Zombie and orphan processes are two terms used to describe certain states of processes in an operating system. Let's understand what each term means:

Zombie Process:

A zombie process is a terminated process that has completed its execution but still has an entry in the process table. When a process finishes its execution, it sends a termination signal to its parent process. The parent process, in turn, is responsible for cleaning up the resources of the terminated child process by calling the wait() or waitpid() system call to retrieve the exit status. Until the parent process performs this cleanup, the terminated process remains in the process table as a zombie.

Zombie processes typically do not consume any system resources other than an entry in the process table. However, if too many zombie processes accumulate, it can exhaust the available entries in the process table, leading to issues.

Orphan Process:

An orphan process is a child process that continues its execution even after its parent process terminates or is completed. When a parent process terminates before its child processes, the operating system assigns the ownership of these child processes to the init process (process ID 1), which acts as the parent for all orphan processes.

Orphan processes are typically not problematic as the init process takes responsibility for cleaning up the resources of orphaned child processes. The init process periodically calls the wait() or waitpid() system call to collect the exit statuses of orphaned child processes, preventing them from turning into zombie processes.

It's important for the parent process to handle the termination signals sent by its child processes and perform the necessary cleanup to avoid the accumulation of zombie processes. Similarly, the init process or any other suitable process should handle orphaned child processes to prevent resource leakage.

Both zombie and orphan processes are normal phenomena in an operating system, but it's essential to handle them appropriately to maintain system efficiency and prevent any adverse effects.

Example 1: Creating a Zombie Process

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main() {

pid\_t childPid = fork();

if (childPid == 0) {

// Child process

exit(0);

} else {

// Parent process

sleep(5); // Parent sleeps for a while without cleaning up the child process

// The child process becomes a zombie in the process table

}

return 0;

}

In this example, the parent process creates a child process using fork(). The child process immediately exits using exit(0). However, the parent process does not collect the exit status of the child process using wait() or waitpid(), causing the child process to become a zombie in the process table.

Example 2: Creating an Orphan Process

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main() {

pid\_t childPid = fork();

if (childPid == 0) {

// Child process

sleep(5); // Child process continues execution even after the parent terminates

exit(0);

} else {

// Parent process

exit(0); // Parent process terminates before the child process

}

return 0;

}

In this example, the parent process creates a child process using fork(). After forking, the parent process immediately terminates using exit(0). However, the child process continues execution even after the parent process has terminated. The child process becomes an orphan and is adopted by the init process (PID 1).

These examples illustrate the scenarios of creating zombie and orphan processes. To avoid zombies, it is important for the parent process to collect the exit status of its child processes using wait() or waitpid(). Similarly, the init process or other appropriate processes should handle orphaned child processes to prevent resource leakage.