

RHSA (Admin 1) Red Hat Enterprise Linux 8 Essentials

Prepared by Eng. Taghreed Elsayed

Lecture 6

جامعة الجميع الذكية

تسعى لتكون الكيان الأول الذي يتبنى مبدأ المجتمع التعليمي
التعاوني ويقدم العلم مقابل العمل لتنمية وبناء المجتمعات.

مزيد من التفاصيل

Monitoring and Managing Linux Processes



جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY

WHAT IS A PROCESS?

- ❑ A process is a running instance of a launched, executable program.
- ❑ A process consists of:
 - an address space of allocated memory,
 - security properties including ownership credentials and privileges,
 - one or more execution threads of program code, and the process state.
- ❖ **The environment of a process includes:**
 - local and global variables,
 - a current scheduling context, and allocated system resources, such as file descriptors and network ports.



- An existing (parent) process duplicates its own address space (fork) to create a new (child) process structure.
- Every new process is assigned a unique process ID (PID) for tracking and security.
- The PID and the parent's process ID (PPID) are elements of the new process environment.
- Any process may create a child process. All processes are descendants of the first system process, which is systemd(1) on a Red Hat Enterprise Linux 7 system.

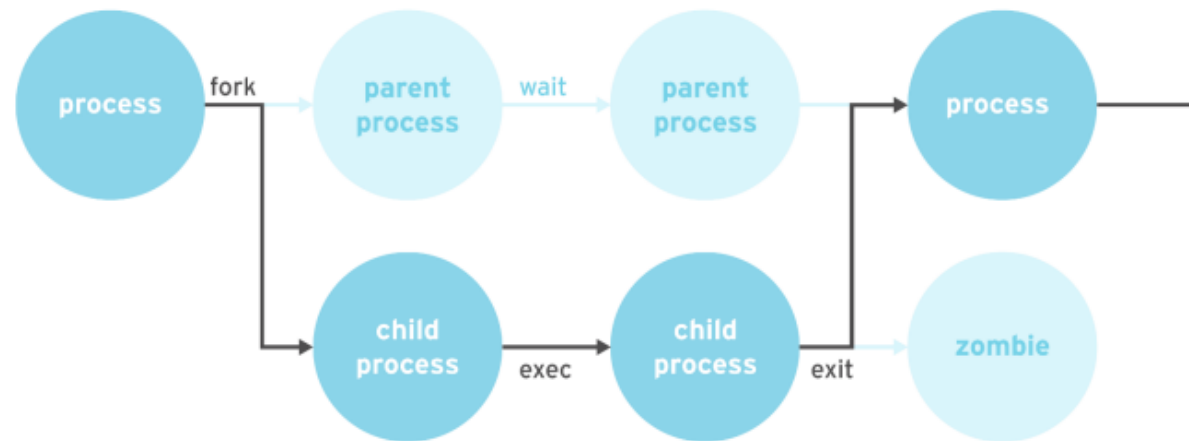


Figure 7.1: Process life cycle

PROCESS STATES

- ❑ In a multitasking operating system, each CPU (or CPU core) can be working on one process at a single point in time.
- ❑ As a process runs, its immediate requirements for CPU time and resource allocation change.
- ❑ Processes are assigned a state, which changes as circumstances require.

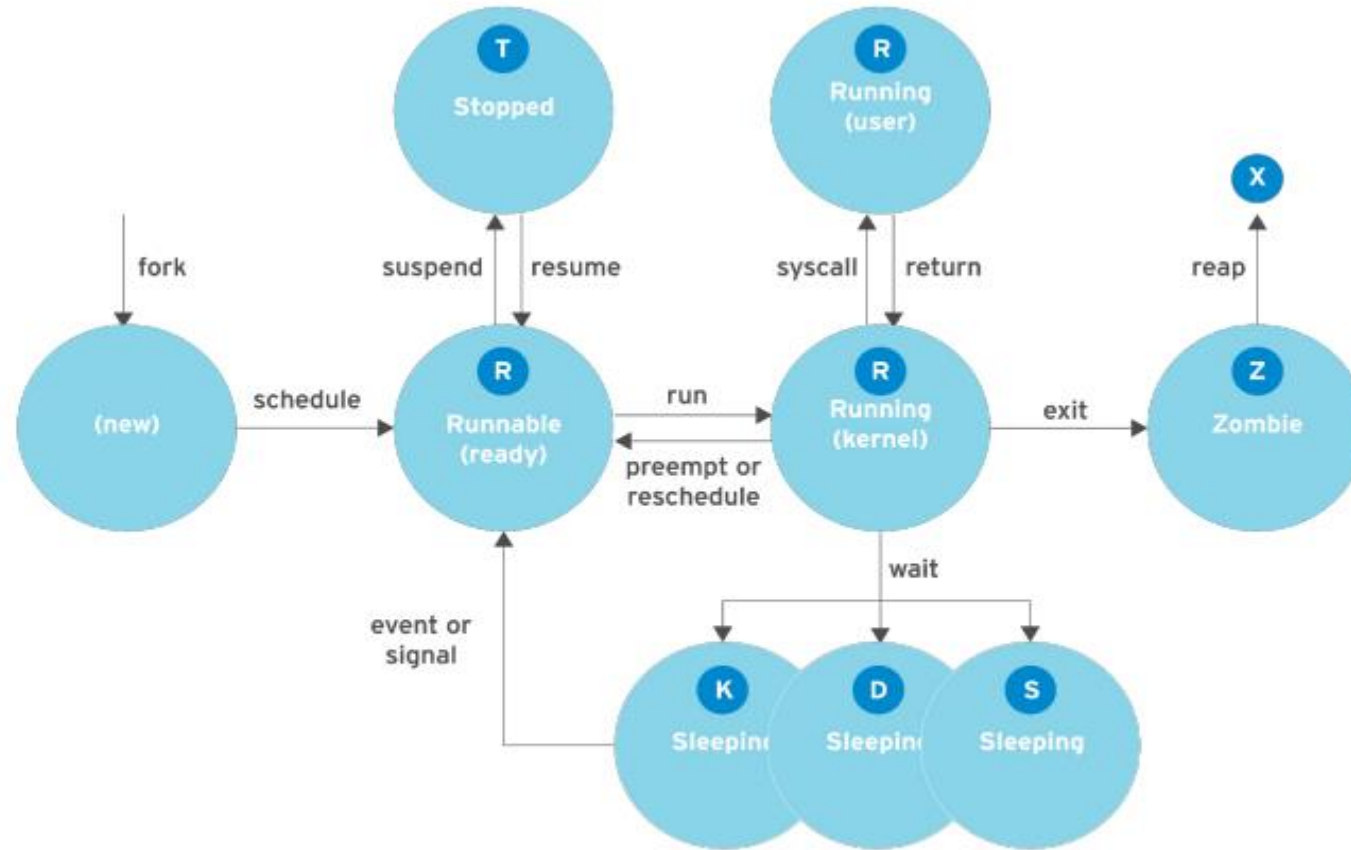


Figure 7.2: Linux process states

Linux process states

NAME	FLAG	KERNEL-DEFINED STATE NAME AND DESCRIPTION
Running	R	TASK_RUNNING: The process is either executing on a CPU or waiting to run. Process can be executing user routines or kernel routines (system calls), or be queued and ready when in the <i>Running</i> (or <i>Runnable</i>) state.
Sleeping	S	TASK_INTERRUPTIBLE: The process is waiting for some condition: a hardware request, system resource access, or signal. When an event or signal satisfies the condition, the process returns to <i>Running</i> .
	D	TASK_UNINTERRUPTIBLE: This process is also <i>Sleeping</i> , but unlike S state, will not respond to delivered signals. Used only under specific conditions in which process interruption may cause an unpredictable device state.

NAME	FLAG	KERNEL-DEFINED STATE NAME AND DESCRIPTION
	K	TASK_KILLABLE: Identical to the uninterruptible D state, but modified to allow the waiting task to respond to a signal to be killed (exited completely). Utilities frequently display <i>Killable</i> processes as D state.
Stopped	T	TASK_STOPPED: The process has been <i>Stopped</i> (suspended), usually by being signaled by a user or another process. The process can be continued (resumed) by another signal to return to <i>Running</i> .
	T	TASK_TRACED: A process that is being debugged is also temporarily <i>Stopped</i> and shares the same T state flag.
Zombie	Z	EXIT_ZOMBIE: A child process signals its parent as it exits. All resources except for the process identity (PID) are released.
	X	EXIT_DEAD: When the parent cleans up (<i>reaps</i>) the remaining child process structure, the process is now released completely. This state will never be observed in process-listing utilities.

LISTING PROCESSES

- ❑ The **ps command** is used for listing current processes.
- ❑ The command can provide detailed process information, including:
 - the **user identification (UID)** which determines process privileges,
 - the unique process identification **(PID)**.
 - the CPU and real time already expended
 - how much memory the process has allocated in various locations
 - the location of process **STDOUT**, known as the controlling terminal, and the current process state.



- ❑ A common display listing (options aux) displays all processes, with columns in which users will be interested, and includes processes without a controlling terminal.
- ❑ A long listing (options lax) provides more technical detail, but may display faster by avoiding the username lookup.
- ❑ The similar UNIX syntax uses the options -ef to display all processes.

```
[student@serverX ~]$ ps aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.1	0.1	51648	7504	?	Ss	17:45	0:03	/usr/lib/systemd/s
syst										

CONTROLLING JOBS

- JOBS AND SESSIONS Job control is a feature of the shell which allows a single shell instance to run and manage multiple commands. A job is associated with each pipeline entered at a shell prompt.
- All processes in that pipeline are part of the job and are members of the same process group. (If only one command is entered at a shell prompt, that can be considered to be a minimal "pipeline" of one command.
- That command would be the only member of that job.) Only one job can read input and keyboard-generated signals from a particular terminal window at a time.
- Processes that are part of that job are foreground processes of that controlling terminal.



- ❖ The `ps` command will show the device name of the controlling terminal of a process in the TTY column.
- ❖ Some processes, such as system daemons, are started by the system and not from a shell prompt.
- ❖ These processes do not have a controlling terminal, are not members of a job, and can not be brought to the foreground.
- ❖ The `ps` command will display a question mark (?) in the TTY column for these processes.
- ❖ **RUNNING JOBS IN THE BACKGROUND**
- ❖ Any command or pipeline can be started in the background by appending an ampersand (&) to the end of the command line.
- ❖ The bash shell displays a job number (unique to the session) and the PID of the new child process.
- ❖ The shell does not wait for the child process and redisplay the shell prompt.

```
[student@serverX ~]$ sleep 10000 &  
[1] 5947  
[student@serverX ~]$
```

The bash shell tracks jobs, per session, in a table displayed with the jobs command.

```
[student@serverX ~]$ jobs  
[1]+  Running                  sleep 10000 &  
[student@serverX ~]$
```

A background job can be brought to the foreground by using the **fg** command with its job ID (%*job number*).

```
[student@serverX ~]$ fg %1  
sleep 10000  
—
```



→ In the preceding example, the sleep command is now running in the foreground on the controlling terminal.

→ The shell itself is again asleep, waiting for this child process to exit. To send a foreground process to the background, first press the keyboard-generated suspend request (Ctrl+z) on the terminal.

```
sleep 10000
^Z
[1]+  Stopped                  sleep 10000
[student@serverX ~]$
```

The job is immediately placed in the background and is suspended



- ❑ The `ps j` command will display information relating to jobs. The PGID is the PID of the process group leader, normally the first process in the job's pipeline.
- ❑ The SID is the PID of the session leader, which for a job is normally the interactive shell that is running on its controlling terminal. Since the example sleep command is currently suspended, its process state is T.

```
[student@serverX ~]$ ps j
PPID   PID   PGID   SID TTY        TPGID STAT   UID    TIME COMMAND
2764   2768   2768   2768 pts/0      6377 Ss     1000    0:00 /bin/bash
2768   5947   5947   2768 pts/0      6377 T      1000    0:00 sleep 10000
2768   6377   6377   2768 pts/0      6377 R+     1000    0:00 ps j
[student@serverX ~]$
```

To start the suspended process running in the background, use the **bg** command with the same job ID.

```
[student@serverX ~]$ bg %1
[1]+  sleep 10000 &
```

```
[student@serverX ~]$
```

The shell will warn a user who attempts to exit a terminal window (session) with suspended jobs. If the user tries exiting again immediately, the suspended jobs are killed.

KILLING PROCESSES

- ☐ **PROCESS CONTROL USING SIGNALS** A signal is a software interrupt delivered to a process.
- ☐ Signals report events to an executing program. Events that generate a signal can be an error, external event (e.g., I/O request or expired timer), or by explicit request (e.g., use of a signal-sending command or by keyboard sequence).
- ☐ The following table lists the fundamental signals used by system administrators for routine process management. Refer to signals by either their short (HUP) or proper (SIGHUP) name.

Fundamental process management signals

SIGNAL NUMBER	SHORT NAME	DEFINITION	PURPOSE
1	HUP	Hangup	Used to report termination of the controlling process of a terminal. Also used to request process reinitialization (configuration reload) without termination.
2	INT	Keyboard interrupt	Causes program termination. Can be blocked or handled. Sent by pressing INTR key combination (Ctrl+c).
3	QUIT	Keyboard quit	Similar to SIGINT , but also produces a process dump at termination. Sent by pressing QUIT key combination (Ctrl+\).
9	KILL	Kill, unblockable	Causes abrupt program termination. Cannot be blocked, ignored, or handled; always fatal.
15 <i>default</i>	TERM	Terminate	Causes program termination. Unlike SIGKILL , can be blocked, ignored, or handled. The polite way to ask a program to terminate; allows self-cleanup.

18	CONT	Continue	Sent to a process to resume if stopped. Cannot be blocked. Even if handled, always resumes the process.
19	STOP	Stop, unblockable	Suspends the process. Cannot be blocked or handled.
20	TSTP	Keyboard stop	Unlike SIGSTOP , can be blocked, ignored, or handled. Sent by pressing SUSP key combination (Ctrl+z).

Commands for sending signals by explicit request

- ❑ Users signal their current foreground process by pressing a keyboard control sequence to suspend (Ctrl+z), kill (Ctrl+c), or core dump (Ctrl+\) the process.
- ❑ To signal a background process or processes in a different session requires a signal-sending command.
 - The **kill** command sends a signal to a process by ID. Despite its name, the kill command can be used for sending any signal, not just those for terminating programs.

```
[student@serverX ~]$ kill PID[student@serverX ~]$ kill -signal PID[student@serverX ~]$ kill -l
```

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP

-- output truncated --



MONITORING PROCESS ACTIVITY

- Linux counts not only processes, but threads individually, as separate tasks. CPU request queues for running threads (nr_running) and threads waiting for I/O resources (nr_iowait) reasonably correspond to process states R (Running) and D (Uninterruptable Sleeping). Waiting for I/O includes tasks sleeping for expected disk and network responses.

```
[student@serverX ~]$ grep "model name" /proc/cpuinfo
model name : Intel(R) Core(TM) i5 CPU          M 520  @ 2.40GHz
model name : Intel(R) Core(TM) i5 CPU          M 520  @ 2.40GHz
model name : Intel(R) Core(TM) i5 CPU          M 520  @ 2.40GHz
model name : Intel(R) Core(TM) i5 CPU          M 520  @ 2.40GHz
[student@serverX ~]$ grep "model name" /proc/cpuinfo | wc -l
4
```

→ use the top command to dynamically view, sort, and stop processes.

```
[student@serverX ~]$  
top
```

- ▶ 2. In the left terminal, determine the number of logical CPUs on this virtual machine.

```
[student@serverX ~]$ grep "model name" /proc/cpuinfo | wc -l  
1
```

- ▶ 3. In the left terminal, run a single instance of the **process101** executable.

```
[student@serverX ~]$  
process101
```

4. In the right terminal, observe the top display. Use the single keystrokes l, t, and m to toggle the load, threads, and memory header lines. After observing this behavior, ensure that all headers are displaying.
5. Note the process ID (PID) for process101. View the CPU percentage for the process, which is expected to hover around 25% or 30%. View the load averages. On a single-CPU virtual machine, for example, the one-minute load average is currently less than a value of 1. The value observed may be affected by resource contention from another virtual machine or the virtual host.
- ▶ 6. In the left terminal, run a second instance of **process101**.

```
[student@serverX ~]$  
process101
```

7. In top, note the process ID (PID) for the second process101. View the CPU percentage for the process, also expected to hover around 25% or 30%. View the one-minute load average again, which may still be less than 1. Wait up to one minute to allow the calculation to adjust to the new workload.

- 8. In the left terminal, run a third instance of **process101**.

```
[student@serverX ~]$  
process101
```

9. In **top**, note the process ID (PID) for the third **process101**. View the CPU percentage for the process, again expected to hover around 25% or 30%. View the one-minute load average again, which now is expected to be above 1. Wait up to one minute to allow the calculation to again adjust to the new workload.
10. Optional: If this virtual machine has more than one logical CPU, slowly start additional **process101** instances until the one-minute load average equals or exceeds the number of logical CPUs. Divide the load average value by the number of CPUs to determine the estimated load average per CPU.
11. When finished observing the load average values, terminate each of the **process101** processes from within **top**

- 11.1. Press k. Observe the prompt below the headers and above the columns.
- 11.2. Type the PID for one of the process101 instances. Press Enter.
- 11.3. Press Enter again to use the default SIGTERM signal 15. Confirm that the selected process is no longer observed in top. If the PID still remains, repeat these terminating steps, substituting SIGKILL signal 9 when prompted.
12. Repeat the previous step for each remaining process101 instance. Confirm that no process101 instances remain in top.
13. In the right window, press q to exit top. Close extra terminal windows.



جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY

