# Week 12 – Processes

Processes carry out tasks within the operating system. A program is a set of machine code instructions and data stored in an executable image on disk and is, as such, a passive entity; a process can be thought of as a computer program in action.

Recall, in Microsoft Windows, we can view the active processes via Task Manager:

Graphical user interface, application, table

Description automatically generated

Linux works relatively the same way, but you have much higher control over the processes. We use the **ps**  command to do this.

If we look at **man ps** we see there are three sets of options that can be used:

A picture containing text

Description automatically generated

To list all processes, use the **ps** command in Linux: **ps aux**

Note: **a** – outputs all running processes for all users, **u** – adds additional info like memory and cpu usage, process state code, owner of process, **x** – include all processes not executed from the terminal, ie daemons.

A picture containing graphical user interface

Description automatically generated

Another Note: The processes in square brackets are parts of the Kernel that are presented as separate processes. A lot of the others are related to the GUI, if you were running a server with no GUI the list would be much smaller and very manageable.

The **ps** command with **aux** options gives the following output:

* USER = user owning the process
* PID = process ID of the process
  + Unique to each process
  + We’ll use this to interact with process for the most part
* %CPU = It is the CPU time used divided by the time the process has been running. (Example: if a process has been running for 10 minutes, and was only actively using the CPU for one of those ten minutes it would appear as 10% in this column).
* %MEM = Ratio of the process’s resident set size to the physical memory on the machine
* VSZ = virtual memory usage of entire process (in KiB)
* RSS = resident set size, the non-swappable physical memory that a task has used (in KiB)
* TTY = controlling tty (terminal)
* STAT = multi-character process state
* START = starting time or date of the process
* TIME = cumulative CPU time
* COMMAND = command with all it’s arguments

User is the owner of the process. This is important, because that’s where they (the process) get their permissions.

Processes have **pids**, or process identifiers. Every process has only one PID, and every PID only one process. The single unique identifier that we can use to interact with processes.

Processes also have resources and such allotted to them – take a look in **man ps** to see some details there.

\*\*Each process has a process state. These states are signified by the STAT column in **ps aux**. They will list single letters to represent a process’s state.

Process State Codes:

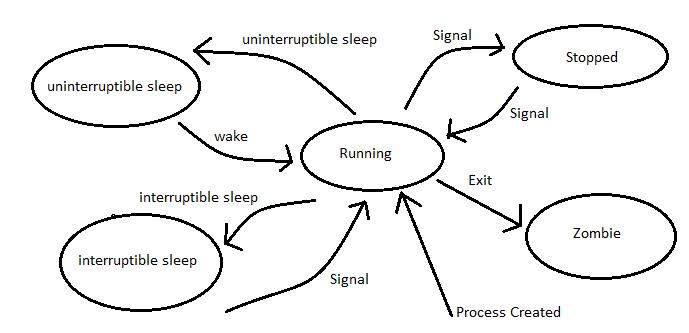
* R
  + The process is running or runnable (on the run queue)
* D
  + Uninterruptible sleep (usually reserved for IO processes)
  + In this state a process cannot be altered, it is waiting for information.
    - This can indicate a problem with a disk.
* S
  + Interruptible sleep (waiting for an event to complete)
* Z
  + Defunct/zombie process.
  + Terminated but not reaped by its parent
* T
  + STopped/terminated, either by a job control signal or because it has been traced

A **Zombie Process** is a process which is dead, takes up no resources, and is waiting.

A **Parent Process** is a process that contains children processes. For example, Firefox may have child processes for each tab.

An **Orphan Process** is a process that has a terminated or stopped parent.

All of this is combined into a sort of state diagram regarding process states:



1. When a process is created (or run), the process will most likely enter the **running** (**R)** state, unless it’s been instructed to enter a different state.
2. We can send different signals to have the process enter different states **(Stopped (T), Uninterruptible sleep (D), Interruptible sleep (S)**.
3. When the process exits, it could potentially be turned into a Zombie process until the parent cleans it up.

You can see this for more information:

<https://idea.popcount.org/2012-12-11-linux-process-states/>

ps -u [username]

Text

Description automatically generated

ps -T – active processes from terminal

Text

Description automatically generated

ps -C [process\_name] (to filter for a specific process)

Text

Description automatically generated

We can run the following command to see current process with a different output:

**ps -o pid,state,command**

Where the -o is the desired output format, with a list of columns we want to display (ie: the pid, state, and command columns)

Text

Description automatically generated

* Here we see two processes running, our bash terminal and the ps -o command that we just ran.

We can use the sleep command to observe process states and manipulate them. Let’s look at **man sleep**:

Text

Description automatically generated

Run the following command: **sleep 100**, then send a stop signal, CTRL-Z:

Text

Description automatically generated

Note that this shortcut (CTRL-Z) can only be used for processes running in the foreground (not background).

We are going to use the kill command next:

**man kill**:

A picture containing graphical user interface

Description automatically generated

Graphical user interface, text

Description automatically generated

man 7 signal:

Text

Description automatically generated

We can send different signals to a certain process by using the **kill** command. The kill command is not for “killing,” it simply sends signals.

Text

Description automatically generated

* Here we see sleep 100 is stopped with a PID of 4104, we can use kill -CONT 4104 to allow it to continue.

Text

Description automatically generated

* Here we reran the sleep 100 command and then used kill -s SIGKILL 4173 to end the process.

We can also have process start in the background by appending “&” symbol to the command

For example, to run the process **sleep** for 100 seconds in background, type the following: **sleep 100 &**

This allows us to still interact with the terminal, as the sleep process will run in the background. Text

Description automatically generated

Let’s create a bash script that runs forever, so that we can interact with it:

Create a new file called **endless\_script.bash** with the following code:

|  |
| --- |
| #! /bin/bash  while true  do  echo Keep running  sleep 3  done |

The script itself is very basic:

* Within a looping structure with no terminating condition
  + Display the text “Keep running”
  + Sleep for 3 seconds
  + Repeat

Change the permissions to 755: **chmod 755 endless\_script.bash**

Text

Description automatically generated

Running the endless\_script.bash we can kill it outright using CTRL-C or stop it using CTRL-Z 🡪 puts it into the sTopped state (T).

Text

Description automatically generated

We can use the **bg** and **fg** commands to changes processes from the background to the foreground. Processes in the background cannot be interrupted via IO:

Text

Description automatically generated

We can add some signal traps to our script. With this, we can intercept any desired signals sent to the process and do other things accordingly:

Let’s edit endless\_script.bash to include the following line: **trap 'echo ignoring puny SIGINT' SIGINT**

Text

Description automatically generated

In this example, we added a trap SIGINT command which will allow the script to intercept the CTRL-C command, and then either ignore it, or do something else (in this case, it’s ignoring it and echoing a string to stdout).

Text

Description automatically generated

Some common signals:

SIGINT – 2 – Terminal

SIGHUP – 1 - Hangup (Daemons often use this to restart)

Interrupt

SIGKILL – 9 – Kill (Can’t be caught/trapped or ignored)

SIGTERM – 15 – Termination

SIGCONT – 18 – Continue executing

SIGSTOP – 20 – Stop Executing

Ideally, signals help processes be better behaved (cleanup can be done on SIGTERM processes, ie, SIGTERM allows processes to end neatly, SIGKILL does not.)

If we kill off a parent process, the process itself will now belong to process id 1. In general, parent processes should wait for their children to die (but it is okay if they die first because **init** will clean up, most likely).

Some other commands to be aware of:

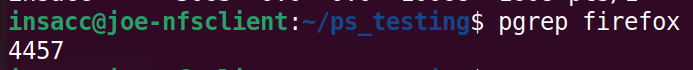
**pgrep**

man pgrep:

Text

Description automatically generated

If we open Firefox, so we have a process to work with, we can **pgrep firefox** to find the PID quickly:



This is much more efficient than using ps -aux.

The **pkill** command lets you kill processes by name:

A screenshot of a computer

Description automatically generated with medium confidence

Two more commands to be aware of **top** and **htop**:

A picture containing text

Description automatically generated

Running **top**:

Text

Description automatically generated with medium confidence

Top options

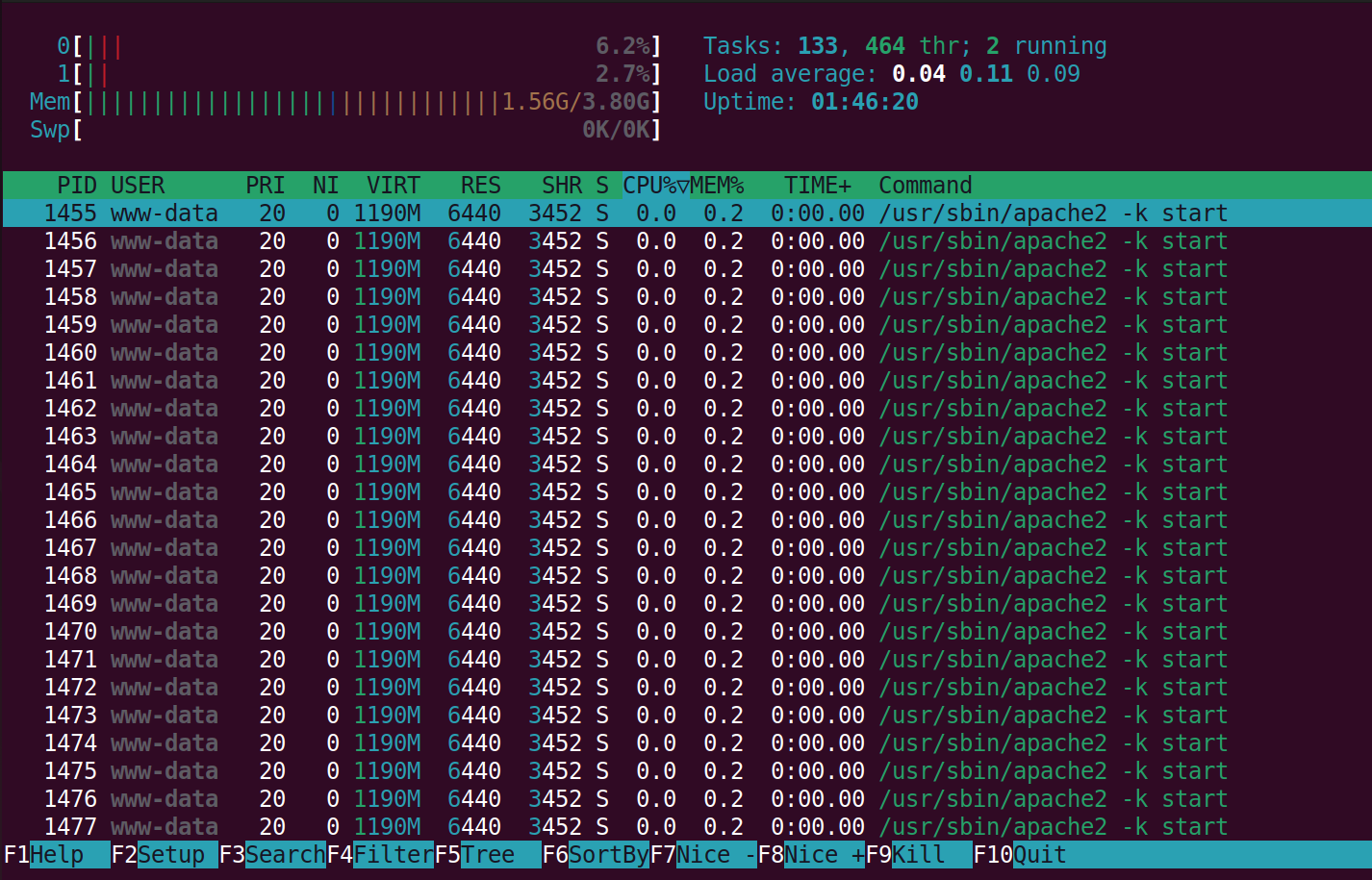
CTRL-C 🡪 Exit (Note: If you CTRL-Z bring to foreground (fg) and then CTRL-C to stop.)

k 🡪 stops & asks for PID to kill, default is the top entry, taken if “enter” is pressed. CTRL+C to bail out.

M 🡪 sorts by memory usage instead of CPU usage.

1 🡪 breaks out the CPU’s separately.

The **htop** command gives a different view (h=hierarchy)



The **pstree** command displays the processes in a hierarchy:

Text

Description automatically generated

**ps -ely** is yet another way to format the **ps** output:

A picture containing graphical user interface

Description automatically generated

**Exercise:**

Create an endless bash script and run it as a foreground process. Stop it by CTRL-Z then show the process state using ps. Kill the job.

Run that same endless bash script, but now as a background process (&). Change it to a foreground process then stop it using CTRL-Z. Kill the job.

Add a trap to the script to trap SIGINT signals.