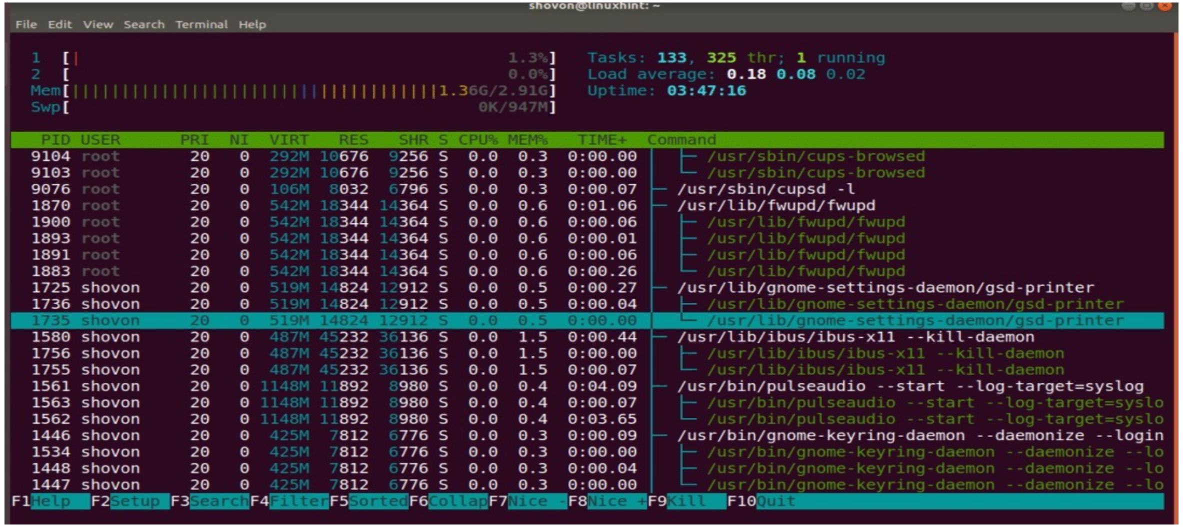
Process Management

Operating Systems
Spring 1401

View all processes running on Linux live

- In Linux, you can see the system status and information of all processes using top or **htop** command.

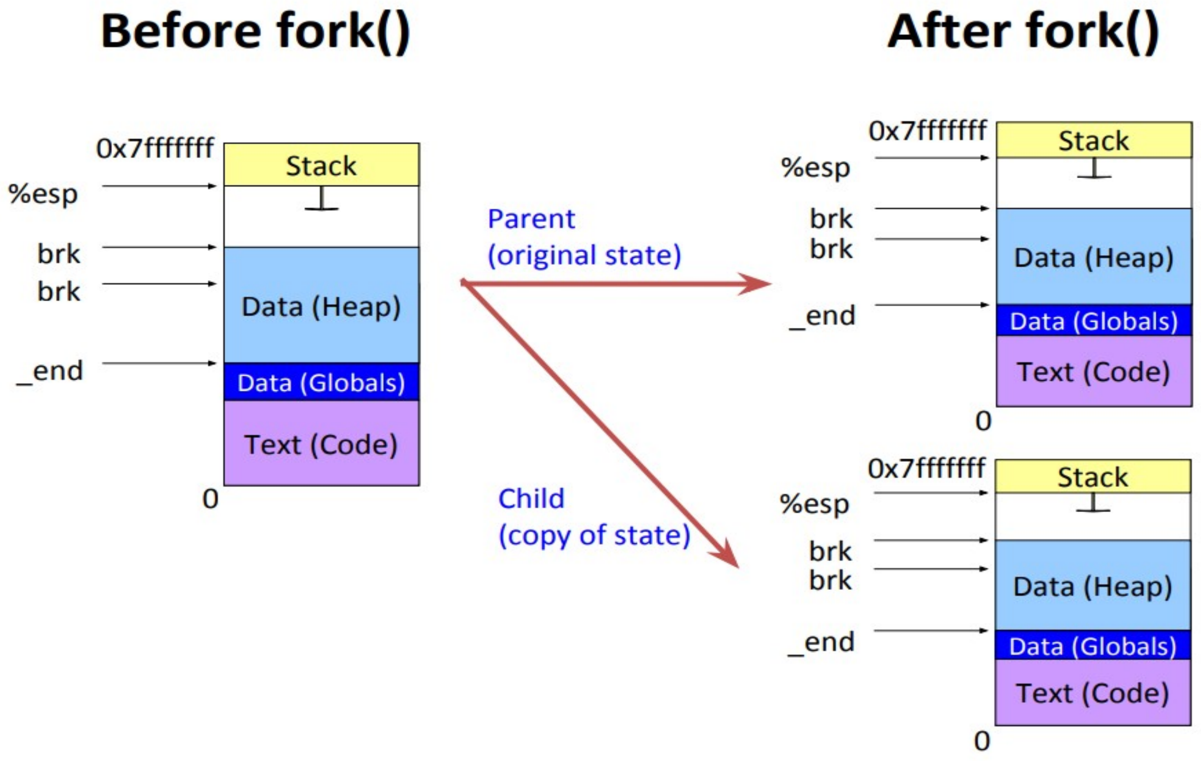


```
  1  [|                                      1.3%]   Tasks: 133, 325 thr; 1 running
  2  [                                       0.0%]   Load average: 0.18 0.08 0.02
  Mem[||||||||||||||||||||||||||||||||||1.36G/2.91G]   Uptime: 03:47:16
  Swp[                                    0K/947M]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 9104 root       20   0  292M 10676  9256 S  0.0  0.3  0:00.00 │  ├─ /usr/sbin/cups-browsed
 9103 root       20   0  292M 10676  9256 S  0.0  0.3  0:00.00 │  └─ /usr/sbin/cups-browsed
 9076 root       20   0  106M  8032  6796 S  0.0  0.3  0:00.07 ├─ /usr/sbin/cupsd -l
 1870 root       20   0  542M 18344 14364 S  0.0  0.6  0:01.06 ├─ /usr/lib/fwupd/fwupd
 1900 root       20   0  542M 18344 14364 S  0.0  0.6  0:00.06 │  ├─ /usr/lib/fwupd/fwupd
 1893 root       20   0  542M 18344 14364 S  0.0  0.6  0:00.01 │  ├─ /usr/lib/fwupd/fwupd
 1891 root       20   0  542M 18344 14364 S  0.0  0.6  0:00.06 │  ├─ /usr/lib/fwupd/fwupd
 1883 root       20   0  542M 18344 14364 S  0.0  0.6  0:00.26 │  └─ /usr/lib/fwupd/fwupd
 1725 shovon     20   0  519M 14824 12912 S  0.0  0.5  0:00.27 ├─ /usr/lib/gnome-settings-daemon/gsd-printer
 1736 shovon     20   0  519M 14824 12912 S  0.0  0.5  0:00.04 │  ├─ /usr/lib/gnome-settings-daemon/gsd-printer
 1735 shovon     20   0  519M 14824 12912 S  0.0  0.5  0:00.00 │  └─ /usr/lib/gnome-settings-daemon/gsd-printer
 1580 shovon     20   0  487M 45232 36136 S  0.0  1.5  0:00.44 ├─ /usr/lib/ibus/ibus-x11 --kill-daemon
 1756 shovon     20   0  487M 45232 36136 S  0.0  1.5  0:00.00 │  ├─ /usr/lib/ibus/ibus-x11 --kill-daemon
 1755 shovon     20   0  487M 45232 36136 S  0.0  1.5  0:00.07 │  └─ /usr/lib/ibus/ibus-x11 --kill-daemon
 1561 shovon     20   0 1148M 11892  8980 S  0.0  0.4  0:04.09 ├─ /usr/bin/pulseaudio --start --log-target=syslog
 1563 shovon     20   0 1148M 11892  8980 S  0.0  0.4  0:00.07 │  ├─ /usr/bin/pulseaudio --start --log-target=syslo
 1562 shovon     20   0 1148M 11892  8980 S  0.0  0.4  0:03.65 │  └─ /usr/bin/pulseaudio --start --log-target=syslo
 1446 shovon     20   0  425M  7812  6776 S  0.0  0.3  0:00.09 ├─ /usr/bin/gnome-keyring-daemon --daemonize --login
 1534 shovon     20   0  425M  7812  6776 S  0.0  0.3  0:00.00 │  ├─ /usr/bin/gnome-keyring-daemon --daemonize --lo
 1448 shovon     20   0  425M  7812  6776 S  0.0  0.3  0:00.04 │  ├─ /usr/bin/gnome-keyring-daemon --daemonize --lo
 1447 shovon     20   0  425M  7812  6776 S  0.0  0.3  0:00.00 │  └─ /usr/bin/gnome-keyring-daemon --daemonize --lo
F1Help  F2Setup F3SearchF4FilterF5SortedF6CollapF7Nice -F8Nice +F9Kill  F10Quit
```

- Using this command, you can find every thing about each process and do everything possible with these processes.
- On Ubuntu, you can easily install it using apt command.

Fork


Before fork()
After fork()
0x7fffffff
%esp
brk
brk
_end
0
Stack
Data (Heap)
Data (Globals)
Text (Code)
Parent
(original state)
Child
(copy of state)

Fork

- **fork**() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

- The child process and the parent process run in separate memory spaces. At the time of **fork**() both memory spaces have the same content. Memory writes, file mappings (mmap()), and unmappings(munmap()) performed by one of the processes do not affect the other.

- The child has its own unique process ID, and this PID does not match the ID of any existing process group (setpgid()) or session.

Fork

```
fork ();   // Line 1
fork ();   // Line 2
fork ();   // Line 3
```

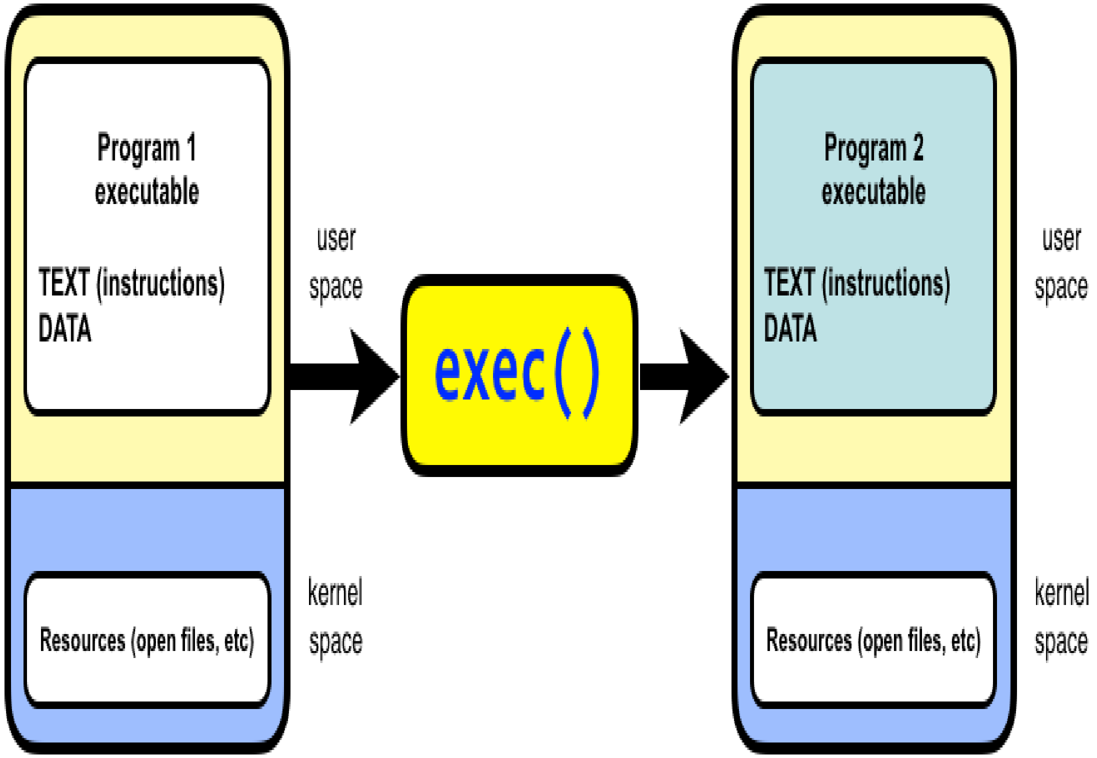
```
      L1       // There will be 1 child
process
    /     \     // created by line 1.
  L2      L2    // There will be 2 child
processes
 /  \    /  \   //  created by line 2
L3  L3  L3  L3  // There will be 4 child
processes
```

exec

exec() family of functions or sys calls replaces current process image with new process image.

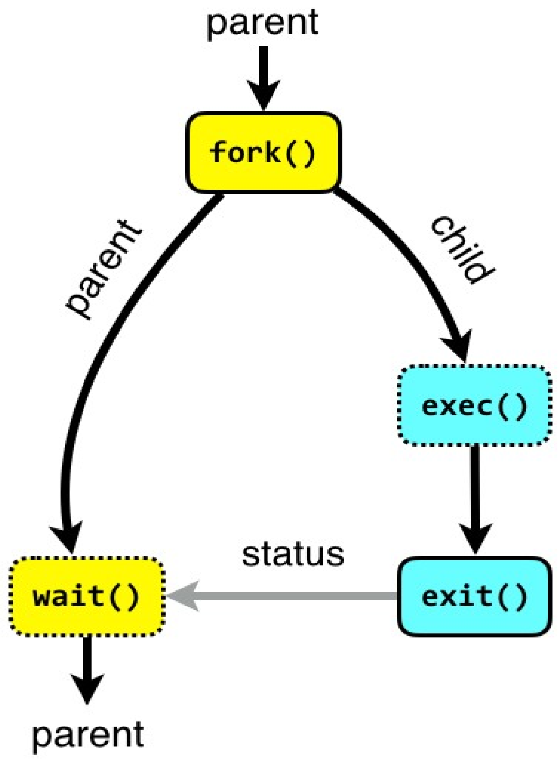
The parent program will be finished after calling exec() even if it still contains other lines of code after the function call.

What to do if we want both?!!



exec + fork

Since the **exec** family of functions **replaces** the current process with a new process image, you need to fork before calling exec, so that the newly forked copy of your process gets replaced (instead of the original getting replaced).

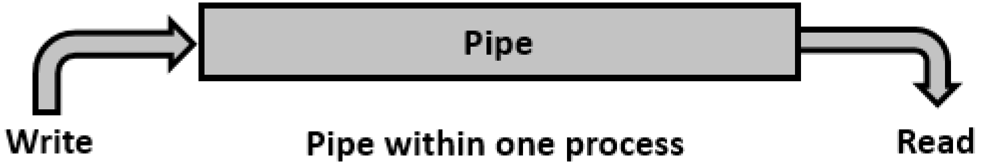


wait

This system call is 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 the terminated child remains in a "zombie" state.

As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes. If a parent process terminates, then its "zombie" children (if any) are adopted by **init**(), which automatically performs a wait to remove the zombies.

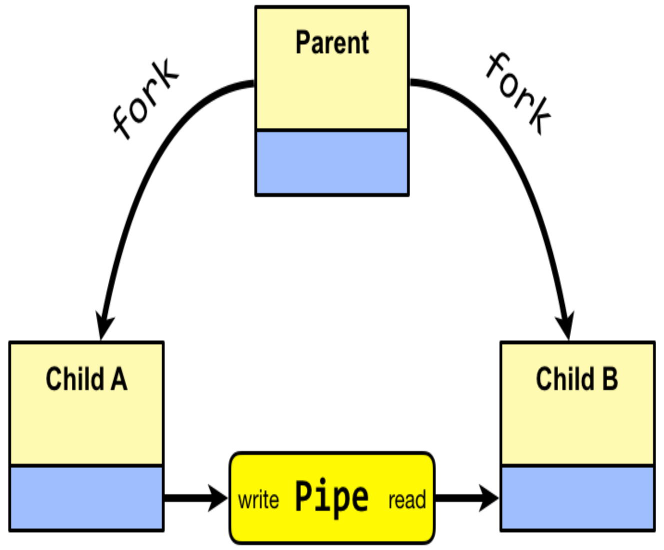
Unnamed pipe



Unnamed pipe is a communication medium between two or more related or interrelated processes. It can be either within one process or a communication between the child and the parent processes. Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc. Communication is achieved by one process writing into the pipe and other reading from the pipe. To achieve the pipe system call, create two files, one to write into the file and another to read from the file.

Unnamed pipe

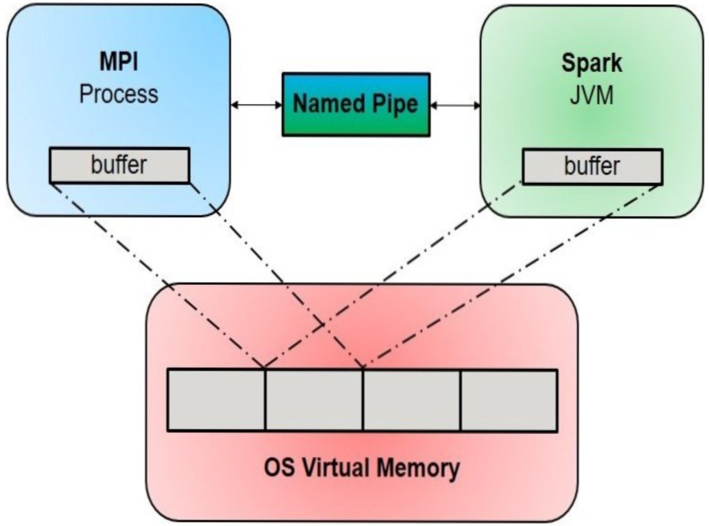
- Unnamed pipe is one-way communication only i.e we can use a pipe such that One process write to the pipe, and the other process reads from the pipe. It opens a pipe, which is an area of main memory that is treated as a ***"virtual file"***.
- The unnamed pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this "virtual file" or pipe and another related process can read from it.
- If a process tries to read before something is written to the pipe, the process is blocked until something is written.



Named pipe

A named pipe is a one-way or duplex pipe that provides communication between the pipe server and some pipe clients. A pipe is a section of memory that is used for interprocess communication. A named pipe can be described as first in, first out (FIFO); the inputs that enter first will be output first.

A named pipe differs from an unnamed pipe in that it can exist beyond the life of its associated processes and must be explicitly deleted. Every instance of a named pipe shares the same name but each instance has its own buffers and handles(file descriptors).


MPI
Process
buffer
Named Pipe
Spark
JVM
buffer
OS Virtual Memory