**State**

As a process executes it changes *state* according to its circumstances. Linux processes have the following states:

**Running**

The process is either running (it is the current process in the system) or it is ready to run (it is waiting to be assigned to one of the system's CPUs).

**Waiting**

The process is waiting for an event or for a resource. Linux differentiates between two types of waiting process; *interruptible* and *uninterruptible*. Interruptible waiting processes can be interrupted by signals whereas uninterruptible waiting processes are waiting directly on hardware conditions and cannot be interrupted under any circumstances.

**Stopped**

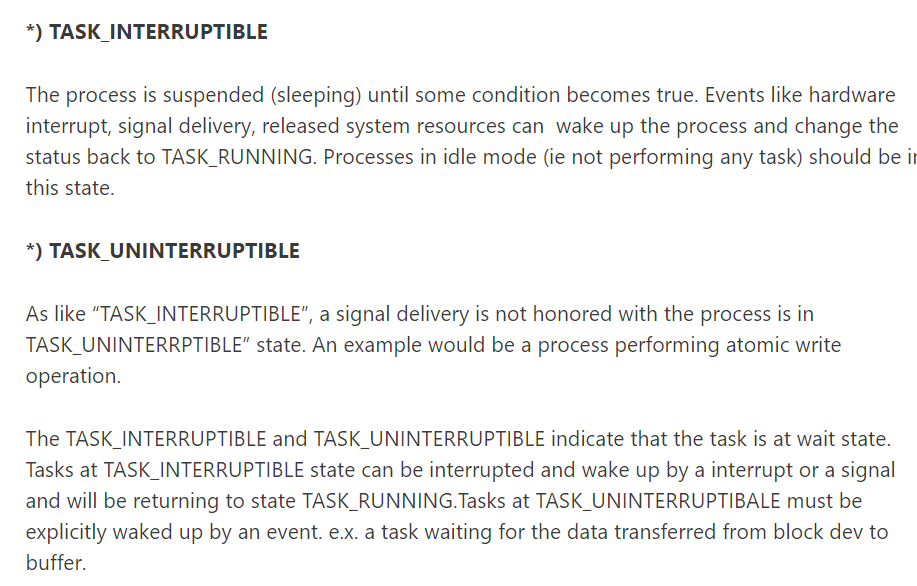
The process has been stopped, usually by receiving a signal. A process that is being debugged can be in a stopped state.

**Zombie**

* EXIT\_ZOMBIE: The process has terminated. It is lingering around just for its parent to collect some statistical information about it.
* EXIT\_DEAD: The final state (just like it sounds). The process reaches this state when it is being removed from the system since its parent has just collected all statistical information by issuing the wait4() or waitpid() system call.

TASK KILLABLE:

LinuxÃÂ® kernel 2.6.25 introduced a new process state for putting processes to sleep called TASK\_KILLABLE, which offers an alternative to the efficient but potentially unkillable TASK\_UNINTERRUPTIBLE and the easy-to-awaken but safer TASK\_INTERRUPTIBLE.



Atomic Operation: An operation during which a [processor](https://www.webopedia.com/TERM/M/microprocessor.html) can simultaneously [read](https://www.webopedia.com/TERM/R/read.html) a location and [write](https://www.webopedia.com/TERM/W/write.html) it in the same [bus](https://www.webopedia.com/TERM/B/bus.html) operation. This prevents any other processor or [I/O](https://www.webopedia.com/TERM/I/I_O.html) [device](https://www.webopedia.com/TERM/D/device.html)from writing or reading memory until the operation is complete.

Atomic implies indivisibility and irreducibility, so an atomic operation must be performed entirely or not performed at all.

Child processes

A process that is created by some other process during run-time. Usually child processes are created to execute some binary from within an existing process. Child processes are created using fork() system call. Normally process are made to run through shell/terminal. In that case the shell becomes the parent and the executed process becomes the child process. On Unix/Linux each process has a parent except the init process(we will learn about this later).

### Daemon Processes

These are special processes that run in background. They are system related process that have no associated terminal. These processes run with root permissions and usually provide services to processes.

Fork(): is a systemcall function like read() function which creates a new process. In Linux Process creation and execution is divided into 2 parts:

Creation of a process/child Process: Fork()

* returns PID of a child process
* No arguments passed
* After creation of the process both child and parent start execution from next instruction.

Can be identified by following:

1. Fork()>0 = Success in process creation
2. Fork()<0 = Unsuccessful Creation (Error)
3. Fork()=0 -> child process value which indicate child process creation.
4. Hence fork() returns childPID to parent process wheareas 0 to childprocess

Exec(): this is used for executing a process it has also many divisions.

Task Running: **R**

Task Interruptible : **S**

Task UnInterruptible: **D**

Task Killable: **K**

Stopped Process: **T**

Exit Zombie: **Z**

Exit Dead: **X**

PROCESS STATE CODES

R running or runnable (on run queue)

D uninterruptible sleep (usually IO)

S interruptible sleep (waiting for an event to complete)

Z defunct/zombie, terminated but not reaped by its parent

T stopped, either by a job control signal or because

it is being traced