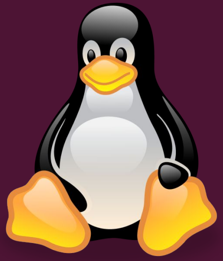
System and Network Engineering - Lecture 7

$ Processes and Signals



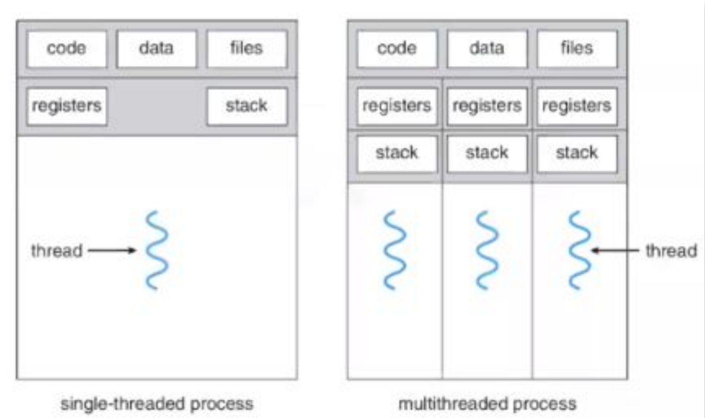
Process

Process is:

- ❑ the instance of a computer program that is being executed by one or many threads.

Process encapsulates:

- ❑ Process memory space:
  - ❑ Text - code of the program
  - ❑ Data - global vars
  - ❑ Heap - dynamically allocated global vars during run time
  - ❑ Stack - function params, return addresses, local variables
- ❑ Operating system
  - ❑ File descriptors (FDs)
  - ❑ Security attributes - process owner and the process' set of permissions (allowable operations).
- ❑ Processor state (context) - reserved registers (e.g. Program Counter that stores next instruction to execute

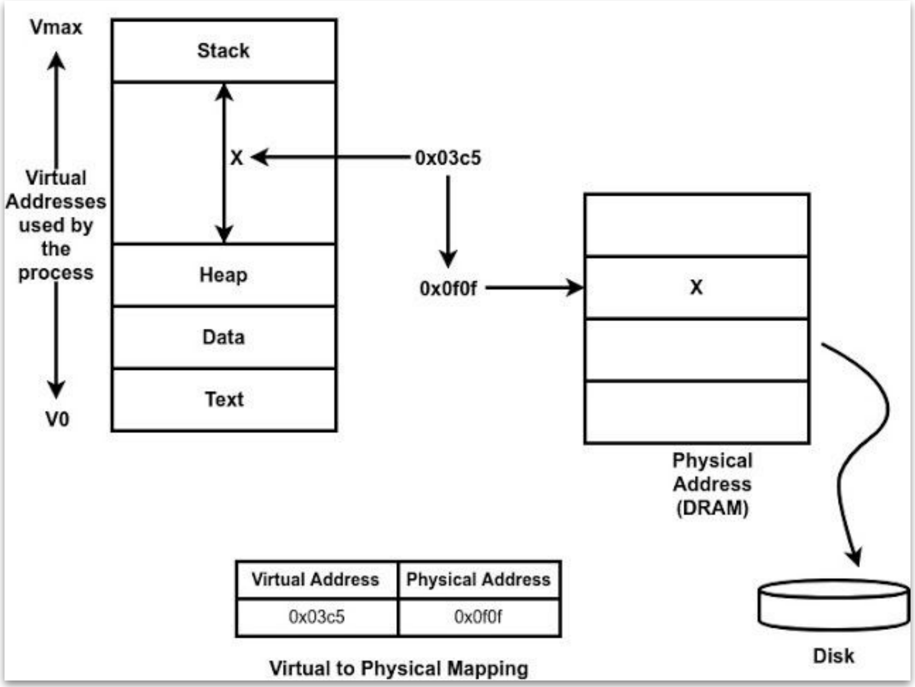


Process

- OS abstraction is used to encapsulate all of the required process data in an address space.

- The address space is defined by a range of address from V0 to some Vmax, and different types of process state will appear in different part of this address space

- The page table maintains the mapping from virtual address to the physical address (RAM or Disk), it also validates whether a particular memory access request by a process is allowed to perform or not.



Virtual Address	Physical Address
0x03c5	0x0f0f

Virtual to Physical Mapping

Process

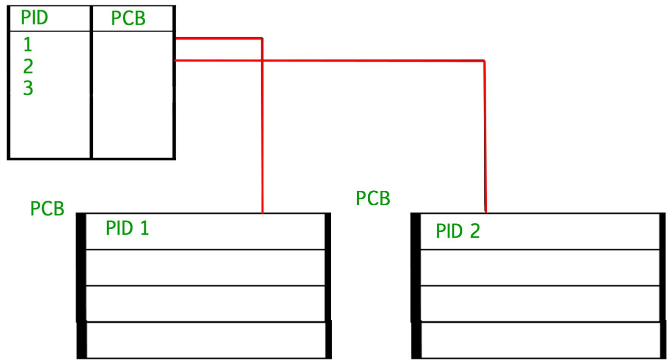
- **Process Control Block (PCB) -** data structure that operating system maintains for every single process to maintain all of the process data for every single process

- **PCB** is created at the very moment a process is created with some initializations like PC points to the first instruction that needs to be executed.

- **PCB** must contain process states like: program counter, stack pointer, all the value of the CPU register, various memory mapping from virtual to physical memory, it may also include a list of open files etc

- During interrupts when OS wants to switch from one process to another - loading and restoring values of fields of PCB to registers (for both processes) is happening, which is also known as **context switching**

- OS maintains pointers to each process's PCB in a **process table** so that it can access the PCB quickly.

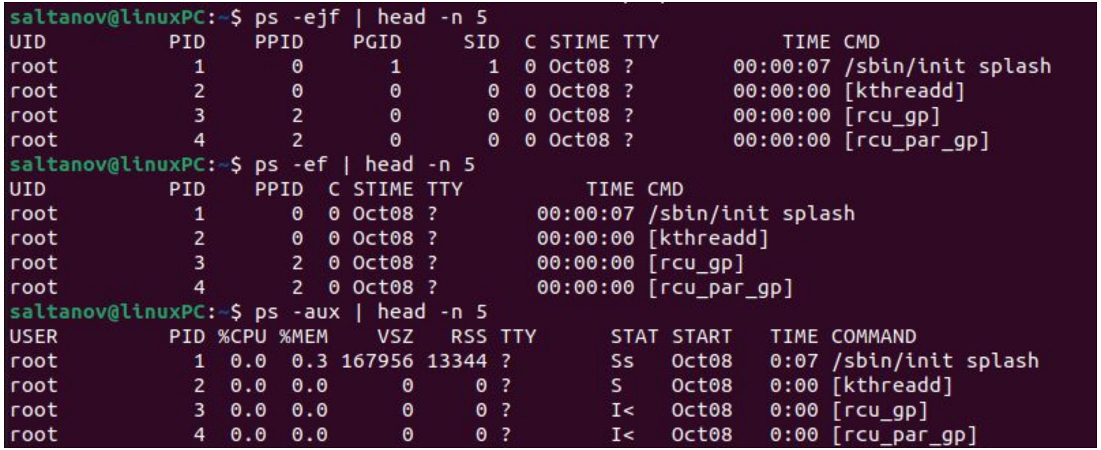


Process table and process control block

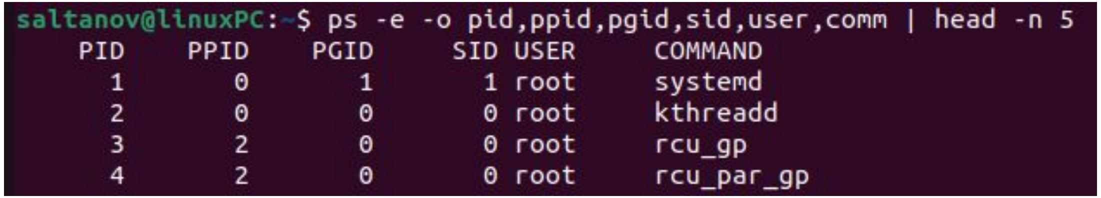
Process

We can view the process context with:

- ❑ `$ps -aux/-ejf`
  - ❑ PID
  - ❑ State
  - ❑ Links
    - ❑ Parent - PPID
    - ❑ Children
  - ❑ Resources usage
  - ❑ Executed time
  - ❑ etc
- ❑ `$top, $htop`
- ❑ `/proc`



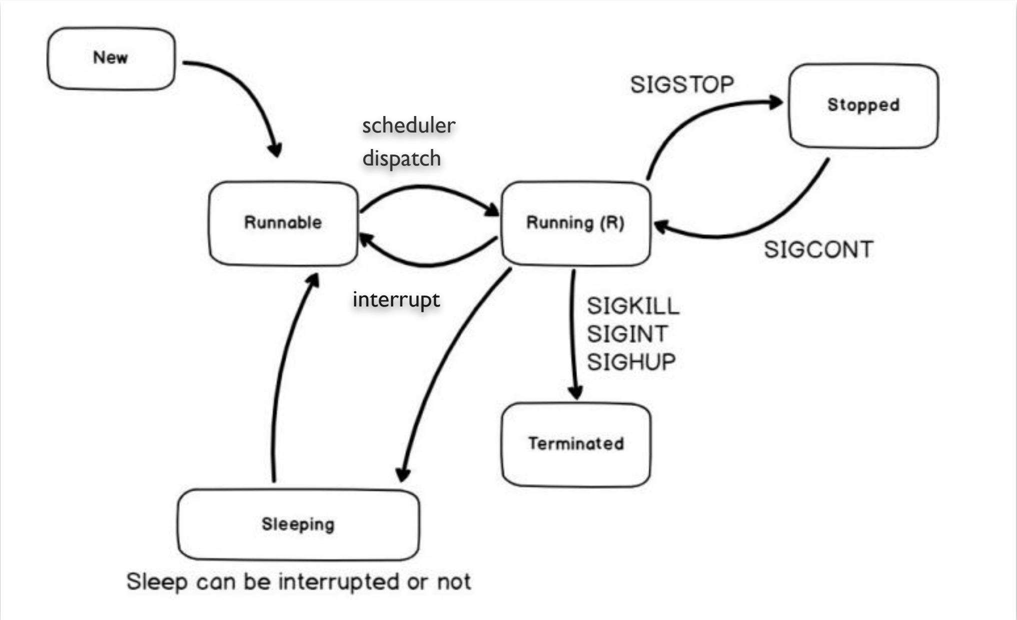
```
saltanov@linuxPC:~$ ps -ejf | head -n 5
UID          PID    PPID    PGID     SID  C STIME TTY          TIME CMD
root           1       0       1       1  0 Oct08 ?        00:00:07 /sbin/init splash
root           2       0       0       0  0 Oct08 ?        00:00:00 [kthreadd]
root           3       2       0       0  0 Oct08 ?        00:00:00 [rcu_gp]
root           4       2       0       0  0 Oct08 ?        00:00:00 [rcu_par_gp]
saltanov@linuxPC:~$ ps -ef | head -n 5
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 Oct08 ?        00:00:07 /sbin/init splash
root           2       0  0 Oct08 ?        00:00:00 [kthreadd]
root           3       2  0 Oct08 ?        00:00:00 [rcu_gp]
root           4       2  0 Oct08 ?        00:00:00 [rcu_par_gp]
saltanov@linuxPC:~$ ps -aux | head -n 5
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.3 167956 13344 ?        Ss   Oct08   0:07 /sbin/init splash
root           2  0.0  0.0      0     0 ?        S    Oct08   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   Oct08   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   Oct08   0:00 [rcu_par_gp]
```



```
saltanov@linuxPC:~$ ps -e -o pid,ppid,pgid,sid,user,comm | head -n 5
    PID    PPID    PGID     SID USER     COMMAND
      1       0       1       1 root     systemd
      2       0       0       0 root     kthreadd
      3       2       0       0 root     rcu_gp
      4       2       0       0 root     rcu_par_gp
```

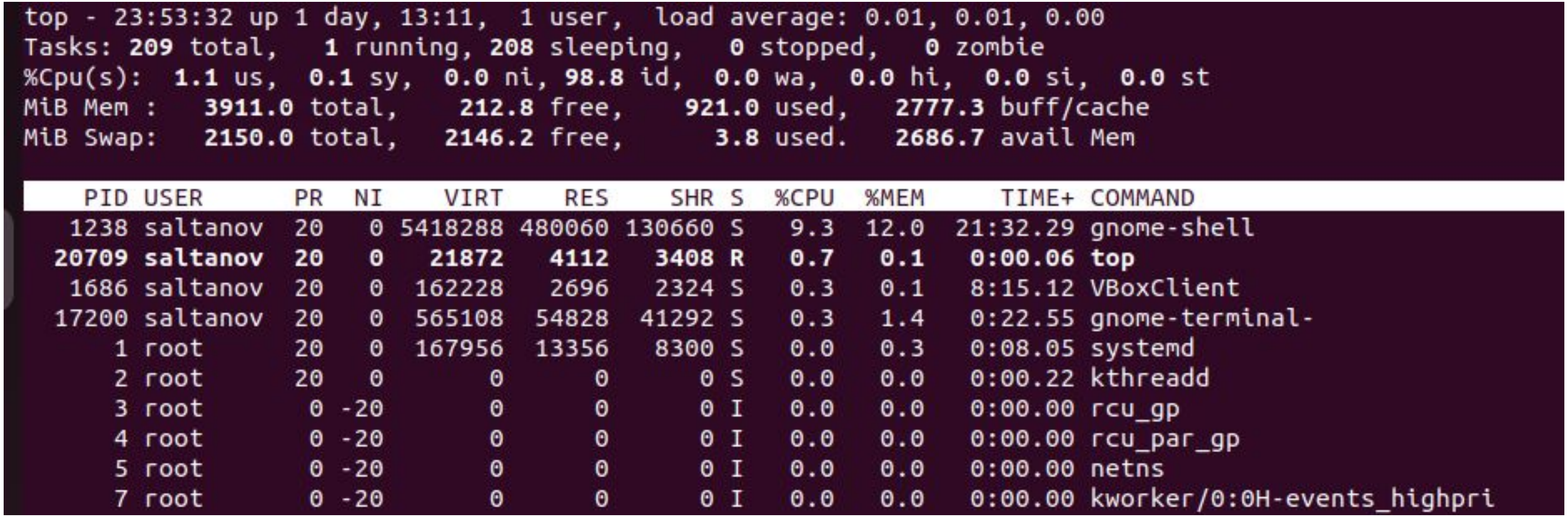
Process states

- ❑ **New -** initial state when process first run (allocation of PCB)
- ❑ **Ready (runnable) -** waiting in a queue to be assigned to a processor
- ❑ **Running -** once assigned by the OS scheduler, the processor executes its instructions
- ❑ **Sleeping (waiting) -** waiting for some events such as I/O, keyboard interrupts etc
- ❑ **Terminated -** once process finishes execution or encounters some errors, then is moved to be removed from the main memory
- ❑ **Stopped -** once process receive the signal to stop its execution



Sleep can be interrupted or not

Process states: Linux example



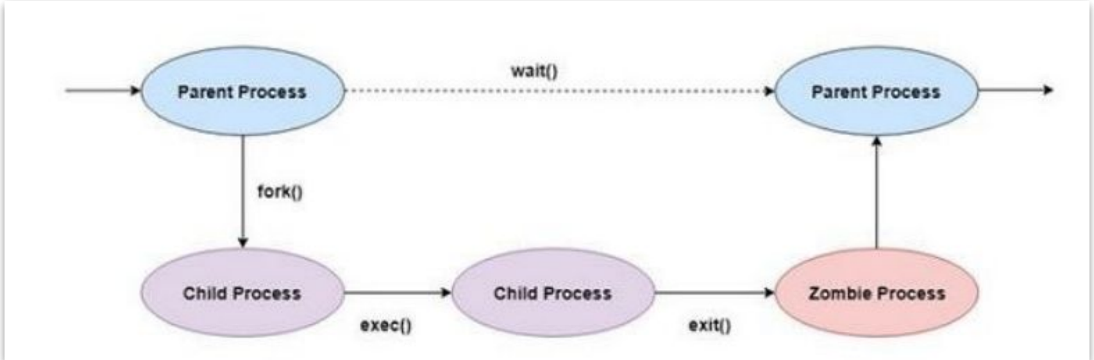
```
top - 23:53:32 up 1 day, 13:11,  1 user,  load average: 0.01, 0.01, 0.00
Tasks: 209 total,   1 running, 208 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.1 us,  0.1 sy,  0.0 ni, 98.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3911.0 total,    212.8 free,    921.0 used,   2777.3 buff/cache
MiB Swap:   2150.0 total,   2146.2 free,      3.8 used.   2686.7 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1238 saltanov  20   0 5418288 480060 130660 S   9.3  12.0  21:32.29 gnome-shell
  20709 saltanov  20   0   21872   4112   3408 R   0.7   0.1   0:00.06 top
   1686 saltanov  20   0  162228   2696   2324 S   0.3   0.1   8:15.12 VBoxClient
  17200 saltanov  20   0  565108  54828  41292 S   0.3   1.4   0:22.55 gnome-terminal-
      1 root      20   0  167956  13356   8300 S   0.0   0.3   0:08.05 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.22 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
      7 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-events_highpri
```

Process states in Linux

- ❑ **There are five Linux process states are defined as:**
  - ❑ **'D' = UNINTERRUPTABLE_SLEEP** is a state where the process is waiting on something. Typically in this state, interrupting could cause some major issues. It is rare to catch a process in this state but when it is, it is usually due to a system call or syscall. For example, mkdir command. It makes a syscall to the kernel and during that enters this state.
  - ❑ **'R' = RUNNING & RUNNABLE -** Most of the time in Linux the distinction does not so matter because it indicates the process is either queued up to run or in the process of running.
  - ❑ **'S' = INTERRRUPTABLE_SLEEP -** going into this process state while waiting for input allows the process to take a back seat and give other processes CPU time. In this state, a process can easily be terminated without issue.
  - ❑ **'T' = STOPPED -** Many applications and console tools/applications allow you to use Control + Z to suspend processes. In this state, the process is put on hold and not responsive.
  - ❑ **'Z' = ZOMBIE -** this is an interim state after a process exits. When a process terminates it is the responsibility of the parent to remove the child process and cleanup the process table. However, when the parent has not done this cleanup, you have a process that is staying

Zombie vs Orphan

- ❑ **Zombie**
  - ❑ Just an entry in process table
  - ❑ Doesn’t consume any resources
  - ❑ Usually means issues with parent
  - ❑ May exhaust PIDs

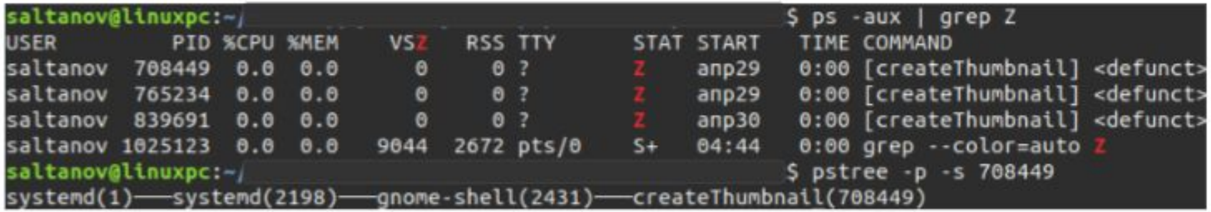


- ❑ **Orphan**
  - ❑ Proces is still running
  - ❑ Parent process terminated earlier
    - ❑ Expectedly
    - ❑ Unexpectedly
  - ❑ Adopted by init system

wait() - are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then terminated the child remains in a "zombie" state

How to kill a process

- ❑ **How to Kill the Sleeping Process**
  - ❑ For an **interruptible sleep** state can be killed with the kill command or “ kill **-9** ” if you want it to happen immediately.
  - ❑ On the other hand the **uninterruptible sleep** state cannot be killed. You can issue a kill to it but that kill will be queued up until it gets out of this state. Alternatively rebooting the system is an option.
  - ❑ Processes stuck in uninterruptible sleep can become very inconsistent if killed and therefore not allowed.
- ❑ **How to Kill the Zombie Process**
  - ❑ First find the Zombie process (3 ways):

1. `$ps wauxf | less` - search for Zombie in the output and find parent
2. 

```
saltanov@linuxpc:~/[redacted]$ ps -aux | grep Z
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
saltanov   708449  0.0  0.0      0     0 ?        Z    anp29   0:00 [createThumbnail] <defunct>
saltanov   765234  0.0  0.0      0     0 ?        Z    anp29   0:00 [createThumbnail] <defunct>
saltanov   839691  0.0  0.0      0     0 ?        Z    anp30   0:00 [createThumbnail] <defunct>
saltanov  1025123  0.0  0.0   9044  2672 pts/0    S+   04:44   0:00 grep --color=auto Z
saltanov@linuxpc:~/[redacted]$ pstree -p -s 708449
systemd(1)───systemd(2198)───gnome-shell(2431)───createThumbnail(708449)
```

3. First, `$ps -aux | grep Z` and then search manually from `$ps -ef` the parent PPID

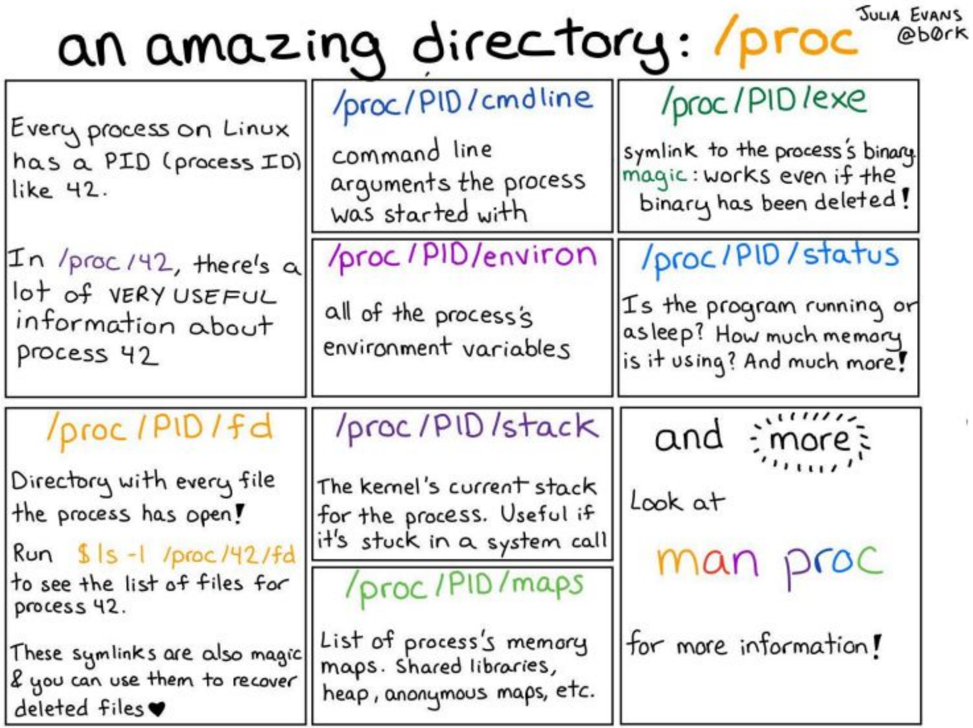
How to kill a process

- **How to Kill the Zombie Process**
  - Second, once zombie process and its parent are found:
    - do you can send a SIGCHLD signal to that parent to tell it to perform this cleanup:

      ```
      $kill -s SIGCHLD <PPID>
      ```
  - In worst case scenarios where the parent is generating a lot of zombie processes that are not getting cleaned up, you can kill or restart the parent. However, this should be used rarely if you are sure this is what you want to do. It can have unintended consequences.

Process is also a “file”

- ❑ Virtual file system /proc/
- ❑ Info about system
- ❑ Info about kernel parameters
  - ❑ /proc/sys/<CLASS>/<PARAMETER>
  - ❑ Files are writable
- ❑ Info about processes /proc/PID/
- ❑ Some files aren’t human-readable
- ❑ Info for all monitoring tools:
  - ❑ $ps for example just aggregate info from /proc and show in the human-readable form



an amazing directory: /proc

Julia Evans @b0rk

Every process on Linux has a PID (process ID) like 42.

In /proc/42, there's a lot of VERY USEFUL information about process 42

/proc/PID/cmdline

command line arguments the process was started with

/proc/PID/exe

symlink to the process's binary. magic: works even if the binary has been deleted!

/proc/PID/environ

all of the process's environment variables

/proc/PID/status

Is the program running or asleep? How much memory is it using? And much more!

/proc/PID/fd

Directory with every file the process has open!

Run $ ls -l /proc/42/fd to see the list of files for process 42.

These symlinks are also magic & you can use them to recover deleted files ♥

/proc/PID/stack

The kernel's current stack for the process. Useful if it's stuck in a system call

/proc/PID/maps

List of process's memory maps. Shared libraries, heap, anonymous maps, etc.

and more

Look at

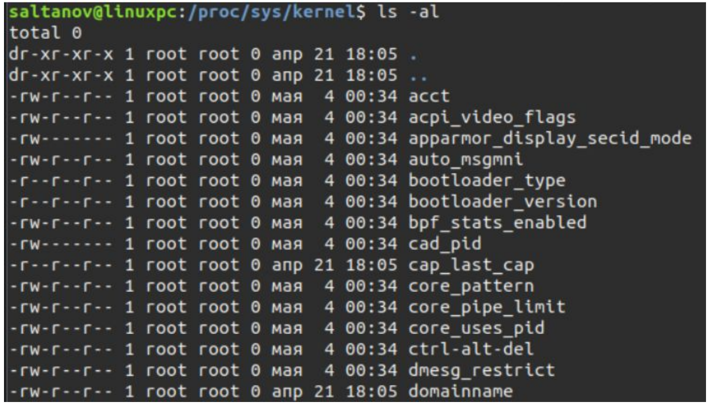
man proc

for more information!

Process is also a “file”

If we don't have tools like top, which may happen in the docker containers we can manage things from /proc . Examples:

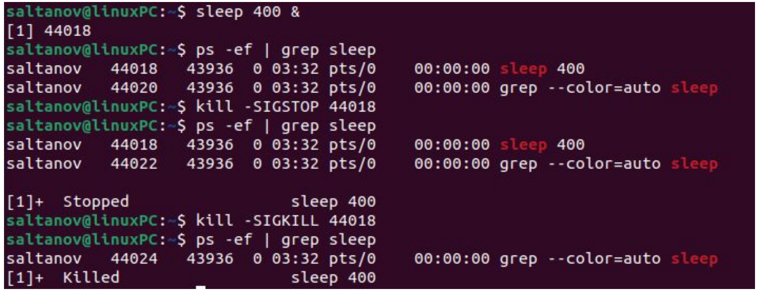
- `$grep MemTotal /proc/meminfo` -Total physical RAM on machine
- `$cat /proc/uptime` - total time since after boot
- `$cat /proc/<pid>/environ` - see what environmental variables was used. Could be useful since some apps keep configurations in there
- `$cat /proc/<pid>/cmdline` bash command from which command was run
- we can change kernel parameters also (RW access rights for the root)



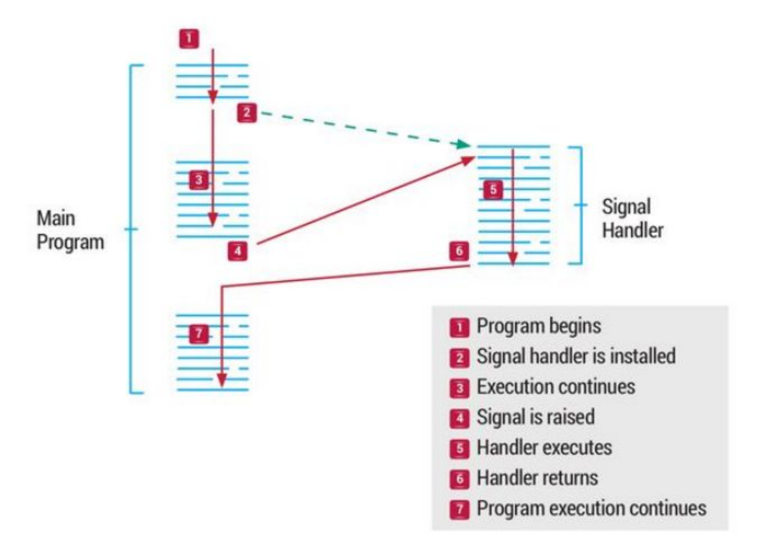
```
saltanov@linuxpc:/proc/sys/kernel$ ls -al
total 0
dr-xr-xr-x 1 root root 0 апр 21 18:05 .
dr-xr-xr-x 1 root root 0 апр 21 18:05 ..
-rw-r--r-- 1 root root 0 мая  4 00:34 acct
-rw-r--r-- 1 root root 0 мая  4 00:34 acpi_video_flags
-rw------- 1 root root 0 мая  4 00:34 apparmor_display_secid_mode
-rw-r--r-- 1 root root 0 мая  4 00:34 auto_msgmni
-r--r--r-- 1 root root 0 мая  4 00:34 bootloader_type
-r--r--r-- 1 root root 0 мая  4 00:34 bootloader_version
-rw-r--r-- 1 root root 0 мая  4 00:34 bpf_stats_enabled
-rw------- 1 root root 0 мая  4 00:34 cad_pid
-r--r--r-- 1 root root 0 апр 21 18:05 cap_last_cap
-rw-r--r-- 1 root root 0 мая  4 00:34 core_pattern
-rw-r--r-- 1 root root 0 мая  4 00:34 core_pipe_limit
-rw-r--r-- 1 root root 0 мая  4 00:34 core_uses_pid
-rw-r--r-- 1 root root 0 мая  4 00:34 ctrl-alt-del
-rw-r--r-- 1 root root 0 мая  4 00:34 dmesg_restrict
-rw-r--r-- 1 root root 0 апр 21 18:05 domainname
```

POSIX signals

- ❑ An IPC mechanism used in Unix to indicate that a particular event has occurred
- ❑ A signal is a short messages sent to a process, or group of processes, containing the number identifying the signal:
  - ❑ No data is delivered with traditional signals
- ❑ In Linux you can view:
  - ❑ `$man signal` OR `$man 7 signal` to search for signals definition
  - ❑ `$kill -l` to see all signals in the numeric forms



```
saltanov@linuxPC:~$ sleep 400 &
[1] 44018
saltanov@linuxPC:~$ ps -ef | grep sleep
saltanov   44018   43936  0 03:32 pts/0    00:00:00 sleep 400
saltanov   44020   43936  0 03:32 pts/0    00:00:00 grep --color=auto sleep
saltanov@linuxPC:~$ kill -SIGSTOP 44018
saltanov@linuxPC:~$ ps -ef | grep sleep
saltanov   44018   43936  0 03:32 pts/0    00:00:00 sleep 400
saltanov   44022   43936  0 03:32 pts/0    00:00:00 grep --color=auto sleep

[1]+  Stopped                 sleep 400
saltanov@linuxPC:~$ kill -SIGKILL 44018
saltanov@linuxPC:~$ ps -ef | grep sleep
saltanov   44024   43936  0 03:32 pts/0    00:00:00 grep --color=auto sleep
[1]+  Killed                  sleep 400
```



POSIX signals

- Two main signals **SIGKILL** and **SIGSTOP** you cant use in the application. It must act only on the kernel level (handled only by OS kernel)
  - As a developer you can catch any other signals and do smth else not just default actions as its supposed to be.

- **Most common signals:**
  - **1 - SIGHUP** - controlling terminal closed, If a command is executed inside a terminal window and the terminal window is closed while the command process is still running, it receives SIGHUP.
  - **2 - SIGINT** - interrupt process stream, terminate the process by issuing ctrl-C
  - **9 - SIGKILL** - terminate immediately/hard kill - this signal can not be ignored, use when 15 doesn't work or when something disastrous might happen if process is allowed to cont.. If program needs to do smth before it exits -> then this part of the execution will be lost.
  - **15 - SIGTERM** - terminated while issues a soft kill, let process to finish execution
  - **19 - SIGSTOP** - Pause the process / free command line, ctrl-Z (1st)
  - **18 - SIGCONT** - Resume process, ctrl-Z (2nd)

- Use **trap** command in the bash scripts to program different signals in desired way