Operating System

Practical-3

CE092: Nevil Parmar

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

Theory:

Fork System Call:

#include <unistd.h>

pid_t fork(void);

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.

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.

Getpid and Getppid System Calls:

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

pid_t getpid(void);
pid_t getppid(void);

getpid() returns the process ID of the calling process.

getppid() returns the process ID of the parent of the calling process.

Wait System Call:

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

pid_t wait(int *_status_);

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.

A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.

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.

Tasks

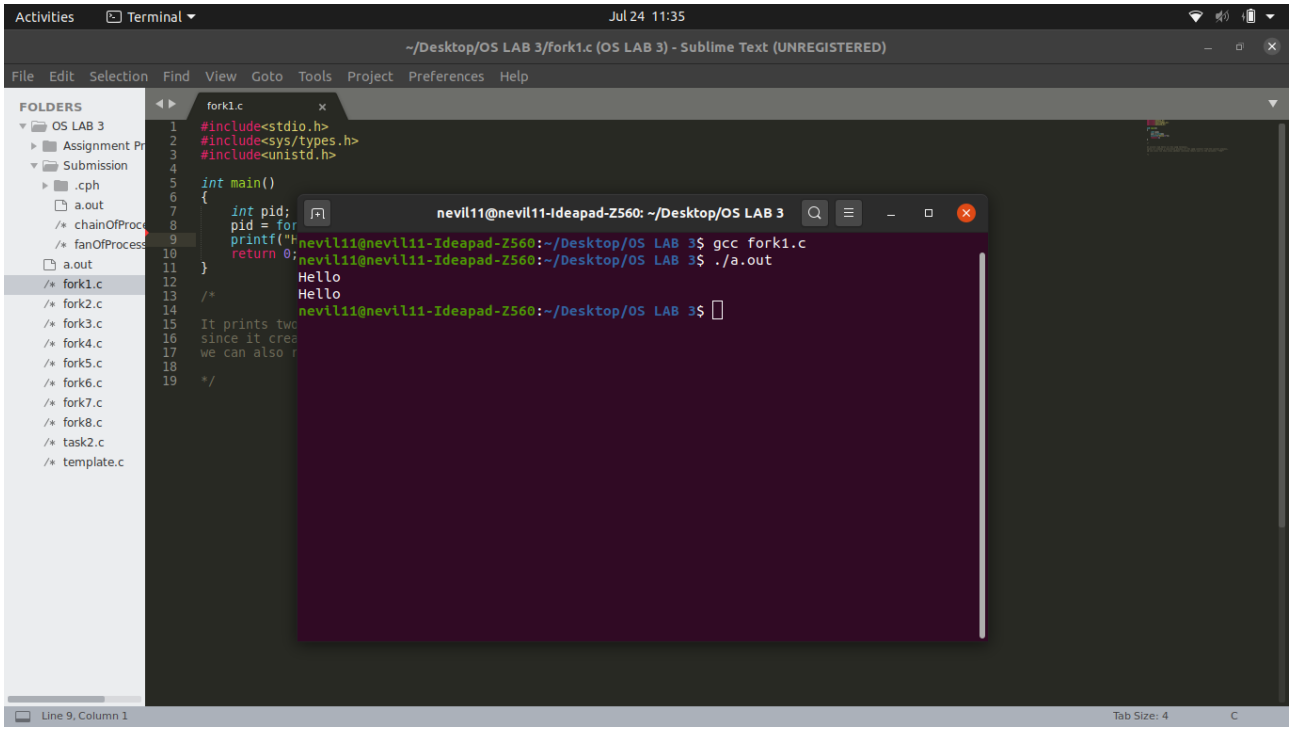
Task 1:

Call fork once, twice, thrice and print “Hello”. Observe and interpret the outcomes.

Subtask 1: Calling fork once

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

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

/*

It prints two hello on the same terminal.
since it creates a new child process of the same con-
text from the parent process.
we can also run this prog without catching re-
turn pid in the variable **PID**

*/
```

Output:

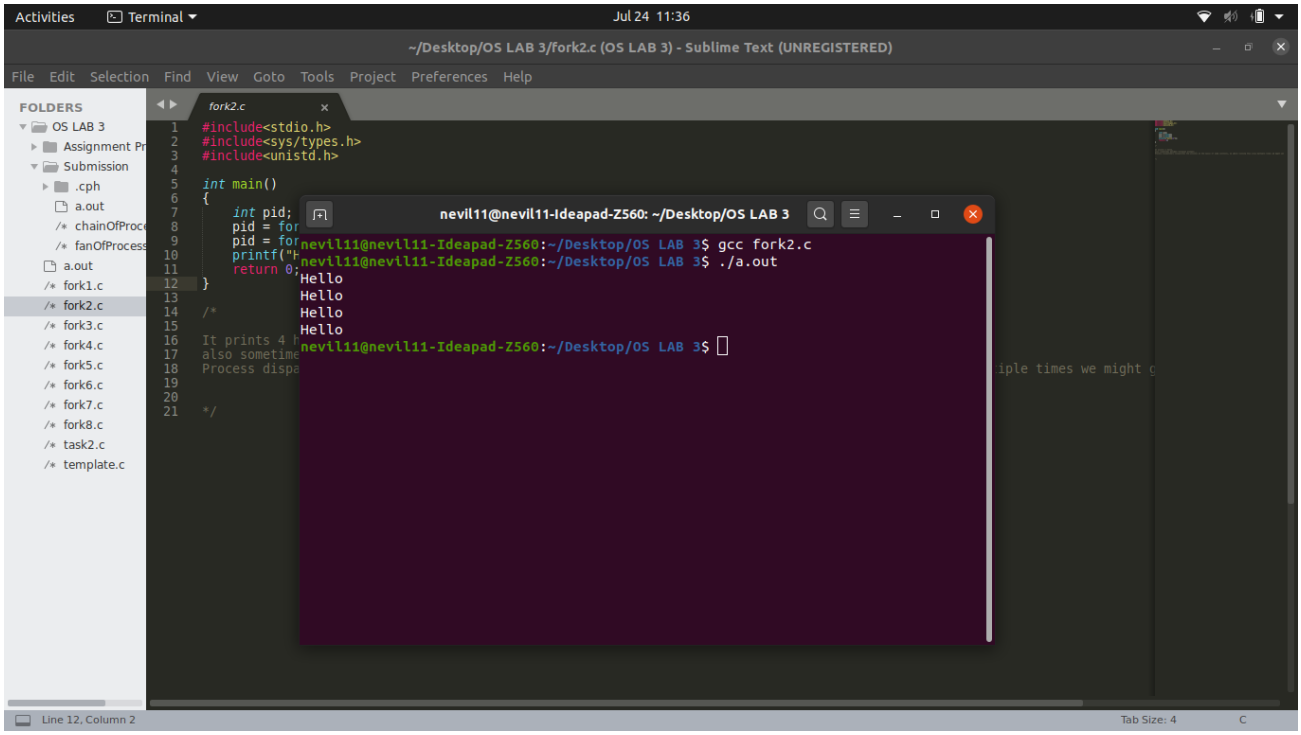


Subtask 2: **Calling fork twice**

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

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

/*
```

```
It prints 4 hello.
also sometimes between terminal prompts .
Process dispatchers dispatches the process on the basis of s
ome contexts, so while running this prog multiple times we m
ight get this kindof behaviour.


*/
```

Output:

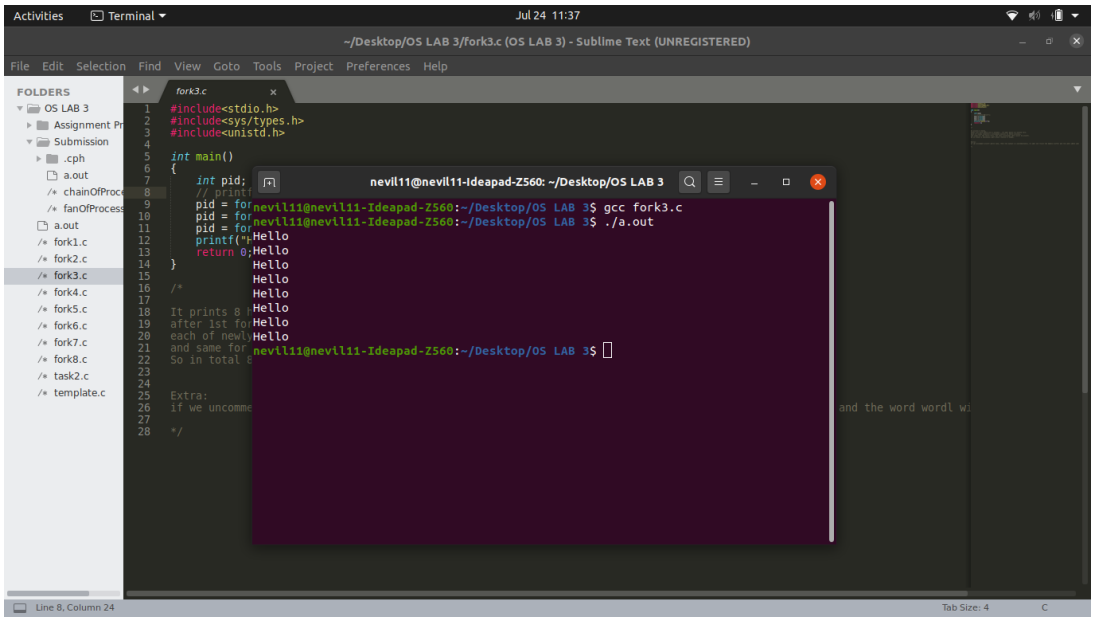


Subtask 3: **Calling fork thrice**

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

int main()
{
    int pid;
    // printf("World");
    pid = fork();
```

```
    pid = fork();
    pid = fork();
    printf("Hello\n");
    return 0;
}

/*

It prints 8 hello.
after 1st fork there are 2 process . IP will point to second
 fork
each of newly createrd 2 processes points to third process.
and same for the third fork, but this processes will point t
o printf.
So in total 8 processes will print :hello: 8 times.


Extra:
if we uncomment printf world line. Then the output is unpred
ictable. It will not flush the memory buffer and the word wo
rdl will stay there.

*/
```

Output:



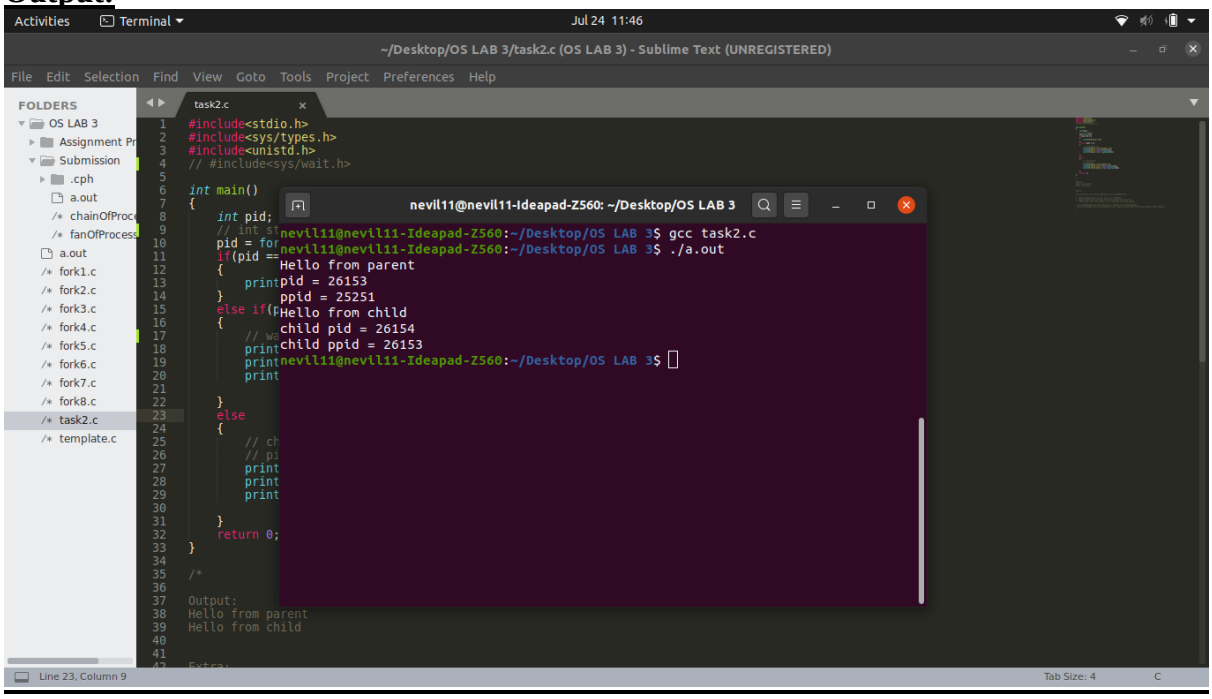
Task 2:

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

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

int main()
{
    int pid;
    // int status;
    pid = fork();
    if(pid == -1)
    {
        printf("Error!\n");
    }
    else if(pid > 0)
    {
        // wait(&status);
        printf("Hello from parent\n");
        printf("pid = %d\n",getpid());
        printf("ppid = %d\n",getppid());

    }
    else
    {
        // child part
        // pid = 0
        printf("Hello from child\n");
        printf("child pid = %d\n",getpid());
        printf("child ppid = %d\n",getppid());

    }
    return 0;
}
```

```
/*

Output:
Hello from parent
Hello from child


Extra:

can be useful to achieve separeate tasks independently

1. Inside parent block we can run ls in terminal
2. whose output will be given to child ( here child block )
    which will sort the output of ls, executed in parent blo
ck.

- By uncommenting the wait line we can achieve the following
 goal.
- we can terminate the child process first, then and only th
en the parent process will execute.

*/
```

Output:



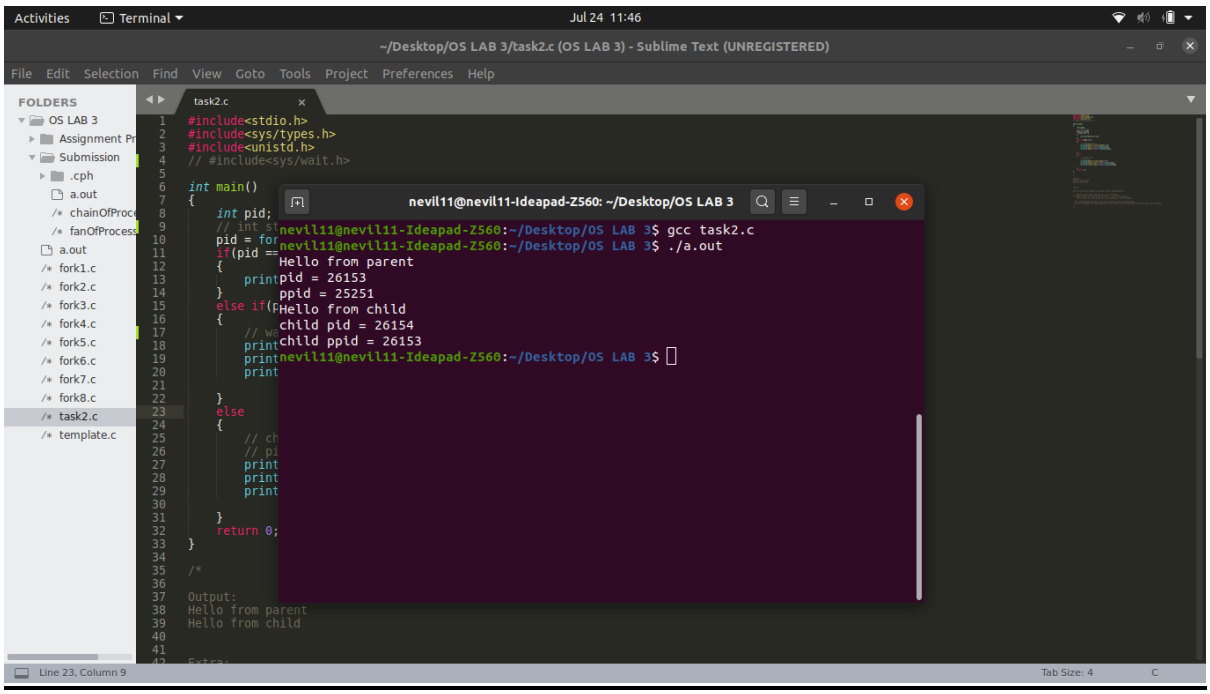
Task 3:

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

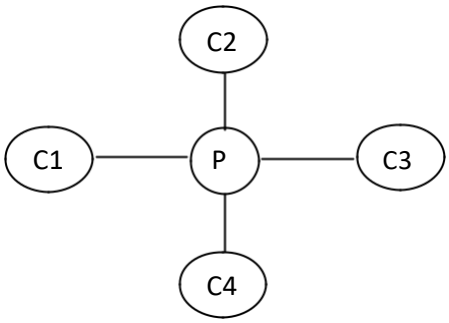
```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>


int main()
{
    int pid;
    int status;
    pid = fork();
    if(pid == -1)
    {
        printf("Error!\n");
    }
    else if(pid > 0)
    {
        wait(&status);
        printf("Hello from parent\n");
        printf("pid = %d\n",getpid());
        printf("ppid = %d\n",getppid());

    }
    else
    {
        // child part
        // pid = 0
        printf("Hello from child\n");
        printf("child pid = %d\n",getpid());
        printf("child ppid = %d\n",getppid());

    }
    return 0;
}

/*
```

```
Output:
Hello from child
Hello from parent


Exaplanation:

Here we are terminating the child process first, and then we
 will execute parent process.

*/
```

Output:

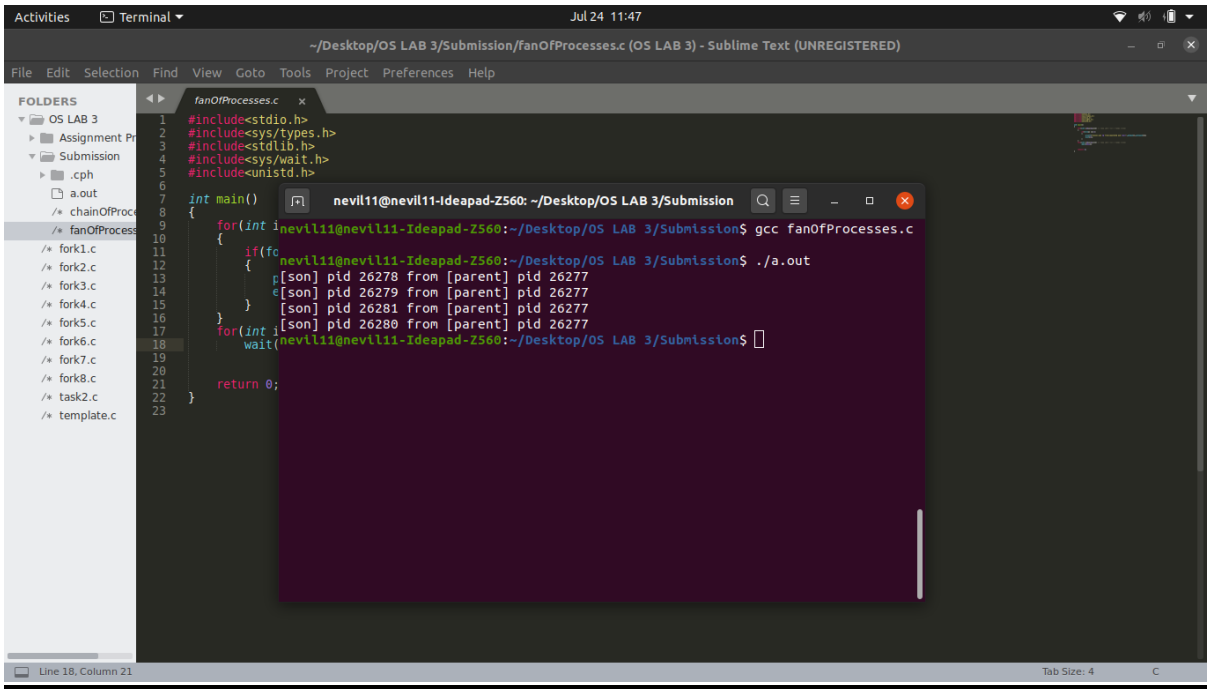


Task 4: Write a program to implement fan of n processes.

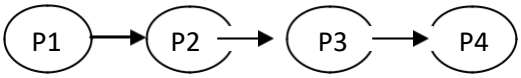



```c
#include<stdio.h>
#include<sys/types.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
    for(int i=0;i<4;i++) // loop will run n times (n=4)
    {
        if(fork() == 0)
        {
            printf("[son] pid %d from [parent] pid %d\n",getpid(),getppid());
            exit(0);
        }
    }
    for(int i=0;i<4;i++) // loop will run n times (n=4)
        wait(NULL);


    return 0;
}
```

Output:



Task 5: Write a program to implement chain of n processes.




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

int n=4;

int foo(const char *whoami) {
    printf("I am a %s.  My pid is:%d  my ppid is %d\n", whoa
mi, getpid(), getppid() );
    return 1;
}

int func(int n)
{
    if (n == 0)
```

```
    {
        return 0;
    }
    int pid = fork();
    if (pid == -1) {
        exit(0);
    }
    if (pid==0) {
        foo("child");
        n = n-1;
        func(n);
        exit(0);
    }
    else {
       wait(NULL);
    }
    return 0;
}

int main()
{
    func(n);
    return 0;
}
```

Output: