CE054_LAB_03_Assignment

Aim : Process Creation and Termination (Use of fork, wait, getpid, and getppid System Calls).

Theory:

1. Fork System Call :
   fork() creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent, except for the following points:
   - The child has its own unique process ID, and this PID does not match the ID of any existing process group.
   - The child's parent process ID is the same as the parent's process ID.

   Syntax:

   ```
   #include<unistd.h>

   pid_t fork(void);
   ```

   - On success, the PID of the child process is returned in the parent, and 0 is returned in the child.
   - On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

   - Zombie Process :
   - A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process.
   - A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

   - Orphan Process :
   - A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.

2. Getpid() and Getppid System Call:
   - getpid() returns the process ID of the calling process.
     Syntax:

     ```
     pid_t getpid(void);
     ```

   - getppid() returns the process ID of the parent of the calling process.
     Sysntax:

     ```
     pid_t getppid(void);
     ```

3. Wait System Call :
   - wait system call is used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.
   - In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state.
   - If a child has already changed state, then this call returns immediately. Otherwise it blocks until either a child changes state or a signal handler interrupts the call.
   - On success, returns the process ID of the terminated child; on error, -1 is returned.

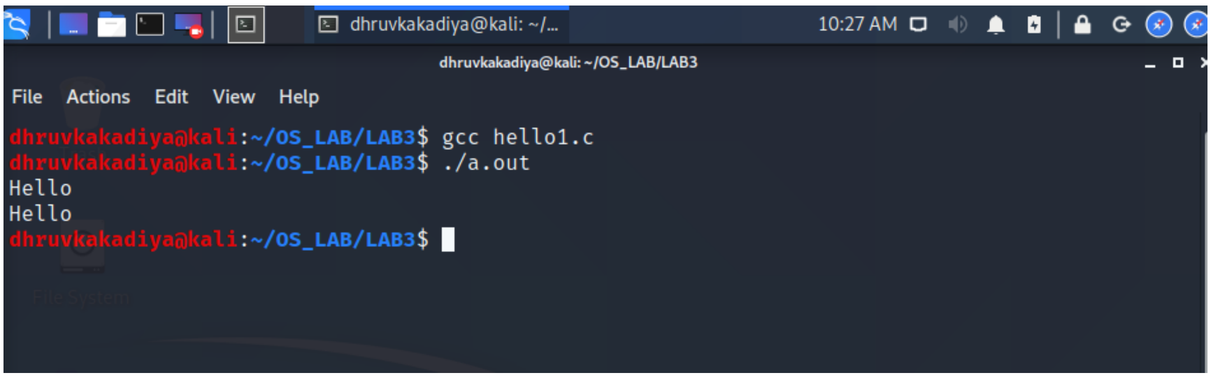
   Syntax:

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

   #include <sys/wait.h>

   pid_t wait(int *status);
   ```

4. Programs:

Task1 :

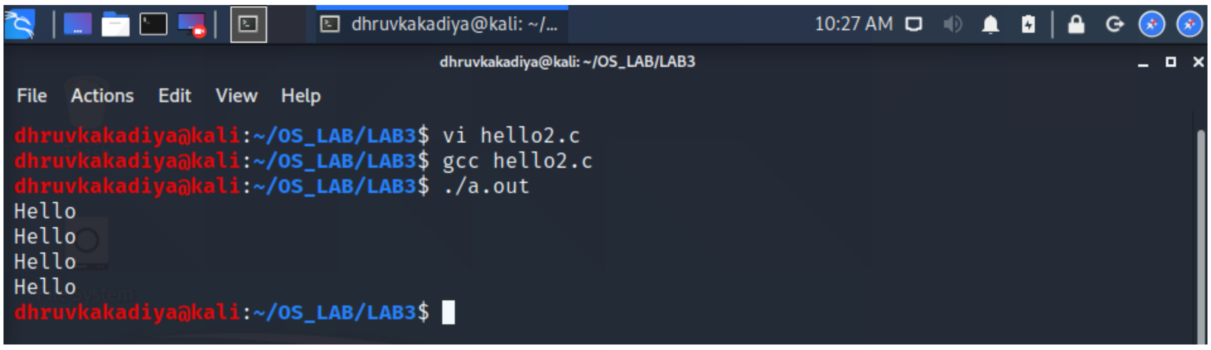
1. Call fork ones, observe and interpret outcomes.
   ```
   #include<sys/types.h>
   #include<unistd.h>
   #include<stdio.h>
   int main()
   {
       fork();
       printf("Hello\n");
       return 0;
   }
   ```


dhruvkakadiya@kali: ~/...
10:27 AM
dhruvkakadiya@kali: ~/OS_LAB/LAB3
File Actions Edit View Help

```
dhruvkakadiya@kali:~/OS_LAB/LAB3$ gcc hello1.c
dhruvkakadiya@kali:~/OS_LAB/LAB3$ ./a.out
Hello
Hello
dhruvkakadiya@kali:~/OS_LAB/LAB3$ 
```

2. Call fork twice. Observe and interpret outcomes.
   ```
   #include<sys/types.h>
   #include<unistd.h>
   #include<stdio.h>
   ```

```
int main()
{
    fork();
    fork();
    printf("Hello\n");
    return 0;
}
```

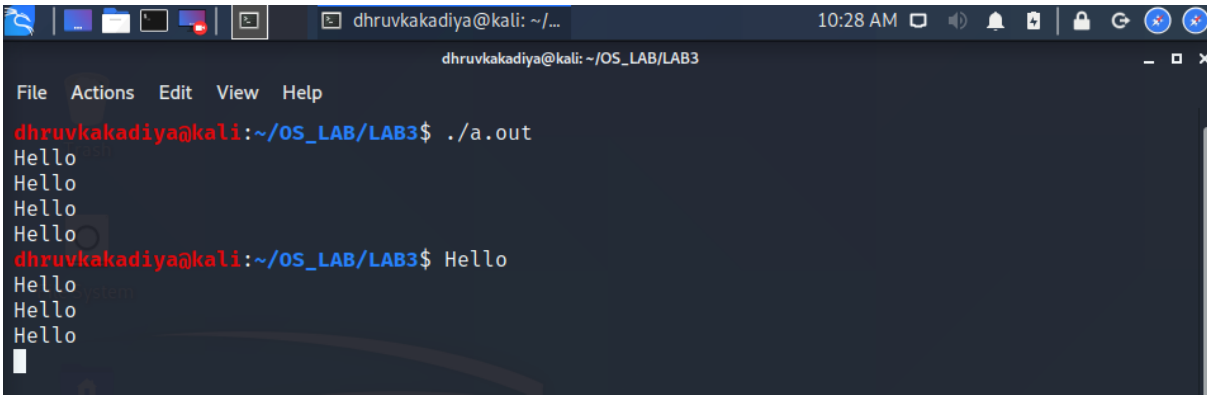


dhruvkakadiya@kali: ~/...
10:27 AM
dhruvkakadiya@kali: ~/OS_LAB/LAB3
File Actions Edit View Help

```
dhruvkakadiya@kali:~/OS_LAB/LAB3$ vi hello2.c
dhruvkakadiya@kali:~/OS_LAB/LAB3$ gcc hello2.c
dhruvkakadiya@kali:~/OS_LAB/LAB3$ ./a.out
Hello
Hello
Hello
Hello
dhruvkakadiya@kali:~/OS_LAB/LAB3$
```

3. Call fork thrice, observe and interpret outcomes.

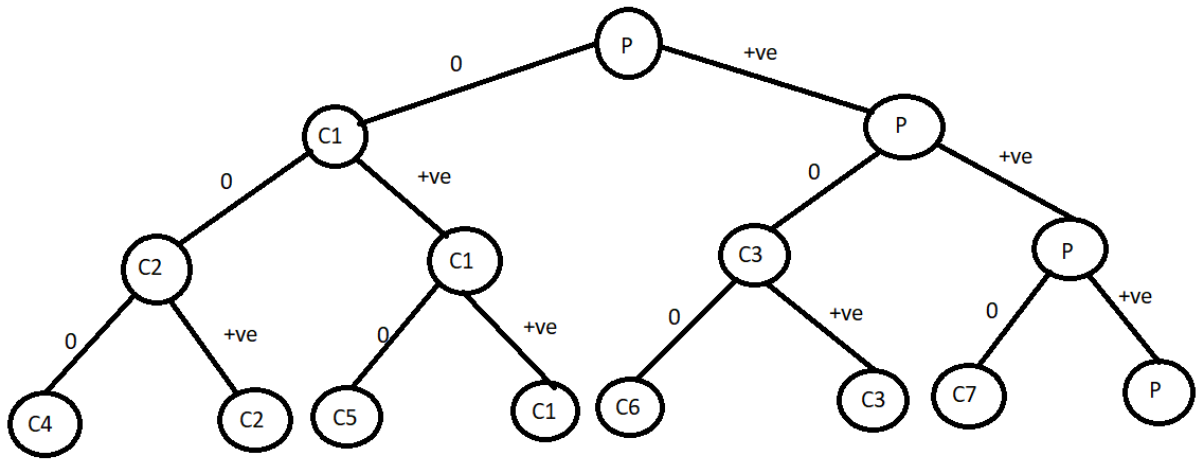
```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
int main()
{
    fork();
    fork();
    fork();
    printf("Hello\n");
    return 0;
}
```



dhruvkakadiya@kali: ~/...
10:28 AM
dhruvkakadiya@kali: ~/OS_LAB/LAB3
File Actions Edit View Help

```
dhruvkakadiya@kali:~/OS_LAB/LAB3$ ./a.out
Hello
Hello
Hello
Hello
dhruvkakadiya@kali:~/OS_LAB/LAB3$ Hello
Hello
Hello
Hello
```

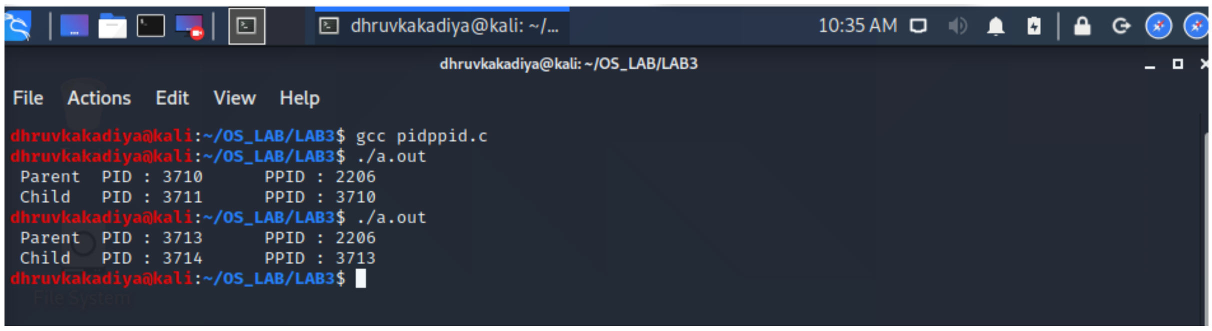
- Fork tree for 3 fork call:



Task2 :

1. Print PID and PPID for parent and child processes. Observe and interpret the outcomes.

```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
int main()
{
        int pid;
        pid = fork();
        if(pid == -1)
                printf("Error!\n");
        else if(pid > 0)
                printf(" Parent\t PID : %d\t PPID : %d\n",getpid(),getppid());
        else
                printf(" Child\t PID : %d\t PPID : %d\n",getpid(),getppid());
        return 0;
}
```

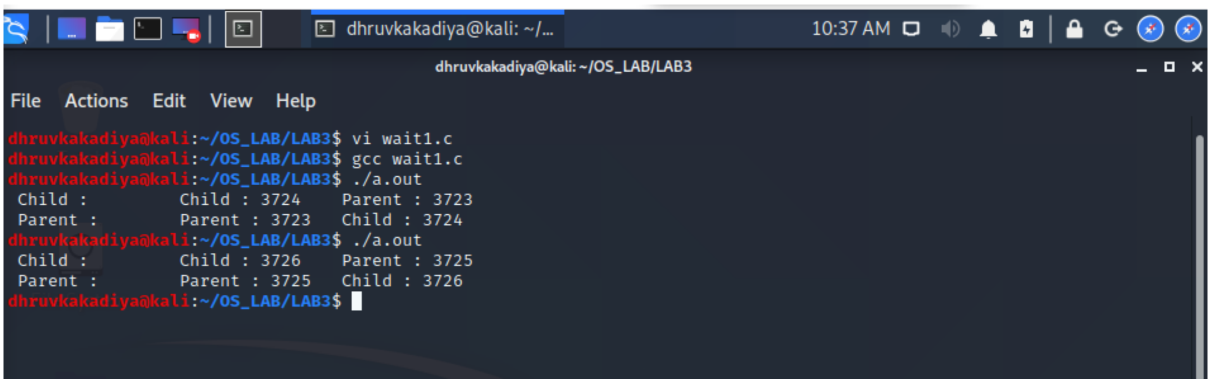


```
dhruvkakadiya@kali:~/OS_LAB/LAB3$ gcc pidppid.c
dhruvkakadiya@kali:~/OS_LAB/LAB3$ ./a.out
 Parent  PID : 3710      PPID : 2206
 Child   PID : 3711      PPID : 3710
dhruvkakadiya@kali:~/OS_LAB/LAB3$ ./a.out
 Parent  PID : 3713      PPID : 2206
 Child   PID : 3714      PPID : 3713
dhruvkakadiya@kali:~/OS_LAB/LAB3$ 
```

Task3 :

1. Add wait to the code of task 2. Observe and interpret the outcomes.

```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<sys/wait.h>
int main()
{
        int pid,status;
        pid = fork();
        if(pid == -1)
                printf("Error!\n");
        else if(pid > 0){
                wait(&status);
                printf(" Parent :\t Parent : %d\t Child : %d \n", getpid(),pid);
        }
        else
                printf(" Child :\t Child : %d\t Parent : %d\t \n",getpid(),getppid());
        return 0;
}
```



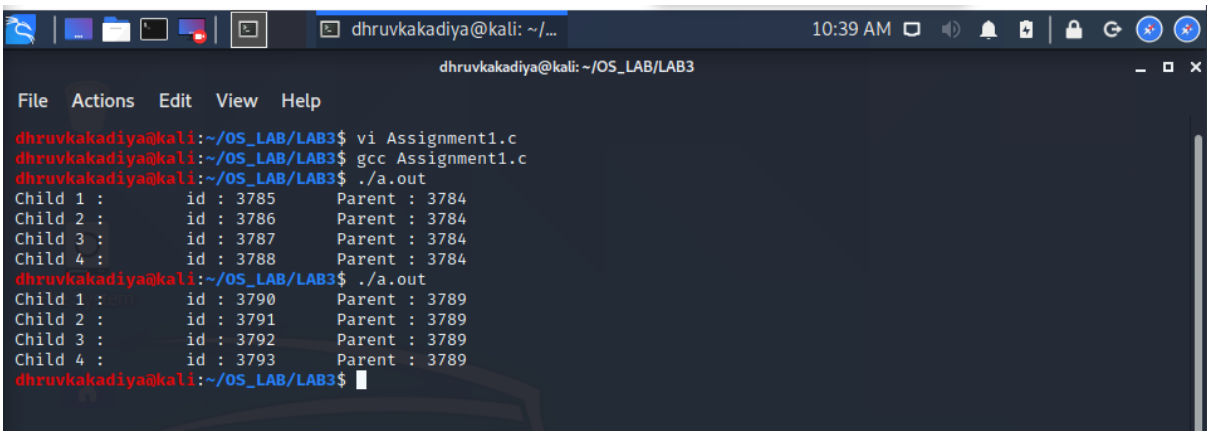
```
dhruvkakadiya@kali:~/OS_LAB/LAB3$ vi wait1.c
dhruvkakadiya@kali:~/OS_LAB/LAB3$ gcc wait1.c
dhruvkakadiya@kali:~/OS_LAB/LAB3$ ./a.out
 Child :         Child : 3724    Parent : 3723
 Parent :        Parent : 3723   Child : 3724
dhruvkakadiya@kali:~/OS_LAB/LAB3$ ./a.out
 Child :         Child : 3726    Parent : 3725
 Parent :        Parent : 3725   Child : 3726
dhruvkakadiya@kali:~/OS_LAB/LAB3$ 
```

Assignment :

1. Write a program to implement fan of n processes.

```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<sys/wait.h>
int main()
{
        int pid,status;
        pid = getpid();
        for(int i= 0; i<4; i++)
                if( pid > 0 ){
                        pid = fork();
                        wait(&status);
                        if (pid == 0)
```

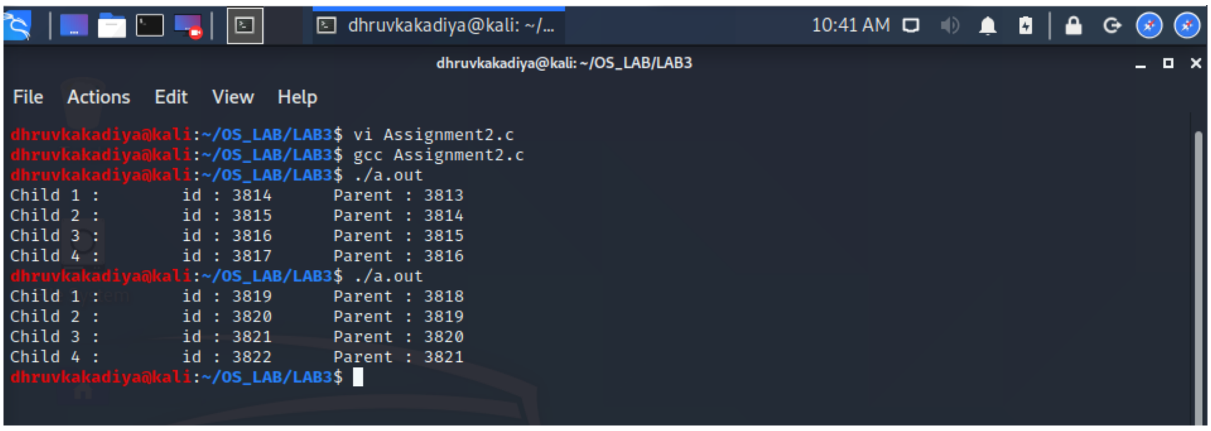
```
                                        printf("Child %d :\t id : %d\tParent : %d\n",i+1,
getpid(), getppid());
                        }
        return 0;
}
```



```
dhruvkakadiya@kali:~/OS_LAB/LAB3$ vi Assignment1.c
dhruvkakadiya@kali:~/OS_LAB/LAB3$ gcc Assignment1.c
dhruvkakadiya@kali:~/OS_LAB/LAB3$ ./a.out
Child 1 :        id : 3785      Parent : 3784
Child 2 :        id : 3786      Parent : 3784
Child 3 :        id : 3787      Parent : 3784
Child 4 :        id : 3788      Parent : 3784
dhruvkakadiya@kali:~/OS_LAB/LAB3$ ./a.out
Child 1 :        id : 3790      Parent : 3789
Child 2 :        id : 3791      Parent : 3789
Child 3 :        id : 3792      Parent : 3789
Child 4 :        id : 3793      Parent : 3789
dhruvkakadiya@kali:~/OS_LAB/LAB3$ 
```

2. Write a program to implement chain of n processes.

```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<sys/wait.h>
int main()
{
        int pid,status;
        pid = 0;
        for(int i= 0; i<4; i++)
                if( pid==0 ){
                        pid = fork();
                        wait(&status);
                        if (pid == 0)
                                printf("Child %d :\t id : %d\tParent :
%d\n",i+1,getpid(),getppid());
                }
        return 0;
}
```



```
dhruvkakadiya@kali:~/OS_LAB/LAB3$ vi Assignment2.c
dhruvkakadiya@kali:~/OS_LAB/LAB3$ gcc Assignment2.c
dhruvkakadiya@kali:~/OS_LAB/LAB3$ ./a.out
Child 1 :        id : 3814      Parent : 3813
Child 2 :        id : 3815      Parent : 3814
Child 3 :        id : 3816      Parent : 3815
Child 4 :        id : 3817      Parent : 3816
dhruvkakadiya@kali:~/OS_LAB/LAB3$ ./a.out
Child 1 :        id : 3819      Parent : 3818
Child 2 :        id : 3820      Parent : 3819
Child 3 :        id : 3821      Parent : 3820
Child 4 :        id : 3822      Parent : 3821
dhruvkakadiya@kali:~/OS_LAB/LAB3$ 
```