Real-World Context

When you open a terminal and run a command like ls, shell forks a child process to create a copy of itself. The child uses exec to replace its memory space with the requested command. The parent waits for the child to complete before taking the next command. This model is at the core of real-world multitasking in Unix-like operating systems. It ensures that multiple processes can run independently while the OS manages scheduling, memory, and I/O resources.

Structured Use Case Challenge

Discover: Students will run simple shell commands (ps, top) to observe parent-child process relationships. Watch how process IDs change when commands run.
Design : Students will write small C programs where: A parent forks multiple children. Each child executes a different command (ls, date, cat file.txt). The parent uses wait() to ensure orderly cleanup and no zombies.
Validate : They will Run their programs under varied conditions:

- Compare behavior when wait() is included vs omitted.
- Confirm that exec() replaces the child's memory space (child cannot return to parent code after successful exec).

Understanding fork() in Process Creation

fork() is a system call used to create a new process. The new process is called the child process. The process that called fork() is the parent process. Both processes (parent & child) will continue executing the same program from the line after the fork() call.

When `fork()` is called, it returns twice: once in the parent process with the child's PID, and once in the child process with a return value of 0

- In the parent process → fork() returns the child's PID (a positive integer).
- In the child process → fork() returns 0.
- If fork() fails → returns -1 (no child created).

Key details about forked processes

1. Separate memory space
   - The child gets a copy of the parent's memory at the time of the fork.
   - Changes made in the child do not affect the parent (and vice versa), except for shared resources like open files.

2. File descriptors are inherited
   - If the parent has open files (like stdout, sockets, or pipes), the child inherits them.
   - This is why both parent and child can print to the same terminal.

3. Execution order is not guaranteed
   - After fork, both parent and child run concurrently.
   - Which one runs first depends on the OS scheduler.

4. Zombie processes
   - When a child finishes before the parent calls wait() or waitpid(), it becomes a zombie (process entry still in the process table until the parent reaps it).

Example 1

Write a C program to demonstrate process creation using the fork() system call . The child process prints its own Process ID (PID) and its parent's Process ID (PPID). The parent process prints its own Process ID (PID) and the child's Process ID.

```
#include <stdio.h>
#include <unistd.h>

int main() {
   pid_t pid = fork();

   if (pid < 0) {
      perror("fork failed");
      return 1;
   } else if (pid == 0) {
      printf("Child process: PID=%d, Parent PID=%d\n", getpid(), getppid());
   } else {
      printf("Parent process: PID=%d, Child PID=%d\n", getpid(), pid);
```

```
    }

    return 0;
}
```

Expected output:

Parent process: PID=1000, Child PID=1001
Child process: PID=1001, Parent PID=1000

exec system call

The exec system call family in Unix- operating systems is used to replace the current process image with a new process image. This means that the currently running program is terminated, and a new program is loaded and executed within the same process. The process ID (PID) remains unchanged, but the code, data, heap, and stack segments of the process are replaced by those of the new program.

- When you call exec, the current process is **overwritten** by the new program — it doesn't create a new process (unlike fork).
- If exec succeeds, it never returns; the new program takes over.
- If it fails (e.g., program not found), it returns -1 and sets errno.

common variants of the exec family:

1. execl(const char *path, const char *arg, ...):Takes a full path to the executable.
2. execv(const char *path, char *const argv[]):Takes a full path to the executable.
3. execle(const char *path, const char *arg, ..., char *const envp[]):Takes a full path to the executable.
4. execve(const char *path, char *const argv[], char *const envp[]):Takes a full path to the executable.
5. execlp(const char *file,
6. const char *arg, ...):Searches for the executable in the directories specified by the PATH environment variable.
7. execvp(const char *file, char *const argv[]):Searches for the executable in the directories specified by the PATH environment variable.

Write a c program to demonstrate the execl system call

Example 2

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    else if (pid == 0) {
        // Child process replaces itself with "ls" command
        execl("/bin/ls", "ls", "-l", NULL);
        // If execl fails
        perror("execl failed");
        exit(1);
    }
    else {
        // Parent waits for child
        wait(NULL);
        printf("Child process finished.\n");
    }

    return 0;
}
```

In this code, in the system call execl("/bin/ls", "ls", "-l", NULL);
/bin/ls → full path to the program.
"ls" → name of the program (first arg, usually same as the executable)."-l" → extra argument passed to ls.NULL → indicates the end of arguments.

Example 3

Write a c program to demonstrate If exec succeeds, it never returns; the new program takes over.

```
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Before exec: PID = %d\n", getpid());

    // Replace current program with "ls"
    execlp("ls", "ls", NULL);

    printf("After  exec: PID = %d\n", getpid());

   // This line will only run if exec fails
    perror("exec failed");

    return 0;
}
```