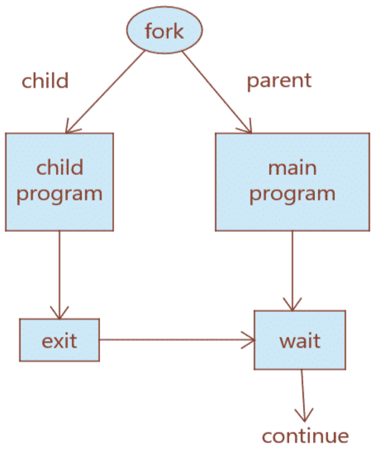
VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

CSE2005 OS LAB (L3 + L4): Exp. No. 4

Demonstration of process system calls, zombie & orphan processes

Fork system call is used for creating a new process, which is called ***child process***, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc (program counter), same CPU registers, same open files which use in the parent process.
It takes no parameters and returns an integer value. Below are different values returned by fork().
Negative Value: creation of a child process was unsuccessful.
Zero: Returned to the newly created child process.
Positive value: Returned to parent or caller. The value contains process ID of newly created child process.



Q1. Write programs using the following system calls of UNIX operating system:

fork, getpid, getppid, exit

Aim :
To write a C program using the system calls - fork, getpid, exit

Algorithm :

1. Start the program, Include header files
2. Check if result of fork is successful
3. if fork() returns a negative value, the creation of a child process was unsuccessful
4. if fork() returns a zero value, the new child process is created
5. A process can use function getpid () to retrieve the process ID assigned to this process.

6. Stop the program

SYSTEM CALLS USING FORK, GETPID,GETPPID, EXIT

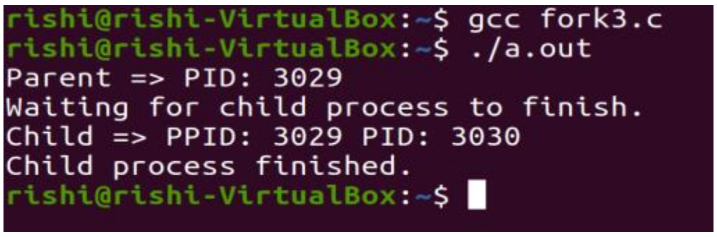
Source Code:-

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

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

  if(pid == 0)
  {
    printf("Child => PPID: %d PID: %d\n", getppid(), getpid());
    exit(EXIT_SUCCESS);
  }
  else if(pid > 0)
  {
    printf("Parent => PID: %d\n", getpid());
    printf("Waiting for child process to finish.\n");
    wait(NULL);
    printf("Child process finished.\n");
  }
  Else
  {
    printf("Unable to create child process.\n");
   }
   return EXIT_SUCCESS;
}
```

OUTPUT:-



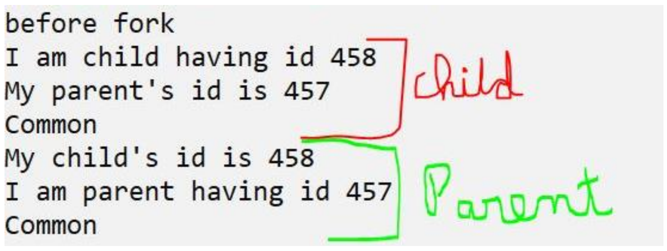
```
rishi@rishi-VirtualBox:~$ gcc fork3.c
rishi@rishi-VirtualBox:~$ ./a.out
Parent => PID: 3029
Waiting for child process to finish.
Child => PPID: 3029 PID: 3030
Child process finished.
rishi@rishi-VirtualBox:~$
```

Q2. Program for wait () system call which makes the parent process wait for the child to finish.

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

```
#include<sys/wait.h>
    int main()
        {
        pid_t p;
        printf("before fork\n");
        p=fork();
        if(p==0)//child
        {
        printf("I am child having id %d\n",getpid());
        printf("My parent's id is %d\n",getppid());
        }
        else//parent
        {
        wait(NULL);
        printf("My child's id is %d\n",p);
        printf("I am parent having id %d\n",getpid());
        }
        printf("Common\n");
    }
```

Output



```
before fork
I am child having id 458
My parent's id is 457
Common
My child's id is 458
I am parent having id 457
Common
```

How it works?

The execution begins by printing “before fork”. Then fork() system call creates a child process. wait() system call is added to the parent section of the code. Hence, the moment processor starts processing the parent, the parent process is suspended because the very first statement is wait(NULL). Thus, first, the child process runs, and the output lines are all corresponding to the child process. Once the child process finishes, parent resumes and prints all its printf () statements. The NULL inside the wait () means that we are not interested to know the status of change of state of child process.

Your task:

Q3. Create a file named my file.txt that contains the following four lines: Child 1 reads this line Child 2 reads this line Child 3 reads this line Child 4 reads this line. Write a C program that forks four other processes. After forking the parent process goes into wait state and waits for the children to finish their execution. Each child process reads a line from the file my file.txt (Child 1 reads line 1, child 2 reads line 2, child 3 reads line 3 and child 4 reads line 4) and each prints the respective line. The lines can be printed in any order.

OR

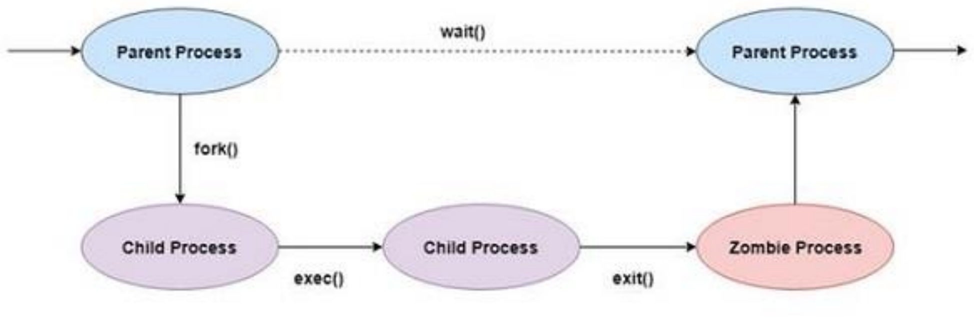
Q3. Create multiple child processes using fork () and Loop, also print the PID and PPID from the child processes.

Q4. **Demonstrates the creation and termination of a zombie process.**

Zombie Processes

A zombie process is a process whose execution is completed but it still has an entry in the process table. Zombie processes usually occur for child processes, as the parent process still needs to read its child's exit status. Once this is done using the wait system call, the zombie process is eliminated from the process table. This is known as reaping the zombie process.

A diagram that demonstrates the creation and termination of a zombie process is given as follows



Zombie Process in Linux

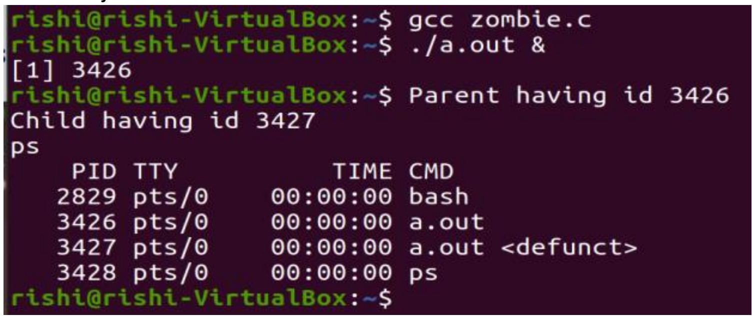
Zombie processes don't use any system resources but they do retain their process ID. If there are a lot of zombie processes, then all the available process ID's are monopolized by them. This prevents other processes from running as there are no process ID's available.

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

int main()

{
  pid_t t;
  t=fork();
  if(t==0)
  {
     printf("Child having id %d\n",getpid());
   }
   else
   {
      printf("Parent having id %d\n",getpid());
      sleep(15);   // Parent sleeps. Run the ps command during this time
```

```
        }
    }
```



```
rishi@rishi-VirtualBox:~$ gcc zombie.c
rishi@rishi-VirtualBox:~$ ./a.out &
[1] 3426
rishi@rishi-VirtualBox:~$ Parent having id 3426
Child having id 3427
ps
    PID TTY          TIME CMD
   2829 pts/0    00:00:00 bash
   3426 pts/0    00:00:00 a.out
   3427 pts/0    00:00:00 a.out <defunct>
   3428 pts/0    00:00:00 ps
rishi@rishi-VirtualBox:~$
```

How it works?

Now Compile the program and run it using ./a.out &. The '&' sign makes it run in the background. After you see the output of both the *printf* statement, note that the parent will go into sleep for 15 seconds. During this time type the command '*ps'.* The output will contain a process with ***defunct*** written in the end. Hence, this is the child process (can be verified from PID) which has become *zombie*.

How to prevent the creation of a Zombie Process?

In order to prevent this use the wait() system call in the parent process. The *wait()* system call makes the parent process wait for the child process to change state. Click here for more details

Q5. **Demonstrates the creation of an orphan process.**

Orphan Processes

Orphan processes are those processes that are still running even though their parent process has terminated or finished. A process can be orphaned intentionally or unintentionally. An unintentionally orphaned process is created when its parent process crashes or terminates.

An orphan process is a process whose parent has finished. Suppose P1 and P2 are two process such that P1 is the parent process and P2 is the child process of P1. Now, if P1 finishes before P2 finishes, then P2 becomes an orphan process. The following programs we will see how to create an orphan process.

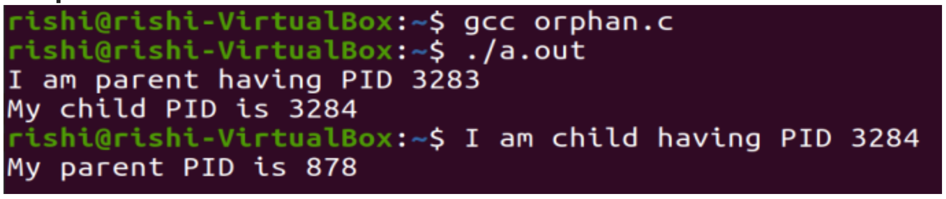
Program to create an orphan process

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

int main()
{
pid_t p;
p=fork();
    if(p==0)
   {
```

```
            sleep(5); //child goes to sleep and in the meantime parent terminates
            printf("I am child having PID %d\n",getpid());
            printf("My parent PID is %d\n",getppid());
        }
        else
        {
            printf("I am parent having PID %d\n",getpid());
            printf("My child PID is %d\n",p);
        }
        return 0;
    }
```

Output:



```
rishi@rishi-VirtualBox:~$ gcc orphan.c
rishi@rishi-VirtualBox:~$ ./a.out
I am parent having PID 3283
My child PID is 3284
rishi@rishi-VirtualBox:~$ I am child having PID 3284
My parent PID is 878
```

How it Works?

In this code, we add sleep (5) in the child section. This line of code makes the child process go to sleep for 5 seconds and the parent starts executing. Since, parent process has just two lines to print, which it does well within 5 seconds and it terminates. After 5 seconds when the child process wakes up, its parent has already terminated and hence the child becomes an orphan process. Hence, it prints the PID of its parent as 1 (1 or other process-id means the init process has been made its parent now) and not 138.

System Calls: Description

1. **fork()**
   - Create a child process
   - Current process (from where fork called) is Parent process
   - Returns positive, negative or 0 value
   - 0 → child process
   - Positive Number → Parent process
   - Negative Number → Error

2. **getpid()**
   - Print process ID (unique value)

3. **getppid()**
   - Print Parent process ID (unique value)

4. **wait()**
    - Put Parent process to hold until child completes
    - If child completes its task then parent resume its task after the wait() instruction

5. **sleep(int seconds)**
    - Put process into sleep mode for seconds until time elapsed or the calling of signal function

6. **kill(int pid, Signal type)**
    - If *pid* is positive, then signal *signal* is sent to the process with the ID specified by *pid*.
    - If *pid* equals 0, then *signal* is sent to every process in the process group of the calling process.
    - If *pid* equals -1, then *sig* is sent to every process for which the calling process has permission to send signals, except for process 1 (*init*).
    - If *pid* is less than -1, then *sig* is sent to every process in the process group whose ID is -*pid*.
    - **Signal Type:** SIGINT, SIGQUIT, SIGKILL and SIGTERM