Understanding process states in Linux[1] [2]

S. Parthasarathy
drpartha@gmail.com

Ref.: states.tex
Ver. code: 20191030a

Abstract

Understanding the concepts of a “process” and its “states” is essential for a clear view of the way Unix/Linux works. This article uses a commonly encountered analogy for explaining these concepts.

1 Background and terminology

Background : In Linux, and in many Unix flavours, a *process* is an instance of an executable *program.* Because Linux is a multi-user system, meaning different users can be running various programs on the system, each running instance of a program must be identified uniquely by the kernel. The kernel recognises prorams as ‘peocesses” .

1.1 What is a process ?

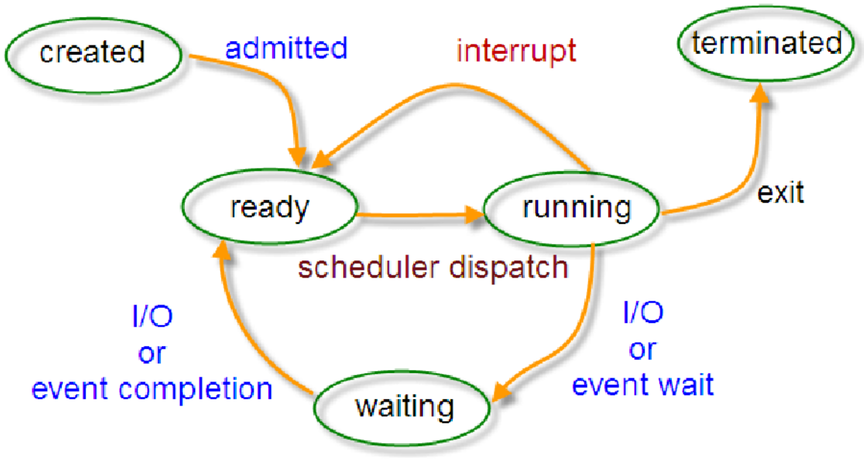
On creation, a program is usually a file kept in some storage medium. A program is thus a passive entity until it is launched, and a process (aka tasks) can be thought of as a program in action. Unix/Linux adds a whole lot of details to a program before it can be handled by the operating system. The bare executable program, along with all necessary details can now be called as a “process”. It is made up of the program instructions, data read from files, other programs or input from a system user.

There is thus a fine shade of difference beween a “program” and a “process” We will use an analogy to explain the difference. We can compare a “program” to a stand-alone car. Processes are like cars in a traffic system. More details are given below.

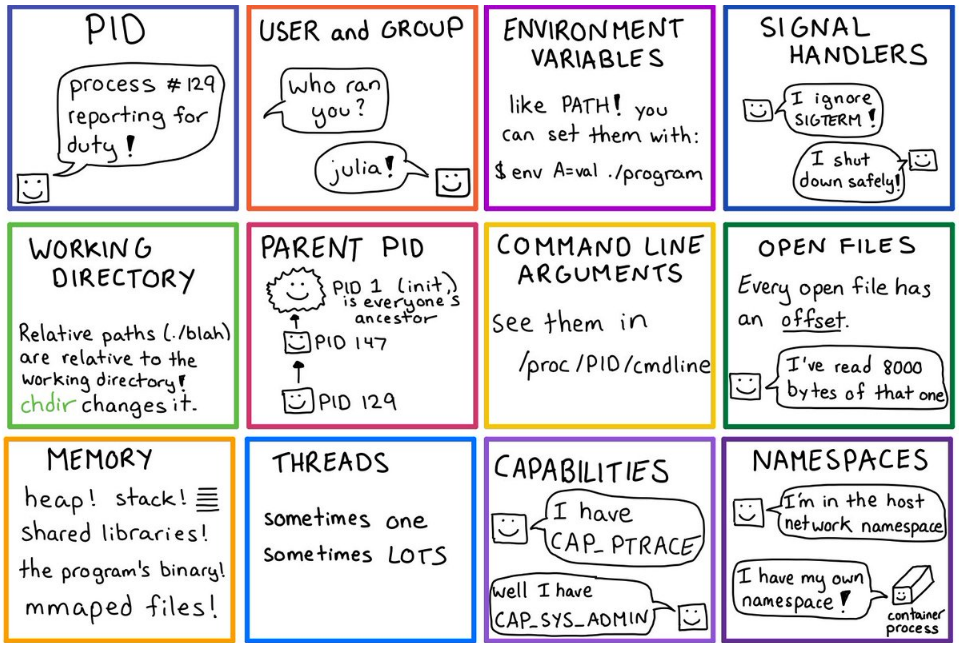
[1]This is a LaTeX document. You can get the LaTeX source of this document from drpartha@gmail.com. Please mention the Reference Code, and Version code, given at the top of this document

[2]Citation details : S. Parthasarathy, Understanding process states in Linux, `http://drpartha.org.in/publications/states.pdf`, Oct. 2019

As in every Unix flavour, in Linux a process can be in a number of *states*. A simplified view of processes is given below.



Processes are dynamic entities since they are constantly changing as their machine code instructions are executed by the CPU. Unix, and its younger avatar Linux are multitasking operating systems.


PID
process #129 reporting for duty!
USER and GROUP
who ran you?
julia!
ENVIRONMENT VARIABLES
like PATH! you can set them with:
$ env A=val ./program
SIGNAL HANDLERS
I ignore SIGTERM!
I shut down safely!
WORKING DIRECTORY
Relative paths (./blah) are relative to the working directory! chdir changes it.
PARENT PID
PID 1 (init) is everyone's ancestor
PID 147
PID 129
COMMAND LINE ARGUMENTS
see them in /proc/PID/cmdline
OPEN FILES
Every open file has an offset.
I've read 8000 bytes of that one
MEMORY
heap! stack! shared libraries! the program's binary! mmaped files!
THREADS
sometimes one sometimes LOTS
CAPABILITIES
I have CAP_PTRACE
well I have CAP_SYS_ADMIN
NAMESPACES
I'm in the host network namespace
I have my own namespace!
container process

A multitasking system is one that allows multiple processes to operate seemingly simultaneously without interfering with each other. This magic happens because processes can dynamically change their states depending upon the context in which they (and other competing processes) evolve.

An analogy : We can compare the above situation to cars. To understand traffic management, we must study the progress of cars. On creation, a car is just a piece of metal, ready to be driven, but currently standing in the warehouse of a car factory. This is similar to a program executable passively stored in a file. Somebody has to fill up fuel, start the car, and drive it to the nearest road. Now the car can be compared to a process. There may be many other cars also on the road, they can all move along as soon as they agree to some traffic rules. In this situation, all cars constantly switch between different states. A car can be in any one of the many gear/brake positions it has. Similarly a process can be in any of the states. The car may be actually moving, or may be stationary (just idling) at any given time.

Process states : A process passess through various stages during its life-time, from creation to exit. Each stage can be called as a *process state*. Processes switch between different states using signals and traps coded into the processes [6] .

Signals, traps : *Signals* are software interrupts sent to a program to indicate that an important event has occurred. A *trap* defines and activates handlers to be run when a process receives signals or other special conditions.

1.2 Watching the processes

Linux offers a remarkable tool (in fact many) for monitoring the evolution of processes in a Linux system. We will use `ps` to observe processes. `ps` can be invoked with various options. Use
`ps -e`
to get a list of all processes running on your system. Use
`ps -o pid,state,command`
to display the state of any given command.

Example:

```
  drpartha@vostro3558:~> ps -o pid,state,command
  PID S COMMAND
 2775 R ps -o pid,state,command
```

The first entry (PID) in the above output is a unique number assigned to each process. The second entry (S) in the above output is a single letter mnemonic for the process state. The mnemomic can be one of the following:

```
PROCESS STATE CODES
 R  ready-to-run or 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
 [...]
```

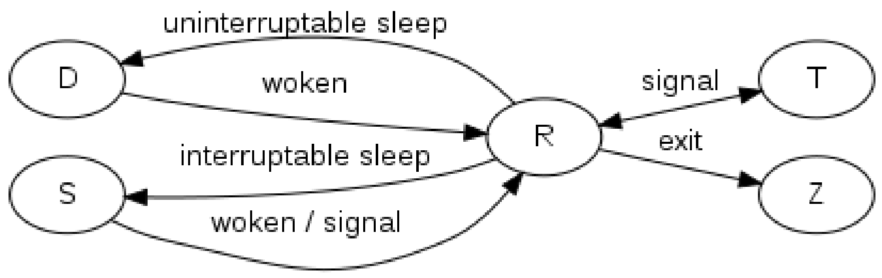
A process starts its life in an R state and finishes after its parent reaps it from the Z "zombie" state (see *Z* state described below).

Use the command `top` to observe the dynamic evolution of all processes and their states. The output will refresh itself automatically and at a pre-defined interval.

In addition, the command `pstree` will show the parent-child relationships of all processes as a hierarchically organised tree.

2 Linux process states

The situation of all cars on the road can be summarised as in the picture below. The figure illustrates the condition/state of each car at any one instant of time. We will use the term *car* or *process* interchangeably in the following discussion.



This is of course a highly simplified diagram, but it demonstrates most of the essential concepts involved [5] . A more elaborate and detailed diagram can be found in [7]. The above diagram shows the states and transitions of only one process. In all modern computers, at any time there may be several processes concurrently. We can make a diagram like the above for each process. Each process will keep switching between the various states depending on the states of other processes. This is similar to the states of a car in a traffic system.

2.1 *R* state

R - The car came to this state from the company warehouse. The R state: ready-to-run/ runnable/running state is when the car is on the road and ready to start moving. A process is born when it is in the memory and is ready to run. In Unix/Linux a process can get created (be born) when it is "spawned" by another process using a "fork" command. The car will actually move, only when conditions are favourable, like when other cars have moved and have given way for our car to move.

There is a subtle difference between a running car and a moving car. The car may be just idling, running but not moving. A process may be in any one of the following states (D, Z, T, S) but not actually executing.
In the case of Linux the process is in the CPU and is a contender for CPU time. When it gets CPU time, from the R state, the process can go to any one of the four other states: $R \to S$, $R \to T$, $R \to Z$, $R \to D$

It will be in any of the above states till the car owner decides to retire the car and withdraw it from the traffic.

2.2 *T* state

T – stopped state, either by a job control signal or because it is being traced. The car is on the road, ready to move, but either its engine is switched off or the brakes have been applied. In Linux, such a condition can occur when a process is being monitored under a symbolic debugger. The process is forced to step from one checkpoint to the next and halts there. It can continue thereafter, only if the next step is requested (by the user).

2.3 *D* state

D – uninterruptible sleep. This happens when I/O operation is going on. In the road traffic analogy, the car is waiting while pedestrians are crossing.

2.4 *S* state

S – sleeping/waiting for an event to arrive. The car is waiting for the signal light to turn green. A process can be made to wait for a pre-defined event to occur.

2.5 Z state

Z – zombie. The proces has terminated by executing the *exit* system call. The process no longer exists. It leaves a record containing its PID, an exit code and some timing statistics for the parent process to collect (harvesting by the parent process). The *zombie* state is the final state of a process.

When a process dies on Linux, it isnt all removed from memory immediately its process descriptor stays in memory (the process descriptor only takes a tiny amount of memory). The processs status becomes EXIT_ZOMBIE and the processs parent is notified that its child process has died with the SIGCHLD signal. The parent process is then supposed to execute the wait() system call to read the dead processs exit status and other information. This allows the parent process to get information from the dead process. After wait() is called, the zombie process is completely removed from memory (harvested). However, if a parent process isnt programmed properly and never calls wait(), its zombie children will stick around in memory until they are cleaned up. This is like abandonig your car in the middle of the road. Other cars can still steer around this car.

3 Transitions

3.1 Transitions $R \to T$ and $T \to R$

The car can start moving or may be stopped for some reason. A ready to run program can start executing instructions, or may pause its execution (to resume eventually).

3.2 Transitions $R \to D$ and $D \to R$

3.3 Transitions $R \to S$ and $S \to R$

3.4 Transitions $R \to Z$

The proces has terminated by executing the *exit* system call. The parent process is then supposed to execute the wait() system call to read the dead processs exit status and other information. After wait() is called, the zombie process is completely removed from memory (harvested). The car is no more part of the traffic system.

4 Concluding remarks

This is a very simplified description of a fairly complex subject. The analogy of cars should not be stretched too far.

A more comprehensive and clear guide is strongly recommended and is available at [8].

You may send constructive suggestions and remarks to :
`drpartha@gmail.com`.

References

[1] L. Lamport, LaTeX : A document preparation system, Pub.: Addison Wesly, 1986 (The LaTeX bible)

[2] M Goossens, F Mittelbach, A Samarin, The LaTeX Companion, Pub.: Addison Wesley, 1994. (The LaTeX gospel)

[3] Nicolas Markey, Tame the beast, URL http:// http://tug.ctan.org/tex-archive/info/bibtex/tamethebeast/ttb_en.pdf

[4] Linux Signals Fundamentals

[5] Marek,
Linux process states,
`https://idea.popcount.org/2012-12-11-linux-process-states/`

[6] Tutorialspoint, Unix / Linux Signals and Traps,
`https://www.tutorialspoint.com/unix/unix-signals-traps.htm`

[7] Bach J Maurice, *The design of the Unix operating system*,
Pub.: Prentice Hall.

[8] All You Need To Know About Processes in Linux [Comprehensive Guide],
`https://www.tecmint.com/linux-process-management/`

***