PROCESSES.

1.1.- Basic ideas about processes

1.- A program is just a file stored in a file on your hard drive (or any other permanent storage device).

2.- A process is a program stored in RAM memory that can be run by the CPU. In other words, is a program in action.

3.- In Linux, you can run one or more processes from the same program at the same time. For example, you can run three times Geany at the same time and Linux will start 3 different processes.

4.- Sometimes processs are run by the CPU and sometimes processes are waiting to be run by the CPU.

5.- The lifetime of a process is the interval of time a process is active and running.

6.- During its lifetime, a process needs system resources such as CPU time to run the code, Memory to hold its code and files/folder/hardware access permissions.

7.-The allocation of resources to each process running in the system is a duty of Linux.

8.- Process can send signals to others processes. Signals provide a mechanism for processes to communicate with each other.

1.2.- Process table. Process IDentifier and Parent Process IDentifier of a process.

1.- The process table holds information about all running processes that that is handled by the operating system.

2.- Each process is represented as an entry in the process table.

3.- Main Information hold in each entry of a process table includes:
- Process owner.
- Memory and CPU usage.
- Process identifier or PID.
- The parent process identifier or PPID.
- Process priority or niceness.
- Process state.
- Starting time and data of the process.
- Acumulative CPU time of the process.

4.- Each process running in the system is is identified by number called Process IDentifier (PID) and by another number called Parent Process IDentifier (PPID).

5.- A PID:
- Must be a unique identification number for each process running on the system.
- Processes can NOT share PID.
- As long as the process exists, it keeps the same PID number.
- When a process is ended (or finished or terminated), its PID is freed and eventually the operating system can assign this number to another process.
- Can not be changed.
- Maximun number of PID is stored in /proc/sys/kernel/pid_max and theorically could be 2^{22}.

6.- Because multiple processes of a program can be run at the same time. Different processes of one single program will have different PIDs.

7.- A process is also identified by its parent process ID or PPID:
- Whenever a process creates another process, the former is called the parent process while latter is called child process.
- A parent process can have several child processes, but a child process can have only one parent.
- Any process running on the system is a child process of a parent process.
- Only init, the first process created when you boot linux, has not a parent process. The PID of init is 1 and its PPID is 0.
- Parent processes manages child processes. That means:
  - Parent processes start and end child processes
  - A parent process is responsible for adding/removing its child processes to/from the process table.
  - A parent process must not be ended until all its child processes were ended and any information about them was released of the process table.

1.3.- Priority and the nice value.

1.- The operating system is able to assign **priorities to process**.

2.- The higher the priority of a process, the greater the amount of resources allocated to that process. In other words, The higher the priority of a process , the greater the amout of CPU time and Memory will be allocated to the process.

3.- When a new process is started, the operating system assigns to the process a value called **nice number** or **NI**.

4.- This number is related to process priority. The higher the nice number, the lower the priority of the process. Essentially, the *nice* value determines how “nice” a process is to other users, which is why a higher number represents a lower priority. A higher “niceness” means the process is “nicer” to other processes, allowing them to receive more CPU time o Memory. A lower “niceness” indicates a higher priority, giving the process more system resources.

5.- The nice value of each process running in the system is stored in the process table.

6.- Nice values range from -20 to 19 where -20 represents the highest Linux process priority, and *19* represents the lowest.

7.- Typically, processes start with a Linux *nice* value of *0*, representing the standard (or neutral) priority.

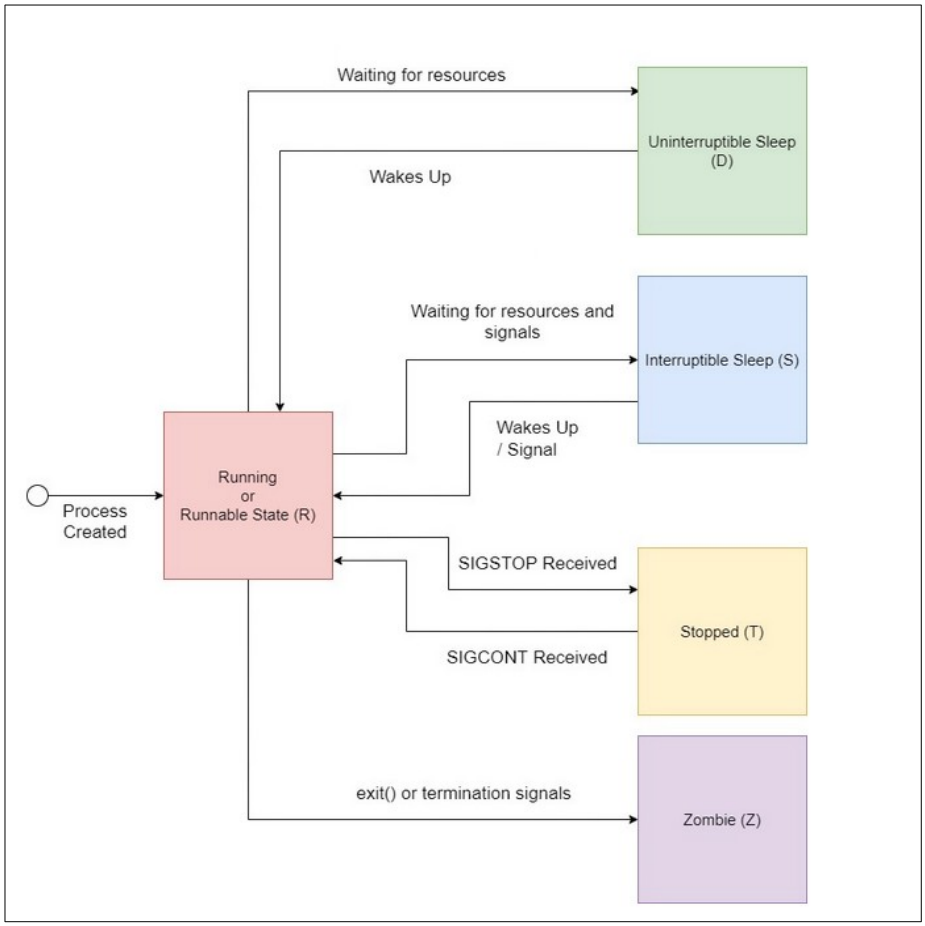
8.- The nice value of a process can be changed.

1.4.- States fo Linux processes

1.- While a process exist, it will be in one of the following five possible states:

- Running or Runnable (R)
- Uninterruptible Sleep (D)
- Interruptable Sleep (S)
- Stopped (T)
- Zombie (Z)

2.- This is the Linux diagram of processes states:



3.- Running or Runnable:
- Process is is either actively executing instructions on the CPU or in a queue waiting to execute instructions.
- An Running/Runnable process accepts signals.
- Process state code: **R**

4.- Interruptible Sleep:
- Process can not run instructions because is waiting for a resource or event (waiting for user input, a file system operation,..) to continue execution.
- An interruptible sleep process accepts signals.
- Process state code: **S**

5.- Uninterruptible Sleep:
- Process can not run instructions because is waiting for I/O operations (access to a printer, hard drive, network card,…) complete.
- An uninterruptible sleep process does not accept signals.
- Process state code: **D**

6.- Stopped:
- Process does not run instructions (it is suspended).
- A process changes to the stop stated when it receives the SIGSTOP signal.
- A process stopped changes to R state if it receives a SIGCONT signal.
- A process stopped ends if it receives a SIGKILL signal.
- Process state code: **T**

7.- Zombie:
- After a process finishes its execution, it enters the Zombie state.
- In this state, the process has completed its execution but still has an entry in the process table.
- A Zombie process is removed from the process table for its parent process.
- Also, a Zombie process is removed from the process table if the computer is powered off.
- A Zombie process does not accept signals because, in fact, it does not exist.
- Generally speaking, is difficult to find Zombie processes because they are quicly removed by their parents.
- Generally speaking, if a zombie process remains in the process table, means that the code of the child or parent process is a bad code (it has not been properly developed).
- Process state code: **Z**
- REMEMBER: A Zombie process does not exist but still remains in the process table until the parent process removes its entry.

1.5.- Signals

1.- Linux signals provide a mechanism for processes to communicate with each other.

2.- Name of signals begins with "SIG". Although signals are numbered, we normally refer to them by their names.

3.- The number of possible signals is limited to 64

4.-There are 2 sets of signals:
- Standardized signals:
  - There are 31 standarized signals that are numbered 1 to 31
  - These are signals with a predefined purpose
- Real-time signals:
  - There are 33 real time signals that are numbered 32 to 64.
  - These are Signals with no predefined purpose (programmers can use these signals as they want).

5.- In this exercise we are going to work with the following standarized signals:
- **SIGTERM:** It is the default signal sent by kill command. It ssks the process to end voluntarily. It ends the process in an orderly way. Default option.
- **SIGKILL:** Unlike **SIGTERM,** forces the process to end. Can't be blocked or handled.
- **SIGSTOP:** Suspend the process execution, putting it in *stopped* state. In this state, the process will do nothing but accept **SIGKILL** and **SIGCONT** signals. Can't be blocked or handled.
- **SIGCONT:** if a process is in stopped state, it will put it back in the *running/runnable* state and resume it execution. If the process is in any other state, it's silently ignored.
- **SIGINT:** Generated when the user type **<Ctrl>+C** in the terminal. It ends the current command processing and wait for user's next command.
- **SIGQUIT:** Generated when the user type **<Ctrl>+D** in the terminal. It ends the process (like SIGTERM) but also causes the shell to write a core file to help the so-called post-mortem debug.

1.6.- Foreground and background processes

1.- Foreground processes:

- These processes are user dependent. In other words, these processes require user interaction.
- If the foreground processes are run from the terminal, the shell prompt remains unavailable and will be available only when the foreground process is terminated or stopped.
- Generally speaking, the process activity remains clearly visible to users.

2.- Background processes:

- These processes are usually run independently from the user. In other words, these processes do not require user interaction.
- These processes can be started by the user or the automatically by the operating system.
- Most of the background processes are started by the operating system during the computer boot procedure.
- If the foreground processes are run from the terminal by a user, the shell prompt remains available.
- If you add an ampersand (&) at the end of a command, the process will be started in the background.
- Generally speaking, the process activity remains invisible to users.

1.7- Ideas that are beyond the scope of this exercise

1.- The Idle Kernel Threads, whose process state code is **I**.
2.- Orphan processes.
3.- Kernel and user Memory areas.
4.- User/Daemon/Kernel processes
5.- Kernel space
6.- Real-time signals