# S.B. JAIN INSTITUTE OF TECHNOLOGY MANAGEMENT & RESEARCH, NAGPUR

## Practical 04

**Aim:** Implement process creation using the fork() system call in Linux to generate parent and child processes, retrieve their PID and PPID, and analyse system behavior by creating orphan and zombie processes.

**Name: Abhinav Doifode**

**USN: CM24002**

**Semester / Year:**

**Academic Session:**

**Date of Performance:**

**Date of Submission:**

❖ **Aim:** Implement process creation using the fork() system call in Linux to generate parent and child processes, retrieve their PID and PPID, and analyse system behaviour by creating orphan and zombie processes.

❖ **Tasks to be done in this Practical.**

1. Adam is working in an IT company. He has been given a task to reduce the load of a system by killing some of the processes running in the LINUX operating system. Which commands will he use to complete the given task with the help of the following operation?
   - Kill processes by name
   - Kill a process based on the process name
   - Kill a single process at a time with the given process ID

2. Write a program for process creation using C
   - Orphan Process
   - Zombie Process

3. Create the process using fork () system call.
   - Child Process creation
   - Parent process creation
   - PPID and PID

❖ **Objectives:**

a. Implement process creation using the fork() system call to generate parent and child processes.
b. Retrieve and display the Process ID (PID) and Parent Process ID (PPID) for both processes.
c. Analyse system behaviour by creating and observing orphan and zombie processes.

❖ **Requirements:**

✓ **Hardware Requirements:**

- Processor: Minimum 1 GHz.
- RAM: 512 MB or higher.
- Storage: 100 MB free space.

✓ **Software Requirements:**

- Operating System: Linux/Unix-based.
- Compiler: GCC Compiler.
- Text Editor: Nano, Vim, or any preferred editor.

## ❖ Theory:

In many operating systems, the fork system call is an essential operation. The fork system call allows the creation of a new process. When a process calls the fork(), it duplicates itself, resulting in two processes running at the same time. The new process that is created is called a child process. It is a copy of the parent process. The fork system call is required for process creation and enables many important features such as parallel processing, multitasking, and the creation of complex process hierarchies.
It develops an entirely new process with a distinct execution setting. The new process has its own address space, and memory, and is a perfect duplicate of the caller process.

### Basic Terminologies Used in Fork System Call in Operating System

- ❖ **Process:** In an operating system, a process is an instance of a program that is currently running. It is a separate entity with its own memory, resources, CPU, I/O hardware, and files.
- ❖ **Parent Process:** The process that uses the fork system call to start a new child process is referred to as the parent process. It acts as the parent process's beginning point and can go on running after the fork.
- ❖ **Child Process:** The newly generated process as a consequence of the fork system call is referred to as the child process. It has its own distinct process ID (PID), and memory, and is a duplicate of the parent process.
- ❖ **Process ID:** A process ID (PID) is a special identification that the operating system assigns to each process.
- ❖ **Copy-on-Write:** The fork system call makes use of the memory management strategy known as copy-on-write. Until one of them makes changes to the shared memory, it enables the parent and child processes to share the same physical memory. To preserve data integrity, a second copy is then made.
- ❖ **Return Value:** The fork system call's return value gives both the parent and child process information. It assists in handling mistakes during process formation and determining the execution route.

### Process Creation

When the fork system call is used, the operating system completely copies the parent process to produce a new child process. The memory, open file descriptors, and other pertinent properties of the parent process are passed down to the child process. The child process, however, has a unique execution route and PID.
The copy-on-write method is used by the fork system call to maximize memory use. At first, the physical memory pages used by the parent and child processes are the same. To avoid unintentional changes, a separate copy is made whenever either process alters a shared memory page.

The return value of the fork call can be used by the parent to determine the execution path of the child process. If it returns 0 then it is executing the child process, if it returns -1 then there is some error; and if it returns some positive value, then it is the PID of the child process.

*Fork() System call*

**Advantages of Fork System Call**

- **Creating new processes** with the fork system call facilitates the running of several tasks concurrently within an operating system. The system's efficiency and multitasking skills are improved by this concurrency.
- **Code reuse:** The child process inherits an exact duplicate of the parent process, including every code segment, when the fork system call is used. By using existing code, this feature encourages code reuse and streamlines the creation of complicated programmes.
- **Memory Optimisation:** When using the fork system call, the copy-on-write method optimises the use of memory. Initial memory overhead is minimised since parent and child processes share the same physical memory pages. Only when a process changes a shared memory page, improving memory efficiency, does copying take place.
- Process isolation is achieved by giving each process started by the fork system call its own memory area and set of resources. System stability and security are improved because of this isolation, which prevents processes from interfering with one another.

Disadvantages of Fork System Call

- **Memory Overhead:** The fork system call has memory overhead even with the copy-on-write optimisation. The parent process is first copied in its entirety, including all of its memory, which increases memory use.
- **Duplication of Resources:** When a process forks, the child process duplicates all open file descriptors, network connections, and other resources. This duplication may waste resources and perhaps result in inefficiencies.
- **Communication complexity:** The fork system call generates independent processes that can need to coordinate and communicate with one another. To enable data transmission across processes, interprocess communication methods, such as pipes or shared memory, must be built, which might add complexity.
- **Impact on System Performance:** Forking a process duplicates memory allocation, resource management, and other system tasks. Performance of the system may be affected by this, particularly in situations where processes are often started and stopped.

## CODES

### 1. FORK.c :

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

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

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

    return 0;
      }
```

### 2. Orphan.c :

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>

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

    if (pid < 0) {
        printf("Fork failed!\n");
        return 1;
    }
    else if (pid == 0) {
        // Child process
        sleep(5);  // Sleep to let parent terminate first
        printf("Child Process (Orphan):\n");
        printf("  PID = %d\n", getpid());
        printf("  Parent PID (now 1 or init) = %d\n", getppid());
    }
    else {
        // Parent process
        printf("Parent Process:\n");
        printf("  PID = %d\n", getpid());
        printf("  Exiting parent...\n");
        exit(0);  // Parent exits immediately
    }

    return 0;
}
```

## 3. Zombie.c :

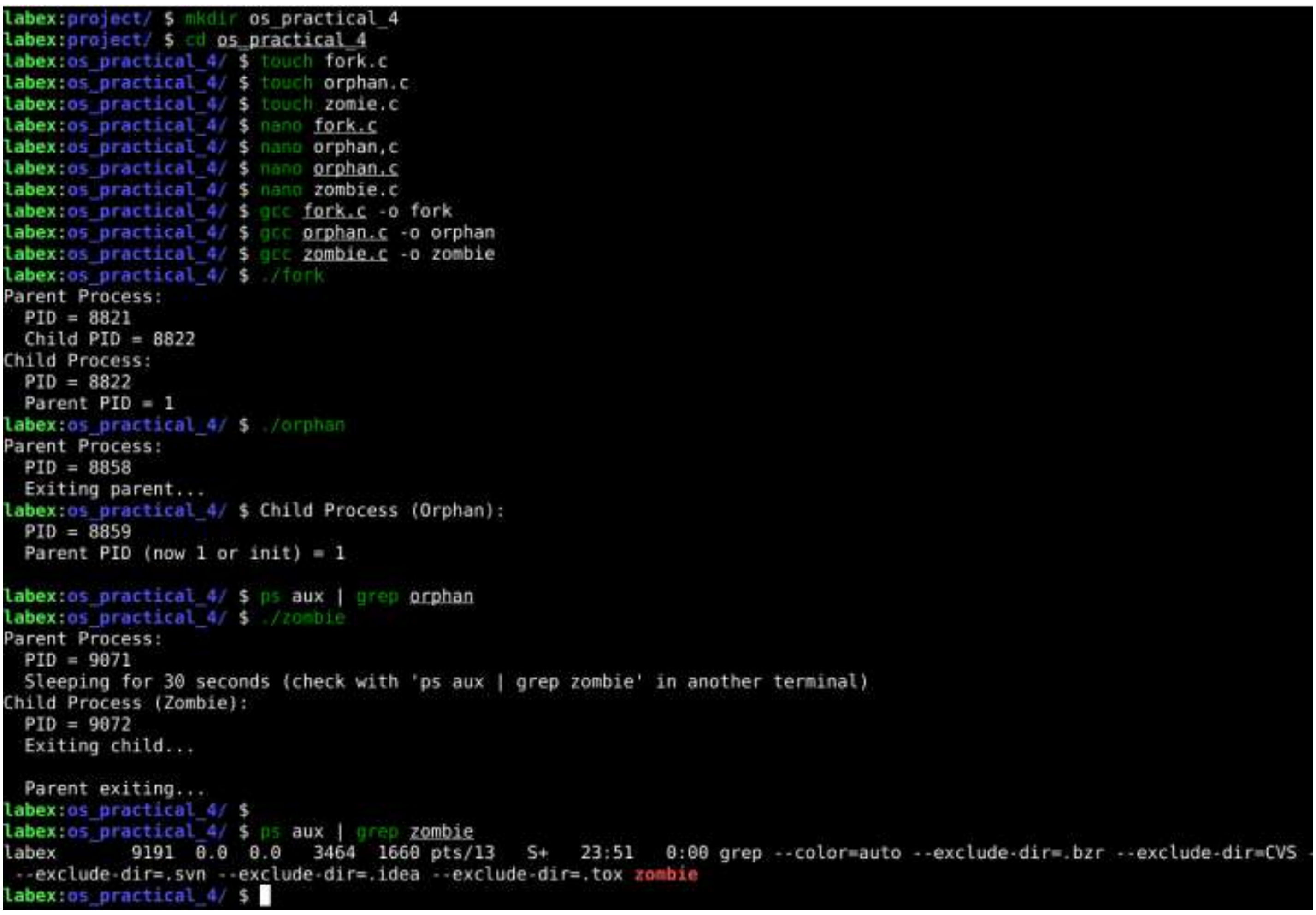
```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>

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

    if (pid < 0) {
        printf("Fork failed!\n");
        return 1;
    }
    else if (pid == 0) {
        // Child process
        printf("Child Process (Zombie):\n");
        printf("  PID = %d\n", getpid());
        printf("  Exiting child...\n");
        exit(0);  // Child exits but parent doesn't collect status
    }
    else {
        // Parent process
        printf("Parent Process:\n");
        printf("  PID = %d\n", getpid());
        printf("  Sleeping for 30 seconds (check with 'ps aux | grep zombie' in another
terminal)\n");
        sleep(30);  // Parent sleeps, doesn't collect child's exit status
        printf("  Parent exiting...\n");
    }

    return 0;
}
```

## ❖ Output :



```
labex:project/ $ mkdir os_practical_4
labex:project/ $ cd os_practical_4
labex:os_practical_4/ $ touch fork.c
labex:os_practical_4/ $ touch orphan.c
labex:os_practical_4/ $ touch zomie.c
labex:os_practical_4/ $ nano fork.c
labex:os_practical_4/ $ nano orphan,c
labex:os_practical_4/ $ nano orphan.c
labex:os_practical_4/ $ nano zombie.c
labex:os_practical_4/ $ gcc fork.c -o fork
labex:os_practical_4/ $ gcc orphan.c -o orphan
labex:os_practical_4/ $ gcc zombie.c -o zombie
labex:os_practical_4/ $ ./fork
Parent Process:
  PID = 8821
  Child PID = 8822
Child Process:
  PID = 8822
  Parent PID = 1
labex:os_practical_4/ $ ./orphan
Parent Process:
  PID = 8858
  Exiting parent...
labex:os_practical_4/ $ Child Process (Orphan):
  PID = 8859
  Parent PID (now 1 or init) = 1

labex:os_practical_4/ $ ps aux | grep orphan
labex:os_practical_4/ $ ./zombie
Parent Process:
  PID = 9071
  Sleeping for 30 seconds (check with 'ps aux | grep zombie' in another terminal)
Child Process (Zombie):
  PID = 9072
  Exiting child...

  Parent exiting...
labex:os_practical_4/ $
labex:os_practical_4/ $ ps aux | grep zombie
labex       9191  0.0  0.0   3464  1660 pts/13   S+   23:51   0:00 grep --color=auto --exclude-dir=.bzr --exclude-dir=CVS -
 --exclude-dir=.svn --exclude-dir=.idea --exclude-dir=.tox zombie
labex:os_practical_4/ $
```

❖ **Conclusion**: In this practical, we conclude that Linux commands like pkill, kill all, and kill effectively reduce system load by terminating processes based on their name or ID, improving system performance and stability.

❖ **Discussion Questions:**

1. **What is the concept of pointers in C?**
2. **What is the difference between malloc and calloc in C?**
3. **What is the purpose of the fork() system call in Unix?**
4. **What is the difference between struct and union in C?**
5. **What does grep do in Linux?**

## ❖ References:

https://www.geeksforgeeks.org/fork-system-call-in-operating-system/?ref=ml_lbp
https://www.scaler.com/topics/fork-system-call/

**Date:____/____/2026**

**Signature**
Course Coordinator
B.Tech CSE(AIML)
Sem: 4 / 2025-26