

**DA-IICT**

**IT215 LAB4**

**Name:** Patel Raj Kamleshbhai

**ID:** 201901306

## Example Code Questions

**Q.** Find out what do we mean by capital letters S, S+, R etc. in the output of ps x command. Does it indicate any information about zombie processes?

**Ans.** Capital letters S,S+,R etc in the output of ps x command are process state codes with each having specific indication.

- **S** : Interruptible sleep ( Waiting for an event to complete)
- **R**: Running Process
- **S+**: Interruptible sleep in foreground process group.
- **D**: Uninterruptible sleep
- **T**: stopped
- **Z**: Defunct/ Zombie process

Yes, If process status code is 'Z' that indicates process is in Zombie state

## PROGRAM -1

### Part-1: Proc-Zombie.c

**CODE:**

```
/* proc-zombi.c */
#include <sys/types.h>
#include <unistd.h> int
main() {
    if (fork() > 0)
        { while (1);
        }
    return 0;
}
```

**OUTPUT:** In the output processes labeled as <defunct>.

**QUE:** Explain how the program above would lead to a defunct process.

**ANS:** A **zombie process** or **defunct process** is a process that has completed execution (via the exit system call) but still has an entry in the process table.

In our code, When we perform fork() while checking 'if' condition, 'if' condition will be satisfied for Parent process since fork() call return pid of new child formed which will be any positive number. After entering into "if" condition, parent process will go into an infinite loop (not terminated). On the Other hand child process formed due to fork() call will encounter "return 0;" statement and will get terminated while parent is executing infinite loop.

Since Child is terminated/completed execution but its parent is still alive so it will become zombie process and keep its entry in process table in case parent asks for it in future.

## **Part-2: proc-Orphan.c**

**CODE:**

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

#include <unistd.h> int

main() {

    if (fork() == 0)

        { while (1);

          }

    return 0;

}
```

**OUTPUT:** In the output, as the child process become orphan, Init Process will be the parent process of child process (after the termination of its parent process).

## PROGRAM-2

### CODE:

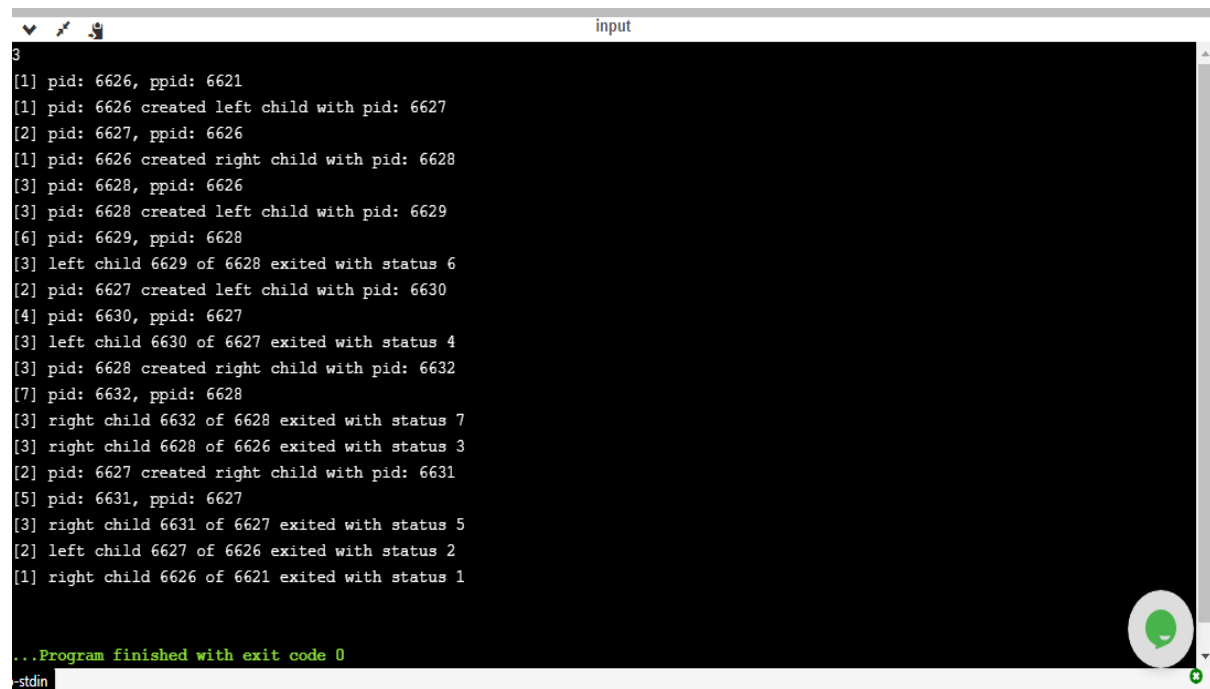
```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<signal.h>
int main()
{
    int n;
    scanf("%d",&n);
    int a=1,status;
    int i=1;
    while(i<n)
    {
        int b=a; //b will indicate current node position
        printf("[%d] pid: %d, ppid: %d\n",b,getpid(),getppid());
        if(fork()==0)
        {
            a=a*2; //a will indicate newly formed left child's position
            printf("[%d] pid: %d created left child with pid: %d\n",b,getppid(),getpid());
        }
        if(a==b)
        {
            if(fork()==0)
            {
                a=(a*2)+1; //a will indicate newly formed right child's position
                printf("[%d] pid: %d created right child with pid: %d\n",b,getppid(),getpid());
            }
            if(a==b) //Basically parent node will wait here and then terminate and not propagate
            further.only its child node will propagate further.
            {
                wait(&status); //waiting for both child to get terminated
                wait(&status);
                if(a%2==1) // if condition is checked to know that current node is weather left or right child of its
                parent node.
                printf("[%d] right child %d of %d exited with status %d\n",b,getpid(),getppid(),a);
                else
                printf("[%d] left child %d of %d exited with status %d\n",b,getpid(),getppid(),a);
                exit(0); //current node is terminated
            }
        }
    }
}
```

```

}
i++;
}
printf("[%d] pid: %d, ppid: %d\n",a,getpid(),getppid());
if(a%2==1)
printf("[%d] right child %d of %d exited with status %d\n",n,getpid(),getppid(),a);
else
printf("[%d] left child %d of %d exited with status %d\n",n,getpid(),getppid(),a);
exit(0);
return 0;
}

```

## OUTPUT: (Tree of Height = 3)



```

input
3
[1] pid: 6626, ppid: 6621
[1] pid: 6626 created left child with pid: 6627
[2] pid: 6627, ppid: 6626
[1] pid: 6626 created right child with pid: 6628
[3] pid: 6628, ppid: 6626
[3] pid: 6628 created left child with pid: 6629
[6] pid: 6629, ppid: 6628
[3] left child 6629 of 6628 exited with status 6
[2] pid: 6627 created left child with pid: 6630
[4] pid: 6630, ppid: 6627
[3] left child 6630 of 6627 exited with status 4
[3] pid: 6628 created right child with pid: 6632
[7] pid: 6632, ppid: 6628
[3] right child 6632 of 6628 exited with status 7
[3] right child 6628 of 6626 exited with status 3
[2] pid: 6627 created right child with pid: 6631
[5] pid: 6631, ppid: 6627
[3] right child 6631 of 6627 exited with status 5
[2] left child 6627 of 6626 exited with status 2
[1] right child 6626 of 6621 exited with status 1

...Program finished with exit code 0
stdin

```

## PROGRAM-3

### CODE:

```
#include<sys/types.h>
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>
#include<sys/wait.h>
#include<unistd.h>
void merge(int parr[],int l,int m,int r)
{
    int n1=m-l+1;
    int n2=r-m;
    int L[n1],R[n2];
    for(int i=0;i<n1;i++)
        L[i]=parr[l+i];
    for(int j=0;j<n2;j++)
        R[j]=parr[m+1+j];
    int i=0;
    int j=0;
    int k=l;
    while(i<n1 && j<n2)
    {
        if(L[i]<=R[j])
        {
            parr[k]=L[i];
            i++;
        }
        else
        {
            parr[k]=R[j];
            j++;
        }
        k++;
    }
    while(i<n1)
    {
        parr[k]=L[i];
        i++;
        k++;
    }
    while(j<n2)
    {
        parr[k]=R[j];
        j++;
        k++;
    }
}
void Merge_sort(int parr[],int l,int r) //simple merge-sort algorithm
```

```

{
if(l<r)
{
int mid =(l+r)/2;
Merge_sort(parr,l,mid);
Merge_sort(parr,mid+1,r);
merge(parr,l,mid,r);
}
else
return;
}
void Bubble_sort(int carr[],int n) //simple bubble sort algorithm
{
for(int i=0;i<n;i++)
{
for(int j=0;j<n-i-1;j++)
{
if(carr[j]>carr[j+1])
{
int temp=carr[j];
carr[j]=carr[j+1];
carr[j+1]=temp;
}
}
}
printf("Output By Child process: ");
for(int i=0;i<n;i++) // printing output of child process
{
printf("%d ",carr[i]);
}
printf("\n");
}
int main()
{
int pid,child_pid;
int n,status;
printf("Enter size of array:");
scanf("%d",&n);
int parr[n];
int carr[n];
int arr[n];
printf("\nEnter numbers: ");
for(int i=0;i<n;i++)
{
scanf("%d",&arr[i]);
parr[i]=arr[i];
carr[i]=arr[i];
}
printf("\nFork call is executed\n");
pid=getpid();
child_pid=fork();

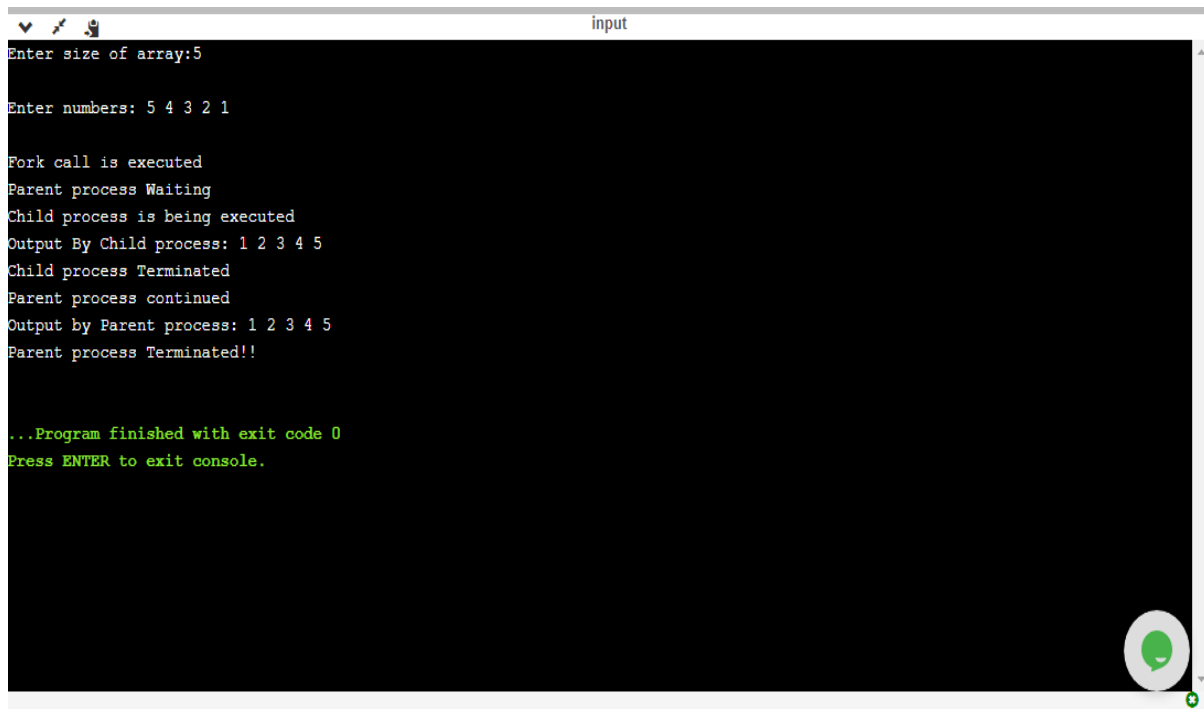
```

```

if(child_pid<0)
{
printf("Child process formation Failed\n");
return 0;
}
if(child_pid==0)
{
printf("Child process is being executed\n");
Bubble_sort(carr,n);
printf("Child process Terminated\n");
exit(0);
}
else
{ printf("Parent process Waiting\n");
wait(&status);
printf("Parent process continued\n");
Merge_sort(parr,0,n-1);
printf("Output by Parent process: ");
for(int i=0;i<n;i++)
printf("%d ",parr[i]);
printf("\n");
printf("Parent process Terminated!!\n");
exit(0);
}
}

```

## OUTPUT:



```

input
Enter size of array:5
Enter numbers: 5 4 3 2 1

Fork call is executed
Parent process Waiting
Child process is being executed
Output By Child process: 1 2 3 4 5
Child process Terminated
Parent process continued
Output by Parent process: 1 2 3 4 5
Parent process Terminated!!

...Program finished with exit code 0
Press ENTER to exit console.

```



## PSEUDO-CODE:

1. Input array of numbers and make separate copy of that array for parent and child respectively.
2. Perform fork() call (child\_pid=fork();)
3. Perform wait() call for parent process. So that child process execute first.
4. Execute child process i.e. sort child\_array with any sorting algorithm lets say Bubble sort. And then print the sorted child\_array within child process and then terminate child process. (Note: Child process will become Zombie process after termination)
5. After the child process is terminated its Parent process will continue its execution and sort parent\_array elements with any other sorting Algorithm lets say merge sort. And then terminate Parent process.

**The above code demonstrates Zombie process quite well but for Orphan process we have to make small change in the code mentioned below:**

```
if(child_pid==0)
{
    sleep(30);
    printf("Child process is being executed\n");
    Bubble_sort(carr,n);
    printf("Child process Terminated\n");
    exit(0);
}
else
{
    printf("Parent process continued\n");
    Merge_sort(parr,0,n-1);
    printf("Output by Parent process: ");
    for(int i=0;i<n;i++)
        printf("%d ",parr[i]);
    printf("\n");
    printf("Parent process Terminated!!\n");
    exit(0);
}
```

Change: "sleep(30)" call is introduced inside execution of child process and wait() call of Parent process is removed.

Now In this case, Sleep(30) call will delay the execution of child process and so Parent process will continue its execution and will get terminated by exit(0) call before child process. As soon as parent process die i.e it gets terminated child process will enter into Orphan state and will make "init process" as its parent.