# BansilalRamnathAgarwalCharitableTrust’s

VishwakarmaInstituteofTechnology, Pune-37

*(AnautonomousInstituteofSavitribaiPhulePuneUniversity)*



**ARTIFICIAL INTELLIGENCE AND DATA SCIENCE**

|  |  |
| --- | --- |
| **Division** | **TY-B** |
| **Batch** | **1** |
| **GR-no** | **12210304** |
| **Rollno** | **38** |
| **Name** | **Sumedh Vilas Kokane** |

**Title:**. System Call

**Lab Assignment :** 3

**Problem Statement :**

**Zombie Process:**

A zombie process occurs when a child process has completed its execution (via the exit() system call), but its process entry still exists in the system's process table. This happens because the parent process has not yet read the child's exit status using the wait() or waitpid() system calls. Even though the child process has terminated, it still occupies a slot in the process table until the parent collects its termination status. This interim state is known as the "zombie" state.

**Demonstration in Code:**

* In the example code, after the first fork() call, the child process carries out its task (sorting the array) and then exits.
* The parent process, however, does not immediately call wait() to gather the child's exit status. Instead, it intentionally delays by sleeping for 5 seconds, allowing the child process to become a zombie.
* During this sleep interval, if you examine the process table using a command like ps, you would see the child process listed in a zombie state (denoted by "Z").
* Once the parent process wakes up and calls wait(), it collects the exit status of the child, removing it from the process table and thus terminating the zombie state.

**2. Orphan Process:**

An orphan process arises when a child process continues to run even after its parent process has terminated. In this situation, the orphaned process is automatically adopted by the init process (or a similar system process), which assumes responsibility for it. This ensures that the orphaned process can complete its execution, despite its parent no longer being active.

**Demonstration in Code:**

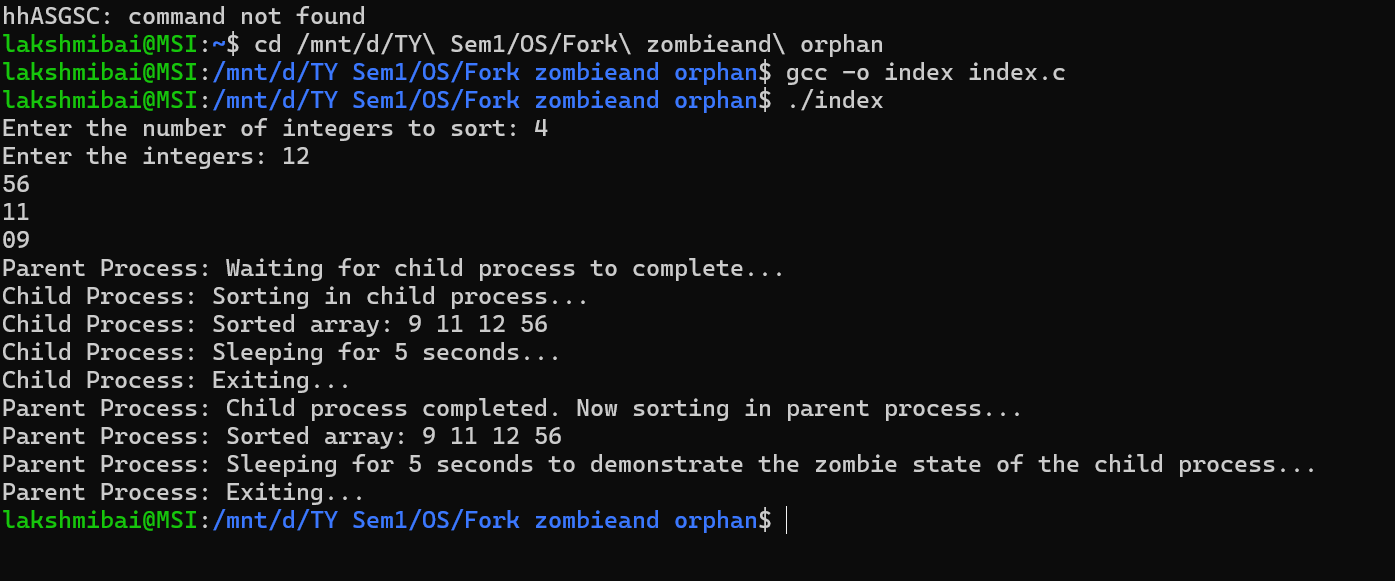
* After demonstrating the zombie state, the parent process creates a second child process using another fork() call.
* This time, the parent process exits immediately after forking, without waiting for the child to complete its task.
* As a result, the second child process becomes an orphan because its parent is no longer running.
* The orphaned process continues its execution and is adopted by the init process. It completes its work under the management of init, illustrating how the operating system handles orphan processes.

**Summary:**

* **Zombie Process:** Occurs when a child process has finished execution, but the parent process has not yet retrieved its exit status. The child process remains in the zombie state, visible in the process table, until the parent clears it up.
* **Orphan Process:** Occurs when a child process is still running, but its parent process has terminated. The orphaned process is adopted by the init process, allowing it to complete its execution under the supervision of the operating system.

**Code & Output**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/wait.h>  void **bubbleSort**(int arr[], int n) {      int i, j, temp;      for (i = 0; i < n-1; i++) {          for (j = 0; j < n-i-1; j++) {              if (arr[j] > arr[j+1]) {                  temp = arr[j];                  arr[j] = arr[j+1];                  arr[j+1] = temp;              }          }      }  }  int **main**() {      int n, i;  **pid\_t** pid;  **printf**("Enter the number of integers to sort: ");  **scanf**("%d", &n);      int arr[n];  **printf**("Enter the integers: ");      for (i = 0; i < n; i++) {  **scanf**("%d", &arr[i]);      }      pid = **fork**();      if (pid < 0) {  **fprintf**(**stderr**, "Fork Failed");          return 1;      }      else if (pid == 0) {  **printf**("Child Process: Sorting in child process...\n");  **bubbleSort**(arr, n);  **printf**("Child Process: Sorted array: ");          for (i = 0; i < n; i++) {  **printf**("%d ", arr[i]);          }  **printf**("\n");  **printf**("Child Process: Sleeping for 5 seconds...\n");  **sleep**(5);  **printf**("Child Process: Exiting...\n");  **exit**(0);      }      else {  **printf**("Parent Process: Waiting for child process to complete...\n");  **wait**(**NULL**);  **printf**("Parent Process: Child process completed. Now sorting in parent process...\n");  **bubbleSort**(arr, n);  **printf**("Parent Process: Sorted array: ");          for (i = 0; i < n; i++) {  **printf**("%d ", arr[i]);          }  **printf**("\n");  **printf**("Parent Process: Sleeping for 5 seconds to demonstrate the zombie state of the child process...\n");  **sleep**(5);  **printf**("Parent Process: Exiting...\n");      }      return 0;  } |

****