Understanding Linux Process States

Author: Yogesh Babar
Technical Reviewer: Chris Negus
Editor: Allison Pranger
08/31/2012

OVERVIEW

A *process* is an instance of a computer program that is currently being executed. Associated with a process is a variety of attributes (ownership, nice value, and SELinux context, to name a few) that extend or limit its ability to access resources on the computer. During the life of a process, it can go through different states.

To get started with the various states of a process, compare a Linux process to a human being. Every human being has different stages of life. The life cycle begins with the parents giving birth to an offspring (synonymous to a process being *forked* by its parent process).

After birth, humans start living their lives in their surroundings and start using available resources for their survival (synonymous to a process being in a *Running* state). At some point in the life cycle, humans need to wait for something that they must have before they can continue to the next step in their lives (this is synonymous to a process being in the *Sleep* state). And just as every human life must come to an end, every process must also die at some point in time.

This tech brief will help you understand the different states in which a process can be.

UNDERSTANDING PROCESS TYPES

There are different types of processes in a Linux system. These types include user processes, daemon processes, and kernel processes.

User Processes

Most processes in the system are user processes. A user process is one that is initiated by a regular user account and runs in user space. Unless it is run in a way that gives the process special permissions, an ordinary user process has no special access to the processor or to files on the system that don't belong to the user who launched the process.

Daemon Process

A daemon process is an application that is designed to run in the background, typically managing some kind of ongoing service. A daemon process might listen for an incoming request for access to a service. For example, the `httpd` daemon listens for requests to view web pages. Or a daemon might be intended to initiate activities itself over time. For example, the `crond` daemon is designed to launch cron jobs at preset times.

Although daemon processes are typically managed as services by the root user, daemon processes often run as non-root users by a user account that is dedicated to the service. By running daemons under different user accounts, a system is better protected in the event of an attack. For example, if an attacker were to take over the `httpd` daemon (web server), which runs as the Apache user, it would give the attacker no special access to files owned by other users (including root) or other daemon processes.

Systems often start daemons at boot time and have them run continuously until the system is shut down. Daemons can also be started or stopped on demand, set to run at particular system run levels, and, in some cases, signaled to reload configuration information on the fly.

Kernel Processes

Kernel processes execute only in kernel space. They are similar to daemon processes. The primary difference is that kernel processes have full access to kernel data structures, which makes them more powerful than daemon processes that run in user space.

Kernel processes also are not as flexible as daemon processes. You can change the behavior of a daemon process by changing configuration files and reloading the service. Changing kernel processes, however, may require recompiling the kernel.

INTRODUCING SYSTEM STATES

When a process is created, the system assigns it a state. The state field of the process descriptor describes the current state of the process. The value of the state field is usually set with a simple assignment, as shown in this example:

```
p->state = TASK_RUNNING
```

Here, **`p`** stands for Process, **`state`** is the flag, and **`TASK_RUNNING`** indicates that process is currently running or ready to run.

Most processes are in one of the following two states:

- A process that is on the CPU (a running process)
- A process that is off the CPU (a not-running process)

Only one process can run at a time on a single CPU. All other processes have to wait or be in some other state. This is why a process that is not running appears in a different state. States include the following:

- Runnable state
- Sleeping state
- Uninterruptable sleep state
- Defunct or Zombie state

In this example of a common process life cycle, the process is as follows:

1. Born or forked
2. Ready to run or runnable
3. Running in user space or running in kernel space
4. Blocked, Waiting, Sleeping, in an Interruptable sleep, or in an Uninterruptable sleep
5. The process is sleeping, but it is present in main memory
6. The process is sleeping, but it is present in secondary memory storage (swap space on disk)
7. Terminated or stopped

Figure 1 illustrates these different process states. The next sections detail each of these points in a process life cycle.

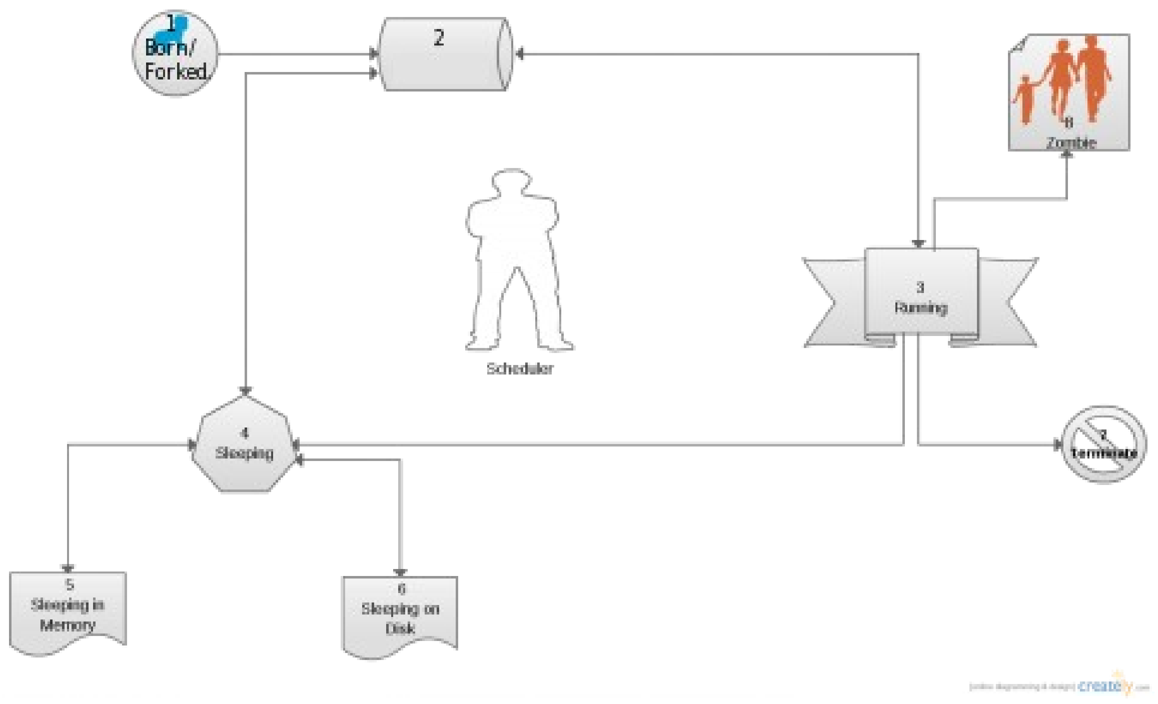

Figure 1: Processes through different states between birth and termination

Starting a Process (Born or Forked)

As per the `fork(2)` man page, **`fork`** creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process (referred to as the parent), except for the following points:

- The child has a unique process ID (see the `setpgid(2)` man page).
- The child's parent process ID is set to the parent's process ID.
- The child does not inherit its parent's memory locks (see `mlock(2)` and `mlockall(2)` man pages).
- The resource utilizations of the child process (see the `getrusage(2)` man page) and CPU time counters (see the `times(2)` man page) are reset to zero.
- The child's set of pending signals is initially empty (see the `sigpending(2)` man page).
- The child does not inherit semaphore adjustments from its parent (see the `semop(2)` man page).
- The child does not inherit record locks from its parent (see the `fcntl(2)` man page).
- The child does not inherit timers from its parent (see the `setitimer(2)`, `alarm(2)`, and `timer_create(2)` man pages).
- The child does not inherit outstanding asynchronous I/O operations from its parent (see the `aio_read(3)` and `aio_write(3)` man pages).
- The child does it inherit any asynchronous I/O contexts from its parent (see the `io_setup(2)` man page).

Running State

The most precious resource in the system is the CPU. The process that is executing and using the CPU at a particular moment is called a *running* process. You can run the **`ps`** and **`top`** commands to see the state of each process. If a process is running, the Running state is shown as R in the state field.

Let's see how a process reaches a Running state. When you fire off a command such as **`ls`**, a shell (bash) searches the directories in the search path stored in the PATH environment variable to find where the **`ls`** command is located. Once the **`ls`** file is found, the shell clones itself using the forking method mentioned earlier, and then the new child process replaces the binary image it was executing (the shell) with the **`ls`** command's executable binary image.

These running processes run in the following spaces:

- In user space
- In system space

The state flag is as follows:

```
p->state = TASK_RUNNING
```

A CPU can execute either in kernel mode or in user mode. When a user initiates a process, the process starts working in user mode. That user mode process does not have access to kernel data structures or algorithms. Each CPU type provides special instructions to switch from user mode to kernel mode. If a user-level process wants to access kernel data structures or algorithms, then it requests that information through system calls that deal with the file subsystem or the process control subsystem. Examples of these system calls include:

- **File subsystem system calls**: **`open()`**, **`close()`**, **`read()`**, **`write()`**, **`chmod()`**, and **`chown()`**
- **Process control system calls**: **`fork()`**, **`exec()`**, **`exit()`**, **`wait()`**, **`brk()`**, and **`signal()`**

When the kernel starts serving requests from user-level processes, the user-level process enters into kernel space. From the 14th and 15th field in the **`/proc/<`*`pid`*`>/stat`** file (where <*pid*> represents the process ID of the process that interests you), you can see how much time a process spent in user mode and in system mode, respectively. Here is how the `proc(5)` man page describes those two fields:

```
utime %lu

     Amount of time that this process  has  been  scheduled  in  user  mode,
     measured  in  clock ticks  (divide  by sysconf(_SC_CLK_TCK).   This
     includes  guest time, guest_time (time spent running a virtual CPU, see
     below), so that applications that are not aware of the guest time field do
     not lose that time from their calculations.

stime %lu

     Amount of time that this process has been  scheduled  in  kernel  mode,
     measured  in  clock ticks  (divide  by sysconf(_SC_CLK_TCK).
```

The **`top`** command's `Cpu` line shows the overall percentage of CPU work in user mode (`us`) and system mode (`sy`):

```
top - 12:27:25 up  2:51,  4 users,  load average: 4.37, 3.64, 3.44
Tasks: 194 total,   2 running, 192 sleeping,   0 stopped,   0 zombie
Cpu(s): 57.0%us,  1.3%sy,  0.0%ni, 41.1%id,  0.0%wa,  0.4%hi,  0.1%si,  0.0%st
```

Runnable State

When a process is in a Runnable state, it means it has all the resources it needs to run, except that the CPU is not available. The Runnable state of this process is shown as R in **ps** output.

Consider a example. A process is dealing with I/O, so it does not immediately need the CPU. When the process finishes the I/O operation, a signal is generated to the CPU and the scheduler keeps that process in the run queue (the list of ready-to-run processes maintained by the kernel). When the CPU is available, this process will enter into Running state.

The state flag is as follows:

```
p->state = TASK_RUNNING
```

Sleeping State

A process enters a Sleeping state when it needs resources that are not currently available. At that point, it either goes voluntarily into Sleep state or the kernel puts it into Sleep state. Going into Sleep state means the process immediately gives up its access to the CPU.

When the resource the process is waiting on becomes available, a signal is sent to the CPU. The next time the scheduler gets a chance to schedule this sleeping process, the scheduler will put the process either in Running or Runnable state.

Here is an example of how a login shell goes in and out of sleep state:

- You type a command and the shell goes into Sleep state and waits for an event to occur.
- The shell process sleeps on a particular wait channel (WCHAN).
- When an event occurs, such as an interrupt from the keyboard, every process waiting on that wait channel wakes up.

To find out what wait channels processes are waiting on for your system, type **`ps -l`** (to see processes associated with the current shell) or **`ps -el`** (to see all processes on the system). If a process is in Sleep state, the WCHAN field shows the system call that the process is waiting on.

Sometimes processes go into Sleep state for a particular amount of time. The Linux kernel uses the **`sleep()`** function, which takes a time value as a parameter that specifies the minimum amount of time (in seconds that the process is set to sleep before resuming execution). This causes the CPU to suspend the process and continue executing other processes until the sleep cycle has finished. When the sleep cycle ends, the scheduler pushes the process to a Ready-to-Run state. When the CPU gets free time, the process goes into Running state.

Some processes never terminate: they keep going into Sleep state at the start of each cycle, waiting for some event to be triggered. Once the event occurs, these processes move to a Running or Runnable state, then return to the Sleep state.

Consider a process that has been forked. If there is enough main memory available to meet what the process needs, the process will enter into Ready-to-Run state (using main memory). But suppose the CPU is busy with some other process. In that case, the process ends up in Ready-to-Run state so it can become a running process when the CPU becomes available.

Now, suppose the process needs some resource that is currently not available. Then the process is taken off of the CPU and the scheduler marks it as a Sleeping state process. If there is not enough main memory, then the swapper process swaps out this newly created process. This time, it sleeps in main memory.

There are two types of sleep states: Interruptible and Uninterruptable sleep states.

Interruptible Sleep State

An Interruptible sleep state means the process is waiting either for a particular time slot or for a particular event to occur. Once one of those things occurs, the process will come out of Interruptible sleep. Output from the **ps** command will show as S in the state field and the process state flag will be as follows:

```
p->state = TASK_INTERRUPTABLE
```

Uninterruptible Sleep State

An Uninterruptible sleep state is one that won't handle a signal right away. It will wake only as a result of a waited-upon resource becoming available or after a time-out occurs during that wait (if the time-out is specified when the process is put to sleep).

The Uninterruptible state is mostly used by device drivers waiting for disk or network I/O. When the process is sleeping uninterruptibly, signals accumulated during the sleep are noticed when the process returns from the system call or trap. In Linux systems. the command **`ps -l`** uses the letter D in the state field (S) to indicate that the process is in an Uninterruptible sleep state. In that case, the process state flag is set as follows:

```
p->state = TASK_UNINTERRUPTABLE
```

LEARN MORE: Read more about D states in the Red Hat Knowledgebase: https://access.redhat.com/knowledge/solutions/59989/

Terminate/Stop State

Processes can end when they call the exit system themselves or receive signals to end. When a process runs the exit system call, it releases its data structures, but it does not release its slot in the process table. Instead, it sends a SIGCHLD signal to the parent. It is up to the parent process to release the child process slot so that the parent can determine if the process exited successfully.

Between the time when the process terminates and the parent releases the child process, the child enters into what is referred to as a Zombie state. A process can remain in a Zombie state if the parent process should die before it has a chance to release the process slot of the child process. The reason you cannot kill a Zombie process is that you cannot send a signal to the process to kill it as the process no longer exists.

A Zombie process has a Z in the state field from the output of the **ps** command. The process state flag is:

```
P->state = TASK_ZOMBIE
```

SUMMARY

From the time a process is born to when it is terminated, the process proceeds through various states. Those states can include Runnable, Running, Sleeping (in memory and on disk), and Zombie states. The output of the **`ps -el`** command lets you see the various states associated with each process on your system.