 last lines, showing `dd` execution times.

As you can see, most of the compute time is consumed by sys(the system). It's the time needed to randomize numbers and to write in the file. The time in user(the userland) is very short, and corresponds with the starting time of the process, calling the system API, and printing the status. The real time is the apparent time for the total execution and is not the sum of the others.

Try to change the value of *count* attribute, and/or use `/dev/zero` as input, then analyse the results.

2.2 Linux Process vs. Thread

We'll discuss the details of the process and thread in the context of Linux.

2.2.1 Concept : Process

A **process is a computer program under execution**. Linux is running many processes at any given time. We can monitor them on the terminal using the `ps` command.

```
$ ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 mai22 ?        00:01:16 /sbin/init
root           2       0  0 mai22 ?        00:00:00 [kthreadd]
...
_rpc         469       1  0 mai22 ?        00:00:01 /sbin/rpcbind -f -w
systemd+     475       1  0 mai22 ?        00:00:02 /lib/systemd/systemd-timesyn
root         482       1  0 mai22 ?        00:00:20 /usr/libexec/accounts-daemon
avahi        483       1  0 mai22 ?        00:00:05 avahi-daemon: running [mfa-l
root         492       1  0 mai22 ?        00:00:00 /usr/sbin/bumblebeed
message+     495       1  0 mai22 ?        00:03:45 /usr/bin/dbus-daemon --syste
root         501       1  0 mai22 ?        00:03:43 /usr/sbin/NetworkManager --n
...
www-data     710     705  0 mai22 ?        00:00:00 nginx: worker process
www-data     711     705  0 mai22 ?        00:00:00 nginx: worker process
...
mfaceri+  334422  278097  2 11:49 ?        00:03:32 /usr/lib/firefox-esr/firefox
mfaceri+  335230    1216  1 12:03 ?        00:02:42 /usr/bin/latexila --gapplica
mfaceri+  335339    1216  0 12:04 ?        00:00:46 atril /home/mfacerias/Deskto
mfaceri+  335345    1216  0 12:04 ?        00:00:00 /usr/lib/atril/atrild
...
mfaceri+  338657  337188  0 14:44 pts/0    00:00:00 ps -ef
```

As we run new commands/applications or the old commands are being completed , we can see the number of processes grow and shrink dynamically. **Linux processes are isolated** and do not interrupt each other's execution.

With a **PID, we can identify any process** in Linux. We can see PID as the second column in the output of the above `ps` command.

Since **many processes are running at any given time** in Linux, they have to **share the CPU**. **Process context switching is expensive in compute** because the kernel has to save old registers and load current registers, memory maps, and other resources.

2.2.2 Concept : Threads

A **thread is a lightweight process**. A **process can do more than** one unit of work concurrently by creating **one or more threads**. These threads, being **lightweight, can be spawned quickly**.

Let's see an example and **identify the process and its thread** in Linux using the `ps -eLf` command. We're interested in these attributes :

— PID : Unique process identifier ;
— LWP : Unique thread identifier inside a process ;
— NLWP : Number of threads for a given process.

```
$ ps -eLf
UID          PID    PPID     LWP  C NLWP STIME TTY          TIME CMD
...
root         690       1     690  0    2 Jun28 ?        00:00:00 /sbin/auditd
root         690       1     691  0    2 Jun28 ?        00:00:00 /sbin/auditd
root         709       1     709  0    4 Jun28 ?        00:00:00 /usr/sbin/ModemManager
root         709       1     728  0    4 Jun28 ?        00:00:00 /usr/sbin/ModemManager
root         709       1     729  0    4 Jun28 ?        00:00:00 /usr/sbin/ModemManager
root         709       1     742  0    4 Jun28 ?        00:00:00 /usr/sbin/ModemManager
```

We can easily identify **single-threaded** and **multi-threaded** processes by their **NLWP** values. PIDs 690 and 709 have an **NLWP of 2 and 4**, respectively. Hence, they are **multi-threaded**, with 2 and 4 threads. All processes having an **NLWP of 1** are **single-threaded**.

When taking a closer look in a **multi-threaded** process, **only one LWP matches its PID**, and the others have different values of LWP. Also, **note that the value assigned to an LWP**, is **never given to another** process.

2.2.3 Concept : Single-threaded vs. multi-threaded process

Any thread created within the process shares the same memory and resources as the process.

In a **single-threaded** process, **the process and thread are the same**. We can also validate that PID and LWP are the same for the single-threaded process.

In a **multi-threaded** process, the process has more than one thread. Such **processes accomplish multiple tasks** simultaneously or almost at the same time, **if the CPU is multi-threaded (and/or multi-core)**.

But, as it's said before, the **thread shares the same address space as the process**. Therefore, **spawning a new thread within a process becomes cheap**, in terms of system resources (computing, I/O, time).

2.2.4 Question : Is Firefox multi-threaded ?

First, run a single instance of Firefox with a single tab. After that, add a tab. At the end add an other Firefox main window with many tabs. Using each scenarios, use `ps -eLf` and analyse the results.

1. How many processes are there ?
2. How many threads are there ?
3. Conclude.

2.3 More jobs from the same shell

Let's assume that we only have one console and that we need to use a command that will last a long time. As an example, we will illustrate this concept by using the `xeyes` command that only opens a window with 2 eyes following the mouse pointer until its killed.

2.3.1 Concept : Starting foreground

From a CLI, **just call `xeyes` and run it**. When a program is invoked like this, it is **named as a foreground execution mode**.

The **console stays busy** until the program stop by itself or is killed.

But it remains possible to **stop its execution**, to put it to sleep, **using** `CRTL+Z`.

```
$ xeyes
  <== here xeyes is running, but the CLI is busy. The eyes are following the mouse pointer
...
^Z   <== here CTRL+Z was keyed. The eyes are asleep and not following the mouse pointer
[1]+  Stopped                 xeyes

$ ps auxf    <== here we can used the CLI again !
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
...
mfaceri+  341092  0.0  0.0   2448   616 pts/0    TN   17:16   0:00 xeyes
...
```

First and foremost, use your mouse to move the `xeyes` window at the left top of your screen. It will be important later.

As you can see, `xeyes` is in stopped state. `[1]` is the ID of this job attached to that console. The CLI began free, and the use of `ps` proves the `T` status.

2.3.2 Concept : Exiting stopped state

At this point, the CLI is free to complete new orders.

So, **it is possible to wake up the program** and put it in **foreground execution mode** again using `fg` command.

Then, we will put it to sleep, using `CRTL+Z`.

```
$ fg
xeyes
 <== here xeyes is woken up and starts following the mouse pointer again
     CLI is busy again
...
^Z   <== here xeyes is stopped again.
[1]+  Stopped                 xeyes

$  <== here the CLI is free again.
```

In this case, `fg` was invoked without any argument, because there is only one job attached to the console. In case of more than one job, its ID is mandatory (see `man fg`).

2.3.3 Concept : Execution in background mode

At this point, the CLI is still free to complete new orders.

So **it is possible to wake up the program** and put it in **background execution mode** using `bg` command.

```
$ bg
[1]+ xeyes &  <== here exyes is waked up and starts following the mouse pointer

$  <== here the CLI became free again.
```

`bg` was invoked without any argument too, because there is just one job attached to the console again. In case of more than one job, its ID is mandatory (see `man bg`).

Notice the ampersand (&) on the status line, meaning `xeyes` run in background mode.

2.3.4 Concept : Direct execution in background mode

At this point, using the free CLI, we are going to start a **second instance** of `xeyes`, **directly in background mode**.

```
$ xeyes &
[2] 342353  <== here we can read the job ID and the PID of the associated process.

$  <== and the CLI is immediatly free
```

In order to do that, we **used the ampersand at the end of the command line**. Note that the CLI began immediately free and that we know the ID of the job and the PID of the associated process. Move the new `xeyes` window at the right top of your screen. Then you have two `xeyes` windows.

2.3.5 Concept : Identifying the jobs

You can use the `job` command to **list the jobs attached to a console**.

```
$ jobs
[1]-  Running                 xeyes &
[2]+  Running                 xeyes &
```

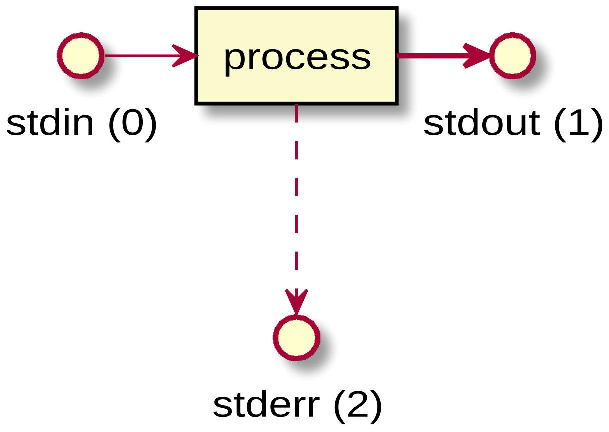
Now, you can see that 2 jobs are running in the background. The 4 eyes are following your mouse pointer. You can address each job, using its job ID.

2.3.6 To do : Juggling multiple jobs

Redo what was done before and analyse all the jobs creation and handling. Then, by replicating the `fg / CTRL+Z / bg` cycle, and with the help of the `jobs` command to find the jobs IDs and eyes movement to find which is stopped or not, manipulate the different jobs. Create 1 or 2 more `xeyes` instances to consolidate your skills.

2.4 Streams

When a **process is started**, it **automatically opens 3 streams** :
— The **standard input (*stdin*)** stream which sends to the program what is entered on the keyboard ;
— The **standard output (*stdout*)** stream which receives what the program wants to display on the screen ;
— The **standard error (*stderr*)** stream which receives error messages from the program and which is normally sent back to the screen too.



Each stream is identified by **a number** (a file handler).

2.4.1 Concept : *stdout* redirection to file

In this case, the display is replaced by a file. It will contain what would normally be displayed on the screen.

```
        $ cmd 1> path_to_file

        $ cmd > path_to_file
```

It is therefore possible to keep track of the execution of a command.

```
        $ ls -l > file_list
```

The example given provides a file that contains a list of files.

2.4.2 Concept : Concatenation vs. creation

Using a single `>`, if the file does not exist, it is created. If it exists, its initial content is overwritten.

```
$ cat file
cat: file: No such file or directory

$ echo "Hello" > file

$ cat file
Hello

$ echo " world !" > file

$ cat file
 world !
```

Using a double `>>` if the file does not exist, it is created. If it exists, the result of the command is concatenated to its initial content.

```
$ cat file
cat: file: No such file or directory

$ echo "Hello" > file

$ cat file
Hello

$ echo " world !" >> file

$ cat file
Hello world !
```

2.4.3 Concept : *stderr* redirection to file

In this case, the display is replaced by a file. It will contain the error messages what would normally be displayed on the screen.

```
$ cmd 2> path_to_file
```

Assuming that `file` does not exist :

```
$ cat file
cat: file: No such file or directory

$ cat file 2> error_file

$ cat error_file
cat: file: No such file or directory
```

It is therefore possible to keep track of the execution errors of a command.

2.4.4 Concept : *stdout* and *stderr* redirection to file

We use this trick to join the two output streams. In fact, we return first *stderr* to *stdout* (2 in 1).

```
$ cat file
cat: file: No such file or directory   <== we see stderr output

$ cat file 2>&1
cat: file: No such file or directory   <== we see stderr output via stdout
```

```
$ cat file 1> error 2>&1  <== we put stdout in a file and stderr in stdout

$ cat error
cat: file: No such file or directory
```

Beware, it is **2>&1** and not 2>1, because in this case you will create a file named 1 !

2.4.5 Concept : *stdin* redirection from file

In this case, the **standard input stream** is replaced by the contents of **a file**.

```
$ cmd < path_to_file
```

It is therefore possible to use a file to automate the sending of what would be entered from the keyboard.

```
$ cat instructions
pq

$ fdisk < instructions
```

This example allows the `fdisk` program to **run automatically** by reading its **instructions from a file** instead of the keyboard. Here, `pq` means *print* and *quit*.

2.4.6 Concept : Pipelining

This is a trick to **redirect the *stdout*** of the first command **to the *stdin*** of the second one.

```
$ cmd1 | cmd2
```

The meta-character `|` is called a "pipe". You **can't use the *stderr*** stream in pipelining.

```
$ ps auxf |grep firefox

mfaceri+  278097  3.2  4.3 4356200 701152 ?      Sl   Jun06 205:07              \_ /usr/lib/
    firefox-esr/firefox-esr
mfaceri+  278190  0.0  1.4 2619204 233308 ?      Sl   Jun06   1:14              |   \_ /usr/
    lib/firefox-esr/firefox-esr ...
```

In this example, we use `grep` as a filter of the `ps auxf` *stdout*, looking for line containing `firefox`. Of course, the header line of `ps` is invisible because `firefox` is not in it.

2.4.7 To do : Examples to decode

Test and comment the following examples :

1. `ls -l -d ~| cut -c 5-7`
2. `ls -l -d ~| cut -d " " -f 4`

2.4.8 To do : *stdout* redirection

1. Use the redirection to obtain a file that contains the list of folders in `/var` ;
2. Add, at the end of this file, the list of folders contained in `/var/spool`.

2.4.9 To do : `stdout` and `stderr` redirection

The command `find /proc -name net` searches for all files and folders named `net` that are in the `/proc` folder. The execution of this command takes quite a long time.

1. Test the command and explain why there are errors ;
2. Modify the command to have the errors in a `/error` file and the result on the screen ;
3. Modify the command to have the errors in a `/error` file and the result in a `/result` file ;
4. Modify the command to have the errors and the result in a `/all` file.