TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

A LAB REPORT ON

Process Concepts

SUBMITTED BY

Sushovan Shakya

THA075BEI046

SUBMITTED TO

Department of Electronics and Computer Engineering

26th August, 2021

PROCESS CONCEPTS

OBJECTIVE

- To understand various process concepts.

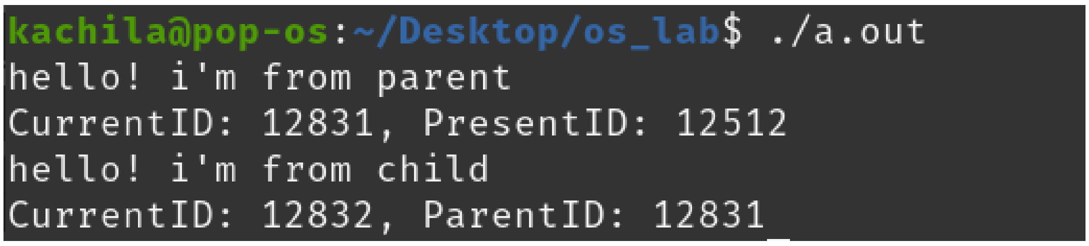
THEORY

fork() :
The fork system call creates a new process. When a program calls fork() there will be two copies of the programs running simultaneously. Each copy can do what it desires independent of the other. The fork() function returns the child's process id to the parent. It
returns 0 to the child and returns a negative number in case of failure.

Source Code

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

int main() {
      if(!fork()) {
            printf("hello! i'm from child \n");
            printf("CurrentID: %d, ParentID: %d\n", getpid(),
getppid());
      }
      else {
            printf("hello! i'm from parent\n");
            printf("CurrentID: %d, PresentID: %d\n", getpid(),
getppid());
      }
      return 0;
}
```

Output



```
kachila@pop-os:~/Desktop/os_lab$ ./a.out
hello! i'm from parent
CurrentID: 12831, PresentID: 12512
hello! i'm from child
CurrentID: 12832, ParentID: 12831
```

RESULT

The process id of each process was printed. getpid() printed the id of the current
process being executed while getppid() printed the parent id of the current process and
it
was found the id of the parent process is the same as the id of the child's parent id.

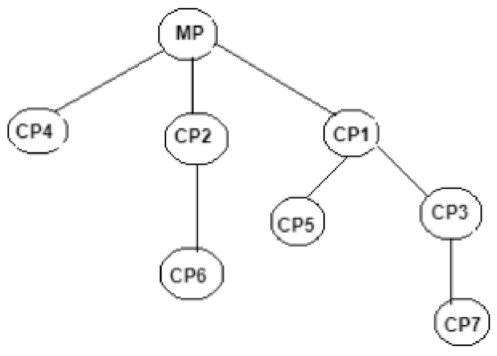
For the following program:

```
main()
{
      fork();
      fork();
      fork();
      printf("process details\n");
}
```

Result

Process detail will be printed 8 times. Initially, there is only the main process.
Upon calling fork() the first time, the process is divided into two, main process and child
process. On the second fork() call, both the child and main process are again divided into two
making four processes. Similarly upon the third fork() call all processes are divided again and
there will be a total of 8 processes.

The structure of process would be as follows.

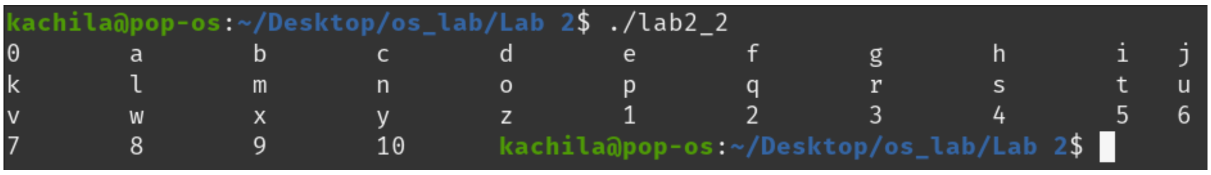


Program 2

Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    int i,d;
    int DEL1=1, DEL2=100000;
    char c;

    if(!fork()) {
        for(c='a';c<='z';c++) {
            printf("%c\t",c);
            fflush(stdout);

            for(d = 0; d < DEL1; d++);
        }
        exit(0);
    }
    else {
        for(i=0;i<=10;i++) {
            printf("%d\t",i);
            fflush(stdout);

            for(d = 0; d < DEL2; d++);
        }
        exit(0);
    }

}
```

Output



```
kachila@pop-os:~/Desktop/os_lab/Lab 2$ ./lab2_2
0       a       b       c       d       e       f       g       h       i   j
k       l       m       n       o       p       q       r       s       t   u
v       w       x       y       z       1       2       3       4       5   6
7       8       9       10      kachila@pop-os:~/Desktop/os_lab/Lab 2$
```

Result

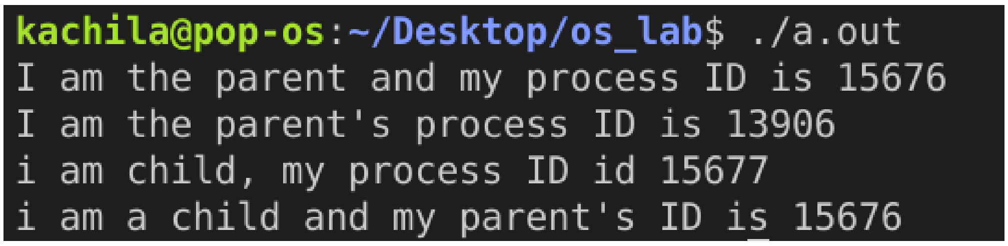
Initially, the values of DEL1 and DEL2 were closer to each other and it was observed that the main process was terminated before complete execution of the child process. When value of DEL2 was made comparatively larger than DEL1, it was observed that the execution for both process took almost similar time and main process was not terminated faster.

Program 3

Source Code

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

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

    if(pid==0)
    {
        printf("i am child, my process ID id %d\n",getpid());
        printf("i am a child and my parent's ID is %d\n",getppid());
        sleep(20);
    }

    else
    {
        //sleep(20);
        printf("I am the parent and my process ID is %d\n",getpid());
        printf("I am the parent's process ID is %d\n",getppid());
    }

    return 0;
}
```

Output



```
kachila@pop-os:~/Desktop/os_lab$ ./a.out
I am the parent and my process ID is 15676
I am the parent's process ID is 13906
i am child, my process ID id 15677
i am a child and my parent's ID is 15676
```

Result

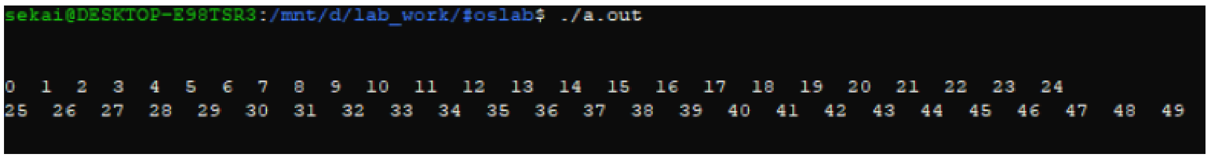
When sleep() is not called during the parent process, the parent process gets terminated before complete execution of the child process. The statements before sleep() gets executed before the main process is terminated as the child’s parent id is same as that of the parent process id, but when the parent id of child is displayed after the sleep(), the parent process id is not the same as child’s parent id as the main process is already terminated.
When sleep() is called on the parent process, it is delayed for a certain time.
The value of argument to sleep was increased and for time delay more than the child process, it was observed the parent process is not terminated before the child and both id displayed has the same value as child process is executed first.

Program 4

Source Code

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

int main() {
    int i = 0, j = 0, pid;
    pid = fork();

    if(pid == 0) {
        printf("\n");

        for(i = 0; i < 25; ++i)
            printf("%d\t", i);

        printf("\n");
    }
    else {
        if(pid > 0) {
            wait(0);
            for(j = 25; i < 500; ++j)
                printf("%d", i);
        }
    }

    return 0;
}
```

Output



```
sekai@DESKTOP-E98TSR3:/mnt/d/lab_work/#oslab$ ./a.out


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
```

Result

When wait() was not called, the child and parent process were executed at random. Upon calling wait process(), the execution of the child was done first i.e. the parent waits for the child to complete.

Zombie Process

A zombie process or defunct process is a process that has completed execution but still has an entry in the process table: it is a process in the "Terminated state". This occurs for the child processes, where the entry is still needed to allow the parent process to read its child's exit status: once the exit status is read via the wait system call, the zombie's entry is removed from the process table and it is said to be "reaped".

A child process always first becomes a zombie before being removed from the resource table. In most cases, under normal system operation zombies are immediately waited on by their parents and then reaped by the system.

Orphan Process

An orphan process is a computer process whose parent process has finished or terminated, though it remains running itself. Any orphaned process will be immediately adopted by the special init system process: the kernel sets the parent to init.

This operation is called re-parenting and occurs automatically. Even though technically the process has the "init" process as its parent, it is still called an orphan process since the process that originally created it no longer exists.

DISCUSSION AND CONCLUSION

In this section of lab, above four program was run on Linux operating system and the corresponding output was observed. We understood the fork() system call along with wait(), getpid(), getppid(), sleep() functions. We learned how a process can be further duplicated into many processes and how processes can be controlled through the use of some function.

Thus, in this section of the lab various process concepts were learned.