1. What happens step by step when you type a command in bash (e.g., ls) until you see the output?

1) From keyboard to the shell

1. You press keys in a terminal app (e.g., GNOME Terminal).
2. The terminal talks to the kernel through a pseudo-terminal (pty): it writes your keystrokes to the pty master; the kernel delivers them to the pty slave that bash is attached to (/dev/pts/N).
3. Bash (via readline) reads bytes from its tty, does line editing, history, etc., until it sees Enter.

2) Bash parses what you typed

4. Bash tokenizes and expands the command line in this (simplified) order:
   brace → tilde → parameter/command/arithmetic → word splitting → globbing.
   Aliases/functions are resolved; variables and wildcards expand.
5. Bash decides what the command is: builtin, function, or external.
   - If builtin (e.g., cd), bash may run it in the shell process.
   - If external (/bin/ls), continue below.

3) Finding the executable

6. Bash searches $PATH (caching results in a hash table) and picks the first executable named ls.

7. The kernel will later enforce that the file is executable (x bit, mount options like noexec) and that you have execute permissions and search (x) on each directory in the path.

4) Setting up I/O, pipes, jobs

8. Bash prepares redirections (>, <, 2>, etc.) and pipelines if any, creating pipes and dup'ing file descriptors as needed.
9. For job control, bash creates/joins a process group and gives the job the controlling terminal.

5) Creating the process

10. Bash calls fork() (or vfork()/posix_spawn()), creating a child.
11. In the child, bash restores default signal handlers, applies redirections, sets environment (ENV, LANG, PATH, PWD, etc.), umask, and other execution context.

6) Entering the kernel to run it

12. The child calls execve(path, argv, envp). Now it's the kernel's turn:
    - If the file is ELF: the kernel maps program segments and, for dynamic binaries, jumps to the dynamic loader (ld-linux), which maps needed shared libraries and performs relocations, then enters the program's _start → main.
    - If it's a script with shebang (#! /usr/bin/python): the kernel runs the interpreter and passes the script path as an argument.
    - If permissions fail, you get EACCES/ENOENT, etc.

7) The program runs and does syscalls

13. ls runs in user space; when it needs the kernel, it uses system calls:

- Determine if stdout is a tty (isatty/ioctl(TIOCGWINSZ)) to decide columns/colors.
- Open the directory (openat()), read entries (getdents64()), stat() files, sort/format output.
- Write output with write() to fd 1 (stdout).

8) From program output to your screen

14. Those write() calls go to the pty slave → kernel relays to the pty master → terminal emulator receives bytes.
15. The terminal renders text: interprets control sequences (ANSI colors, bold, cursor moves) and draws glyphs on screen.

9) Process exit and shell prompt

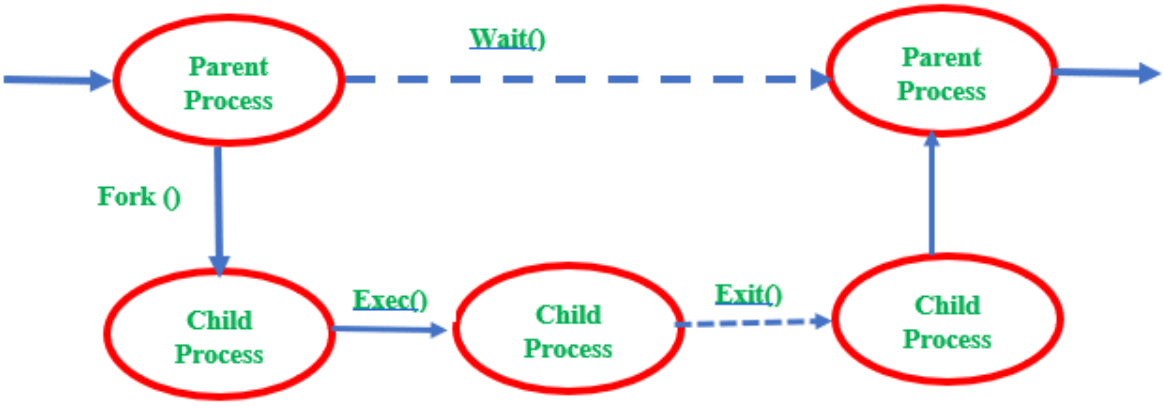
16. ls calls exit(); the kernel reclaims resources and sends SIGCHLD to bash.
17. Bash **waitpid()**s for the child, retrieves the exit status ($?), updates job tables, and prints the prompt again.

2. Explain the types of processes in Linux: daemon, zombie, orphan. How can you detect them?

Zombie Process

A Zombie process is a process that has been terminated but still has entry in a process table. The zombie process usually occur in the child processes. The process is a zombie for a very short time.

After the process has completed all of its tasks, it reports to the parent process that it is about to terminate.



A zombie process is unable to terminate itself because it is treated as a dead process. Therefore, the parent process needs to execute a command to terminate the child.

Characteristics of Zombie Processes

- **Short-lived:** Typically, zombie processes are short-lived, existing only briefly until the parent process performs a wait() system call to retrieve the exit status.
- **Non-functional:** Zombie processes do not use CPU resources but still occupy space in the process table.

Orphan Process

A child process that remains running even after its parent process is terminated or completed without waiting for the child process execution is called an orphan. A process becomes an orphan unintentionally. Sometimes, orphan processes become intentional due to the long running time required to complete the assigned task without user attention. An orphan process has controlling terminals.

Characteristics of Orphan Processes

- Unintentionally Created: Orphan processes are often created unintentionally when a parent process crashes or is terminated unexpectedly.
- Continues Running: Orphan processes continue to execute until they complete their tasks or are terminated.

Daemon Process

Daemon processes are started working when the system will be bootstrapped and terminate only when the system is shutdown. It does not have a controlling terminal. It always runs in the background.

Characteristics of Daemon Processes

- **Background Operation:** Daemons run in the background, performing system-level tasks without direct user intervention.
- **Long-lived:** Daemons are designed to run for extended periods, handling tasks such as system maintenance, network services, or hardware management.

Detecting daemons:

ps -ef | grep sshd

systemctl status cron

Detecting zombies:

ps aux | grep Z

Detecting orphans:

ps -o pid,ppid,cmd

3. Why do we need Inter-Process Communication (IPC)? List some IPC mechanisms and real-life examples.

Processes need to communicate with each other in many situations. Inter-Process Communication or IPC is a mechanism that allows processes to communicate.

- It helps processes synchronize their activities, share information and avoid conflicts while accessing shared resources.
- There are two method of IPC, shared memory and message passing. An operating system can implement both methods of communication.

1. **Pipes (| and FIFOs)**
    - One-way communication channel between processes.

    Example:

    ls | grep txt
    - Named pipes (FIFOs) can be created with mkfifo.
2. **Message Queues**
    - Allow processes to send/receive structured messages (like mailboxes).
    - Example: A logging system where multiple apps send logs to a central queue.
3. **Shared Memory**
    - Fastest IPC (direct memory access).
    - Multiple processes map the same memory region.
    - Needs synchronization (e.g., semaphores) to avoid race conditions.
    - Example: Databases (e.g., PostgreSQL) use shared memory for cache.

4. **Semaphores**
    - Synchronization mechanism to control access to resources.
    - Example: Multiple processes writing to a file must use a semaphore to avoid corruption.
5. **Signals**
    - Simple notifications to processes (like interrupts).
    - Example: kill -9 1234 sends a SIGKILL to terminate process 1234.
6. **Sockets**

- Enable communication between processes, either **local (Unix domain sockets)** or over a **network (TCP/UDP)**.
- Example:
    - Web server (nginx) and PHP application talk via Unix sockets.
    - Browser communicates with websites using TCP sockets.

7. **Files & Memory-Mapped Files**
    - Processes can share info by reading/writing the same file.
    - Example: /var/log/syslog is read by log monitoring tools.
    - mmap() allows processes to map files into memory for efficient IPC.

Real-Life Examples

- **Shell pipeline** (cat file | grep word | sort) → Pipes
- **System logs** (apps → syslog daemon) → Message Queues / Sockets

- **Databases** (shared cache between worker processes) → Shared Memory + Semaphores
- **Web browser & plugins** → Sockets or Shared Memory
- **Kill command** → Signals