Linux Process Lifecycle: `fork` – `exec` – `wait` Mechanics

🧬 1. `fork()` — Cloning a Process

Purpose:

`fork()` creates a **new child process** by duplicating the **parent's memory space**, file descriptors, and execution context.

Key Characteristics:

- Returns **0** in the child process
- Returns **child PID** in the parent process
- Returns **-1** on error

```
pid_t pid = fork();
if (pid == 0) {
    // Child process
} else if (pid > 0) {
    // Parent process
} else {
    // Error
}
```

Behind the Scenes:

- **Copy-on-write (COW)** is used — memory is not physically copied unless modified.
- The child gets a new PID, but shares open file descriptors initially.

⏳ 3. `wait()` / `waitpid()` — Synchronizing with Child Termination

Purpose:

Allows a parent process to **pause** until its child process **exits**, so it can collect the **exit status** and prevent **zombie processes**.

Example:

```
pid_t pid = wait(&status);  // blocks until any child exits
```

```
pid_t pid = waitpid(child_pid, &status, 0);  // wait for a specific ch
```

- Macros like `WIFEXITED(status)` and `WEXITSTATUS(status)` extract exit codes.

🪦 Zombie and Orphan Processes

- **Zombie**: A child that has exited but has not been waited on; still occupies an entry in the process table.
- **Orphan**: A child whose parent exited before it; reparented to `init` (`PID 1`) which will reap it.

🔍 Tools to Observe This Lifecycle

- `strace ./a.out` — trace system calls like `fork`, `execve`, `wait`
- `ps -ef` — view process hierarchy
- `top` or `htop` — live process info

✅ Conclusion

Understanding `fork()`, `exec()`, and `wait()` is crucial for:

- Writing custom shells or process supervisors
- Building daemons and services

- Handling concurrency and job control

This trio defines how processes are spawned, programs are executed, and resources are released — forming the backbone of Unix/Linux multitasking.