# Assignment 2a

**Title** : Process Control System Calls

**AIM:**

Process control system calls: The demonstration of FORK, EXECVE and WAIT system calls
along with zombie and orphan states.
Implement the C program in which main program accepts the integers to be sorted. Main
program uses the FORK system call to create a new process called a child process. Parent
process sorts the integers using sorting algorithm and waits for child process using WAIT
system call to sort the integers using any sorting algorithm. Also demonstrate zombie and
orphan states.
Implement the C program in which main program accepts an integer array. Main program uses
the FORK system call to create a new process called a child process. Parent process sorts an
integer array and passes the sorted array to child process through the command line arguments
of EXECVE system call. The child process uses EXECVE system call to load new program
which display array in reverse order.

**OBJECTIVE**:

This assignment covers the UNIX process control commonly called for process creation,
program execution and process termination. Also covers process model, including process
creation, process destruction, zombie and orphan processes.

**THEORY:**

Process in UNIX:
A process is the basic active entity in most operating-system models.
Process IDs

Each process in a Linux system is identified by its unique process ID, sometimes referred to as
pid. Process IDs are 16-bit numbers that are assigned sequentially by Linux as new processes
are created.
When referring to process IDs in a C or C++ program, always use the pid_t typedef, which is
defined in <sys/types.h>.A program can obtain the process ID of the process it’s running in
with the getpid() system call, and it can obtain the process ID of its parent process with the
getppid() system call.
Creating Processes
Two common techniques are used for creating a new process.
using system() function.
using fork() system calls.
1. Using system
The system function in the standard C library provides an easy way to execute a command
from within a program, much as if the command had been typed into a shell. In fact, system
creates a subprocess running the standard Bourne shell (/bin/sh)and hands the command to that
shell for execution.
The system function returns the exit status of the shell command. If the shell itself cannot be
run, system returns 127; if another error occurs, system returns –1.
2. Using fork
A process can create a new process by calling fork. The calling process becomes the parent,
and the created process is called the child. The fork function copies the parent's memory image
so that the new process receives a copy of the address space of the parent. Both processes
continue at the instruction after the fork statement (executing in their respective memory images).

## Zombie Processes

If a child process terminates while its parent is calling a wait function, the child process

vanishes and its termination status is passed to its parent via the wait call. But what happens
when a child process terminates and the parent is not calling wait? Does it simply vanish? No,
because then information about its termination—such as whether it exited normally and, if so,
what its exit status is—would be lost. Instead, when a child process terminates, is becomes a
zombie process.
A zombie process is a process that has terminated but has not been cleaned up yet. It is the
responsibility of the parent process to clean up its zombie children. The wait functions do this,
too, so it’s not necessary to track whether your child process is still executing before waiting
for it. Suppose, for instance, that a program forks a child process, performs some other
computations, and then calls wait. If the child process has not terminated at that point, the
parent process will block in the wait call until the child process finishes. If the child process
finishes before the parent process calls wait, the child process becomes a zombie. When the
parent process calls wait, the zombie child’s termination status is extracted, the child process is
deleted, and the wait call returns immediately.

## Orphan Process:

An Orphan Process is nearly the same thing which we see in real world. Orphan means
someone whose parents are dead. The same way this is a process, whose parents are dead, that
means parents are either terminated, killed or exited but the child process is still alive.
In Linux/Unix like operating systems, as soon as parents of any process are dead, re-parenting occurs, automatically. Re-parenting means processes whose parents are dead, means
Orphaned processes, are immediately adopted by special process. Thing to notice here is that
even after re-parenting, the process still remains Orphan as the parent which created the
process is dead, Reasons for Orphan Processes:
A process can be orphaned either intentionally or unintentionally. Sometime a parent

process exits/terminates or crashes leaving the child process still running, and then they become orphans.
Also, a process can be intentionally orphaned just to keep it running. For
example when you need to run a job in the background which need any manual
intervention
and going to take long time, then you detach it from user session and leave it there.
Same way,
when you need to run a process in the background for infinite time, you need to do the
same
thing. Processes running in the background like this are known as daemon process.

## Daemon Process:

It is a process that runs in the background, rather than under the direct control of a user; they are usually initiated as background processes.

## **Code:-**

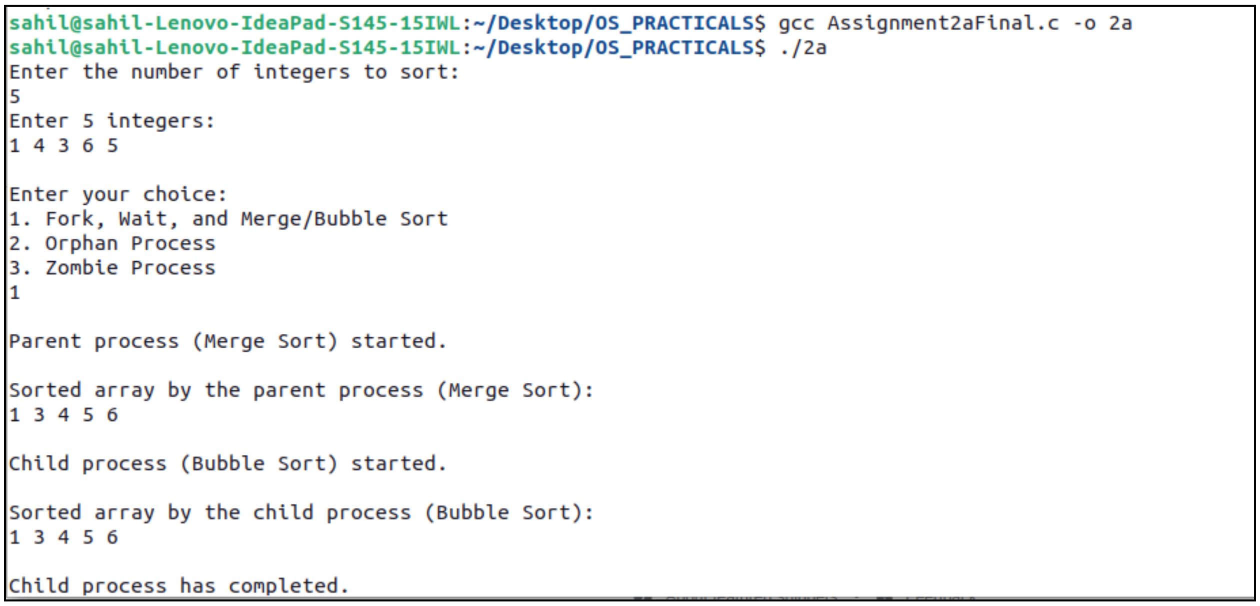
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

// Bubble Sort
void bubbleSort(int arr[], int n) {
        int temp, i, j;
        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;
        }
        }
        }
}

// Merge Sort
```

```
void merge(int arr[], int l, int m, int r) {
        int i, j, k;
        int n1 = m - l + 1;
        int n2 = r - m;

        int L[n1], R[n2];

        for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
        for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

        i = 0;
        j = 0;
        k = l;

        while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
        arr[k] = L[i];
        i++;
        }
        else {
        arr[k] = R[j];
        j++;
        }
        k++;
        }

        while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
        }

        while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
        }
}
```

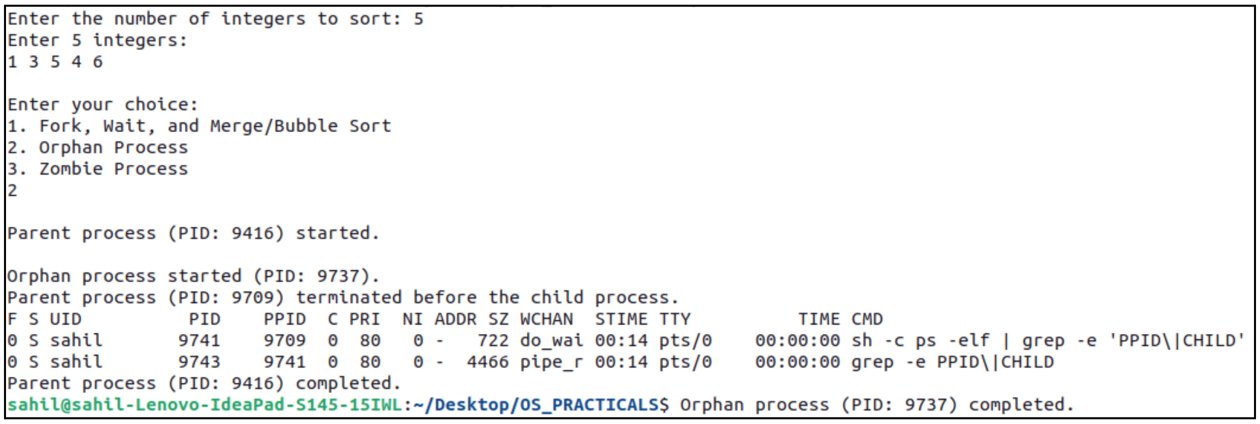
```
void mergeSort(int arr[], int l, int r) {
        if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
        }
}

int main() {
        int n, i;
        printf("Enter the number of integers to sort: ");
        scanf("%d", &n);
        int arr[n];

        printf("Enter %d integers:\n", n);
        for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
        }

        int choice;
        printf("\nEnter your choice:\n");
        printf("1. Fork, Wait, and Merge/Bubble Sort\n");
        printf("2. Orphan Process\n");
        printf("3. Zombie Process\n");
        scanf("%d", &choice);

        switch (choice) {
        case 1: {
        pid_t pid = fork();

        if (pid < 0) {
                printf("Fork failed.\n");
                exit(1);
        } else if (pid == 0) {
                printf("\nChild process (Bubble Sort) started.\n");
                bubbleSort(arr, n);
                printf("\nSorted array by the child process (Bubble Sort):\n");
                for (i = 0; i < n; i++)
```

```
            printf("%d ", arr[i]);
            printf("\n");
        } else {
            printf("\nParent process (Merge Sort) started.\n");
            mergeSort(arr, 0, n - 1);
            printf("\nSorted array by the parent process (Merge Sort):\n");
            for (i = 0; i < n; i++)
            printf("%d ", arr[i]);
            printf("\n");
            wait(NULL);
            printf("\nChild process has completed.\n");
        }
        break;
        }
        case 2: {
        pid_t pid = fork();

        if (pid < 0) {
            printf("Fork failed.\n");
            exit(1);
        } else if (pid == 0) {
            // Orphan process
            printf("\nOrphan process started (PID: %d).\n", getpid());
            printf("Parent process (PID: %d) terminated before the child process.\n",
    getppid());

            // Sleep to create an orphan process
            sleep(10);

            printf("Orphan process (PID: %d) completed.\n", getpid());
        } else {
            // Parent process
            printf("\nParent process (PID: %d) started.\n", getppid());

            sleep(5);
         system("ps -elf | grep -e 'PPID\\|CHILD'");

            printf("Parent process (PID: %d) completed.\n", getppid());
        }
        break;
        }
        case 3: {
        pid_t pid = fork();

        if (pid < 0) {
                printf("Fork failed.\n");
                exit(1);
        } else if (pid == 0) {
                // Child process
                printf("\nChild process (Zombie) started.\n");

                printf("Child process (PID: %d) completed.\n", getpid());
        } else {
                // Parent process
         sleep(10);
         char command[100];
         sprintf(command,"ps -elf | grep %d",getpid());
         system(command);
                printf("\nParent process (Zombie) started.\n");
                printf("Parent process will sleep to create a Zombie (PID: %d).\n", pid);
         printf("Parent process (PID: %d) completed.\n", getppid());


         wait(NULL);

        }
                 break;
        }
        default:
        printf("Invalid choice.\n");
        break;
        }

        return 0;
}
```

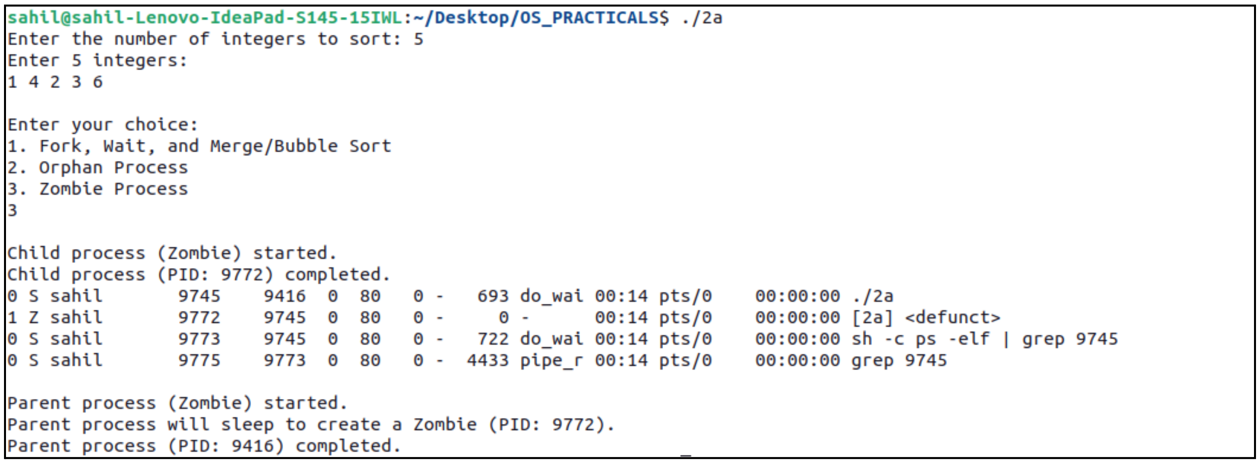
## Output:-



```
sahil@sahil-Lenovo-IdeaPad-S145-15IWL:~/Desktop/OS_PRACTICALS$ gcc Assignment2aFinal.c -o 2a
sahil@sahil-Lenovo-IdeaPad-S145-15IWL:~/Desktop/OS_PRACTICALS$ ./2a
Enter the number of integers to sort:
5
Enter 5 integers:
1 4 3 6 5

Enter your choice:
1. Fork, Wait, and Merge/Bubble Sort
2. Orphan Process
3. Zombie Process
1

Parent process (Merge Sort) started.

Sorted array by the parent process (Merge Sort):
1 3 4 5 6

Child process (Bubble Sort) started.

Sorted array by the child process (Bubble Sort):
1 3 4 5 6

Child process has completed.
```



```
Enter the number of integers to sort: 5
Enter 5 integers:
1 3 5 4 6

Enter your choice:
1. Fork, Wait, and Merge/Bubble Sort
2. Orphan Process
3. Zombie Process
2

Parent process (PID: 9416) started.

Orphan process started (PID: 9737).
Parent process (PID: 9709) terminated before the child process.
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S sahil       9741    9709  0  80   0 -   722 do_wai 00:14 pts/0    00:00:00 sh -c ps -elf | grep -e 'PPID\|CHILD'
0 S sahil       9743    9741  0  80   0 -  4466 pipe_r 00:14 pts/0    00:00:00 grep -e PPID\|CHILD
Parent process (PID: 9416) completed.
sahil@sahil-Lenovo-IdeaPad-S145-15IWL:~/Desktop/OS_PRACTICALS$ Orphan process (PID: 9737) completed.
```



```
sahil@sahil-Lenovo-IdeaPad-S145-15IWL:~/Desktop/OS_PRACTICALS$ ./2a
Enter the number of integers to sort: 5
Enter 5 integers:
1 4 2 3 6

Enter your choice:
1. Fork, Wait, and Merge/Bubble Sort
2. Orphan Process
3. Zombie Process
3

Child process (Zombie) started.
Child process (PID: 9772) completed.
0 S sahil       9745    9416  0  80   0 -   693 do_wai 00:14 pts/0    00:00:00 ./2a
1 Z sahil       9772    9745  0  80   0 -     0 -      00:14 pts/0    00:00:00 [2a] <defunct>
0 S sahil       9773    9745  0  80   0 -   722 do_wai 00:14 pts/0    00:00:00 sh -c ps -elf | grep 9745
0 S sahil       9775    9773  0  80   0 -  4433 pipe_r 00:14 pts/0    00:00:00 grep 9745

Parent process (Zombie) started.
Parent process will sleep to create a Zombie (PID: 9772).
Parent process (PID: 9416) completed.
```