Linux Basics 4: Processes

To work effectively on a Linux system, it is necessary to understand a number of the basic aspects of Linux:

- The Filesystem (directories and files)
- Users, Groups, and File Permissions
- **Processes**
  - **what is a process**
  - **process attributes**
  - **process ID (PID)**
  - **environment variables**
  - **exit status**
  - **signals**
  - **process commands**

Processes

The term **process** is used to refer to a program (e.g., command) that is *being executed* (i.e., is running).

Every process has a set of **attributes**, including:

- **process ID** (**PID**) and **parent process ID** (**PPID**)
- **real UID** (**RUID**) and **effective UID** (**EUID**)
- **real GID** (**RGID**) and **effective GID** (**EGID**)
- address space, program counter, run state, etc.
- open files/devices (**file descriptor table**)
- **umask**, **nice** value (priority), resource limits, etc.
- **signal** masks and handlers, alarms/timers, etc.

Processes (contd.)

Every process (except the first) is *created by another process*, which is called its **parent**.

The first process, **init**, is created by the kernel boot process and has PID of 1 (one).

Each process will be in one of several states:

- running/runnable (running or in run queue)
- sleeping/suspended (e.g., waiting for I/O device)
- defunct/zombie (terminated, waiting for collection)
- stopped (job control)

Threads

Each process will have at least one **thread of execution** (one sequence of statements in the program that are being executed).

Linux supports **OS threads**, so each process can in fact contain *multiple threads*.

Most Linux commands/tools show only processes by default; options must be used to show threads.

When viewing threads, one must know these terms:

- **thread ID** (**TID**): unique system-wide ID for an *os thread*
- **thread group**: the set of OS threads in a process
- **thread group ID** (**TGID**): ID for a thread group, is the same as the PID of the containing process

Environment Variables

One important process attribute is the **environment**.

The environment is a *list of variable-value pairs*, which are coded as **C strings** with the format: `"variable=value"`.

These variables are referred to as **environment variables**.

Environment variables are given *uppercase* names by convention (to distinguish them from normal **shell variables**).

The environment is generally *inherited* when a subprocess is created (via `fork()`).

A standard set of environment variables are used to pass basic parameters to all programs.

Environment Variables (contd.)

Key standard environment variables:

PATH – the directories to search for commands

HOME – the current user's home directory

USER,LOGNAME – current username

PWD – current working directory

HOSTNAME – hostname of machine

SHELL – default shell for current user

TERM – terminal type

LANG,LC_* – set of locale related variables.

Exit Status

When a Linux/UNIX process terminates, it returns **exit status** information to the kernel.

This exit status information is supposed to be collected by the process' parent.

One element of the exit status is whether the program had *terminated normally or abnormally*:

- **normal termination**: program called `exit()` (or a related function) or returned from its `main`
- **abnormal termination**: program called `abort()`, received a terminating **signal**, or canceled the thread

Exit Status (contd.)

When a process terminates normally, it must return a *non-negative integer between 0 and 255* to indicate its success/failure status:

- 0 (zero) indicates **success**
- any other value indicates **failure**, with the value possibly representing the reason for failure
- 1 is the default failure return code

Signals

Signals are a mechanism for notifying processes that some *event* has occurred.

They can be considered both as **software interrupts** and as a simple **interprocess communication** (**IPC**) **mechanism**.

Signals can be generated in several ways:

- process/program can cause explicitly (e.g., `kill()`)
- process/program can cause due to error (e.g., **segmentation fault**)
- user can cause by typing *special terminal keys* (e.g., `ctrl-c`)
- user can send using `kill` command

Signals (contd.)

Signals have *default actions* that occur when they are received by a process, which can be one of:

- terminate process
- terminate process and generate **core file**
- ignore signal
- stop process

Processes can also be set to *ignore* or (temporarily) *block* most signals.

However, two signals cannot be ignored/blocked (SIGKILL and SIGSTOP).

Signals (contd.)

Some key signals (for users):

SIGTERM – standard kill signal (but can be ignored/blocked)

SIGKILL – terminate process (cannot be ignored/blocked)

SIGINT – sent by typing **interrupt key** (`ctrl-c`)

SIGQUIT – sent by typing **quit key** (`ctrl-\`)

SIGSEGV – a **segmentation fault** (**segfault**), caused by an illegal memory reference in a program

SIGBUS – a hardware error has occurred

SIGHUP – controlling terminal has disconnected ("hang up")

Key Process Commands

ps – list current process information

- `ps -eF`
- `ps aux`
- `ps -eFT` (show all threads)

pstree – show tree of processes

- `pstree`

top – realtime display of CPU usage by process

- `top`

kill – terminate a process (send it a signal)

- `kill 4521`
- `kill -9 4521` or `kill -SIGKILL 4521`

printenv – print environment variable info

- `printenv PATH`