Process Creation and Execution

LAB # 05

Spring 2023

CSE-204L Operating Systems Lab

Submitted by: **Ali Asghar**

Registration No.: **21PWCSE2059**

Class Section: **C**

"On my honor, as student of University of Engineering and Technology, I have neither given nor received unauthorized assistance on this academic work."

Submitted to:

Engr. Madiha Sher

Date:

4th April 2023

Department of Computer Systems Engineering

University of Engineering and Technology, Peshawar

Objective:
This lab describes how a program can create, terminate, and control child processes. Actually, there are a few distinct operations involved: **creating a new child process**, and **coordinating the completion of the child process with the original program**.

****

WHAT IS A PROCESS? :

A **process** is basically a **single running program**. It may be a **``system**'' program (e.g login, update, csh) or **program initiated by the user** (pico, a.exe or a user written one).

When UNIX runs a process it gives each process a unique number - a **process ID**, **pid**.

The UNIX command **ps** will list all current processes running on your machine and will list the **pid**.

The C function **int getpid( )** will return the **pid** of process that called this function.

Processes are the primitive units for allocation of system resources. Each process has its own **address space** and (usually) one thread of control. **A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.**

Processes are organized hierarchically. Each process has a **parent** process which explicitly arranged to create it. The processes created by a given parent are called its **child** processes.

A child inherits many of its attributes from the parent process.

Every process in a UNIX system has the following attributes:

- **some code**
- **some data**
- **a stack**
- **a unique process id number (PID)**

When UNIX is first started, there's only one visible process in the system. This process is called "**init**", and its **PID** is **1**. The only way to create a new process in UNIX is to duplicate an existing process, so "**init**" is the ancestor of all subsequent processes. When a process duplicates, the parent and child processes are identical in every way except their **PIDs**; the child's code, data, and stack are a copy of the parent's, and they even continue to execute the same code. **A child process may, however, replace its code with that of another executable file, thereby differentiating itself from its parent**. For example, when "**init**" starts executing, it quickly duplicates several times. Each of the duplicate child processes then replaces its code from the executable file called "**getty**" which is responsible for handling user logins.

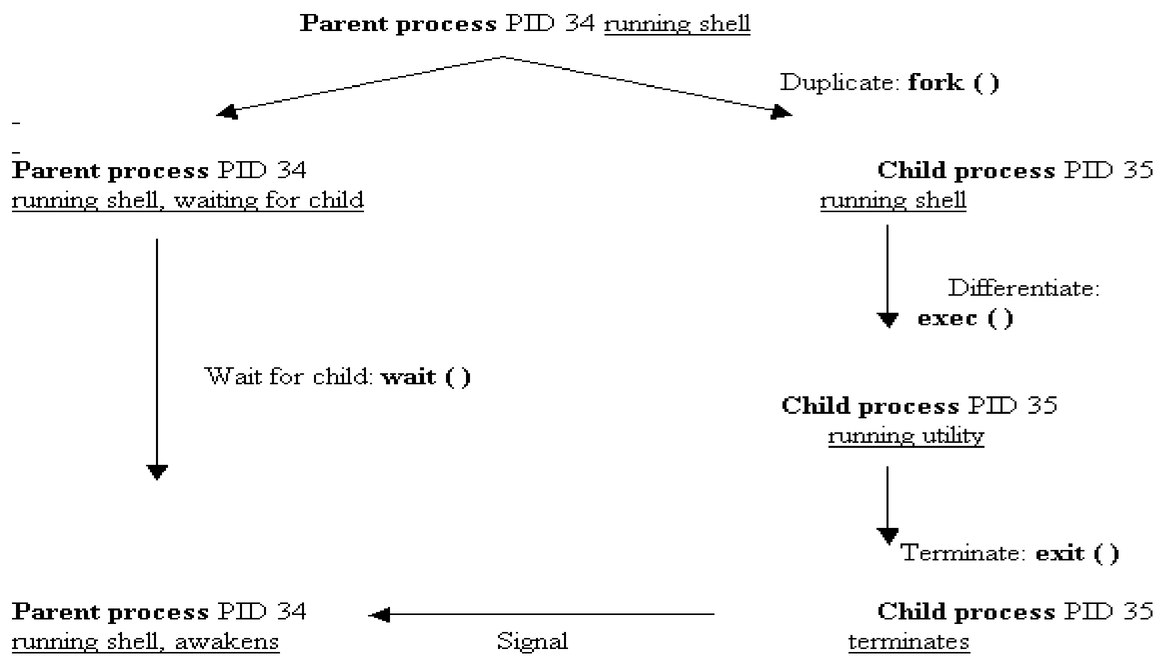
When a child process terminates, its death is communicated to its parent so that the parent may take some appropriate action.

A process that is waiting for its parent to accept its return code is called a zombie process.

If a parent dies before its child, the child (orphan process) is automatically adopted by the original "init" process whose PID is 1.

Its very common for a parent process to suspend until one of its children terminates. For example, when a shell executes a utility in the foreground, it duplicates into two shell processes; the child shell process replaces its code with that of utility, whereas the parent shell waits for the child process to terminate. When the child process terminates, the original parent process awakens and presents the user with the next shell prompt.

Here's an illustration of the way that a shell executes a utility:


Parent process PID 34 running shell
Duplicate: **fork ( )**
Parent process PID 34 running shell, waiting for child
Child process PID 35 running shell
Differentiate: **exec ( )**
Wait for child: **wait ( )**
Child process PID 35 running utility
Terminate: **exit ( )**
Parent process PID 34 running shell, awakens
Signal
Child process PID 35 terminates

A program usually runs as a single process. However later we will see how we can make programs run as several separate communicating processes.

Running UNIX commands from C :

We can run commands from a C program just as if they were from the UNIX command line by using the **system( )** function.

int system (char *string) -- where **string** can be the name of a UNIX utility, an executable shell script or a user program. System returns the exit status

of the shell. System is prototyped in **<stdlib.h>**

Example: Call **ls** from a program

File Lab5_0.c :

```
main( )
{    printf(``Files in Directory are:n");
     system(``ls -l");
}
```

system is a call that is made up of 3 other system calls: **execl( ), wait( )** and **fork( )** (which are prototyped in <unistd.h>)

PROCESS CREATION CONCEPTS :

This section gives an overview of processes and of the steps involved in creating a process and making it run another program.

Each process is named by a process ID number. A unique process ID is allocated to each process when it is created. The **lifetime of a process ends when its termination is reported to its parent process**; at that time, all of the process resources, including its process ID, are freed.

Processes are created with the fork system call (so the operation of creating a new process is sometimes called forking a process). **The child process created by fork is a copy of the original parent process, except that it has its own process ID**.

After forking a child process, both the parent and child processes continue to execute normally.

If you want your program to wait for a child process to finish executing before continuing, you must do this explicitly after the fork operation, by calling **wait**. This function gives you limited information about why the child terminated--for example, its exit status code.

A newly forked child process continues to execute the same program as its parent process, at the point where the fork call returns. You can use the return value from fork to tell whether the program is running in the parent process or the child.

Having several processes run the same program is only occasionally useful. **But the child can execute another program using one of the exec functions.** The program that the process is executing is called its process image. **Starting execution of a new program causes the process to forget all about its previous process image; when the new program exits, the process exits too, instead of returning to the previous process image.**

____

____

Process Identification :

The **pid_t** data type represents process IDs. You can get the process ID of a process by calling **getpid**. The function **getppid** returns the process ID of the parent of the current process (this is also known as the parent process ID). Your program should include the header files `**unistd.h**' and `**sys/types.h**' to use these functions.

Data Type: **pid_t**

The **pid_t** data type is a signed integer type which is capable of representing a process ID. In the GNU library, this is an **int**.

Function: **pid_t getpid (void)**

The **getpid** function returns the **process ID** of the current process.

Function: **pid_t getppid (void)**

The **getppid** function returns the **process ID of the parent** of the current process.

Creating Multiple Processes :

A special type of process important in the Unix environment is the **daemon**.

The **fork** function is the primitive for creating a process. It is declared in the header file `**unistd.h**'.

Function: **pid_t fork (void)**

The **fork** function **creates a new process**.

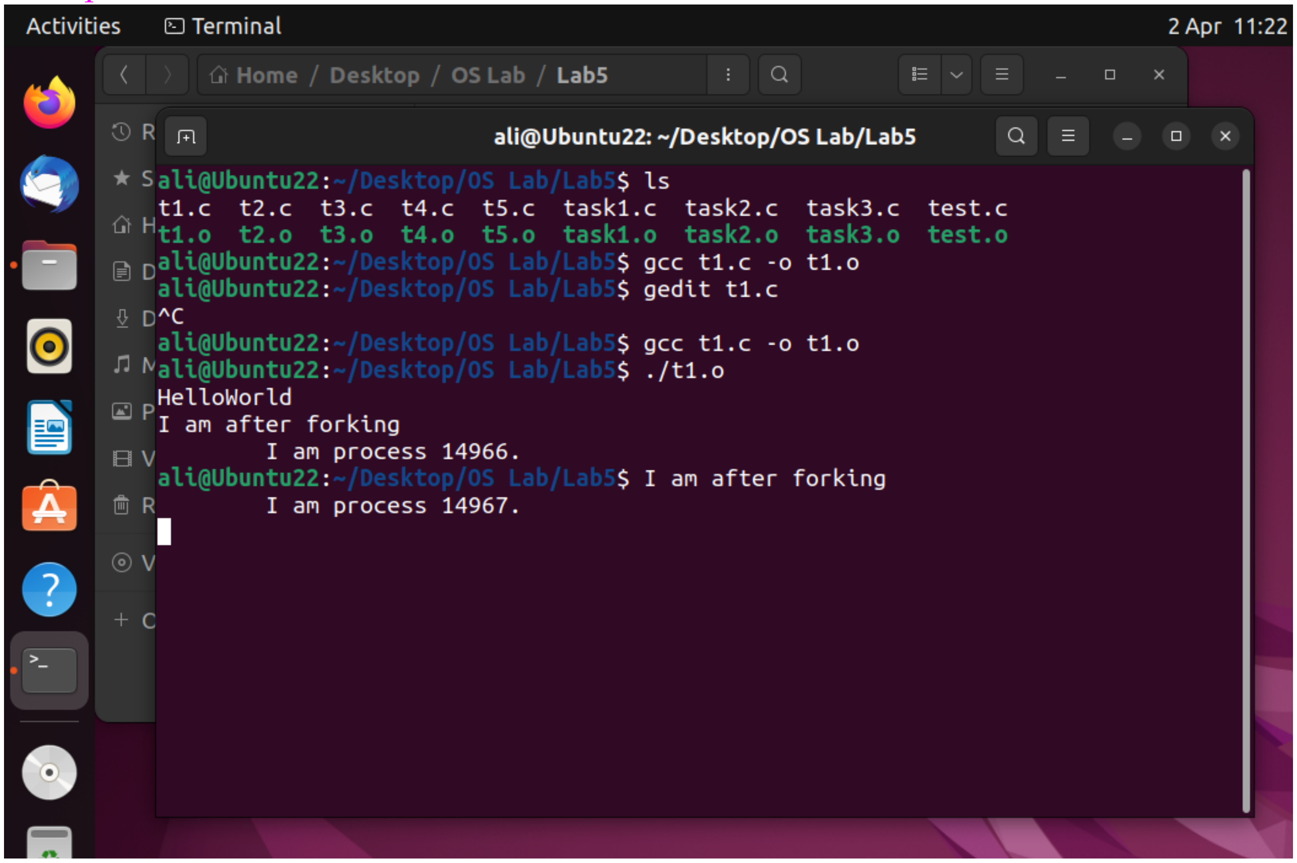
If the operation is **successful**, there are then both **parent** and **child** processes and both see **fork** return, but with different values: it returns a value of **0** in the **child** process and returns the **child's process ID** in the **parent** process.

If process creation **failed**, fork returns a value of **-1** in the **parent** process and **no child is created**.

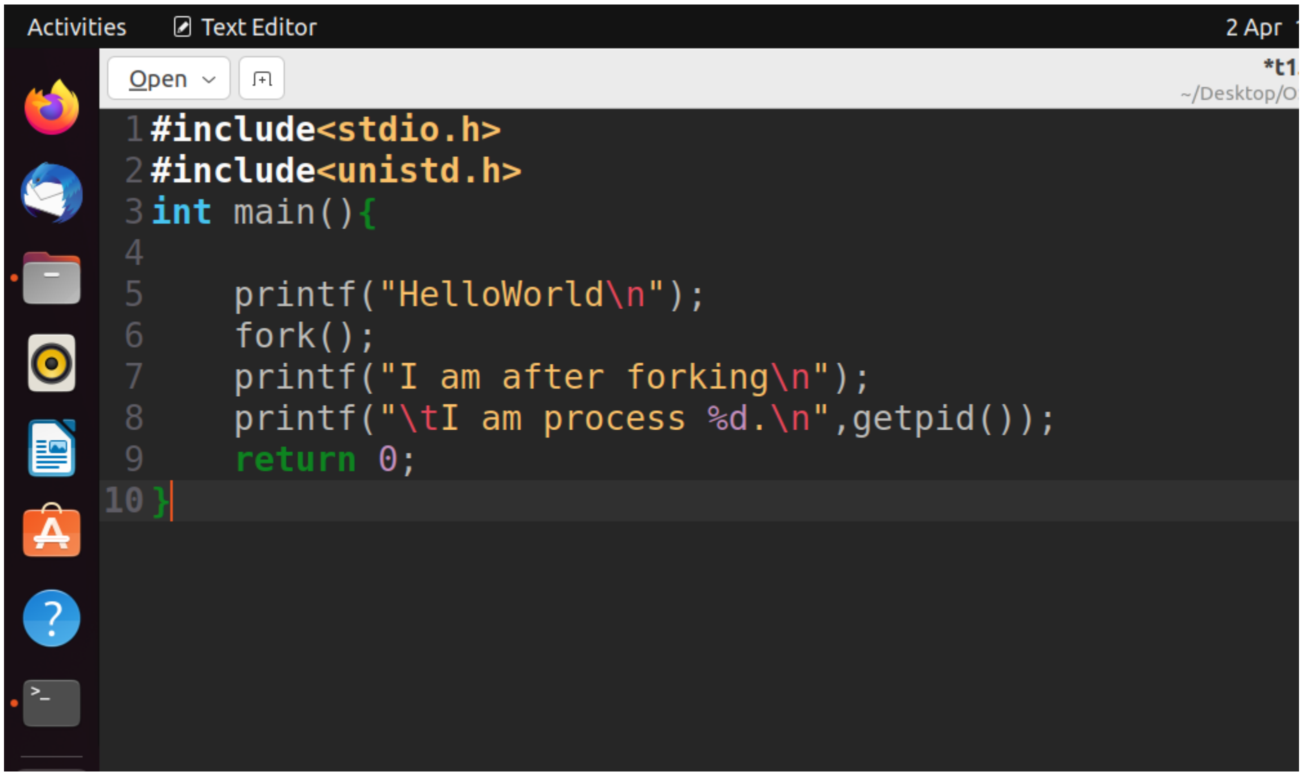
The specific attributes of the child process that differ from the parent process are:

The child process has its own unique process ID.
The parent process ID of the child process is the process ID of its parent process. The child process gets its own copies of the parent process's open file descriptors.
Subsequently changing attributes of the file descriptors in the parent process won't affect the file descriptors in the child, and vice versa. However, the file position associated with each descriptor is shared by both processes. The elapsed processor times for the child process are set to zero.
The child doesn't inherit file locks set by the parent process.
The child doesn't inherit alarms set by the parent process.
The set of pending signals for the child process is cleared.

Example Lab5_1.c :


Activities
Terminal
2 Apr 11:22
Home / Desktop / OS Lab / Lab5
ali@Ubuntu22: ~/Desktop/OS Lab/Lab5

```
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ls
t1.c  t2.c  t3.c  t4.c  t5.c  task1.c  task2.c  task3.c  test.c
t1.o  t2.o  t3.o  t4.o  t5.o  task1.o  task2.o  task3.o  test.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gcc t1.c -o t1.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gedit t1.c
^C
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gcc t1.c -o t1.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ./t1.o
HelloWorld
I am after forking
        I am process 14966.
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ I am after forking
        I am process 14967.
```

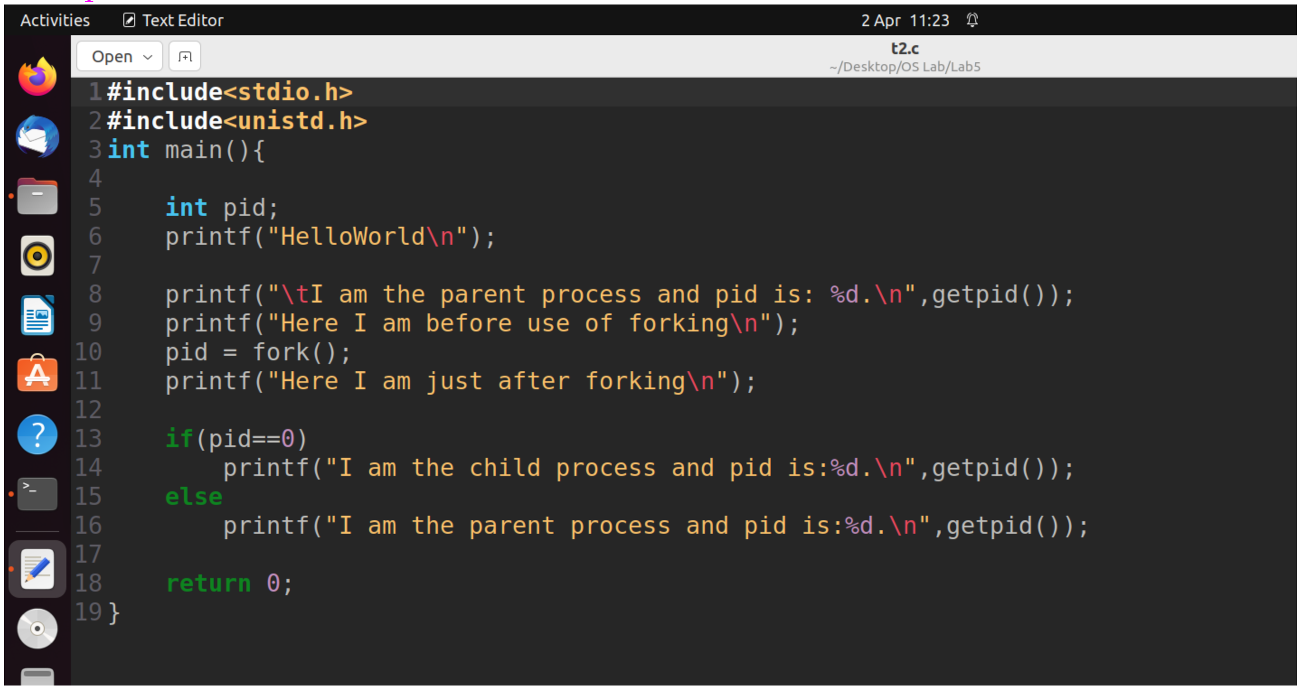


```c
#include<stdio.h>
#include<unistd.h>
int main(){

    printf("HelloWorld\n");
    fork();
    printf("I am after forking\n");
    printf("\tI am process %d.\n",getpid());
    return 0;
}
```

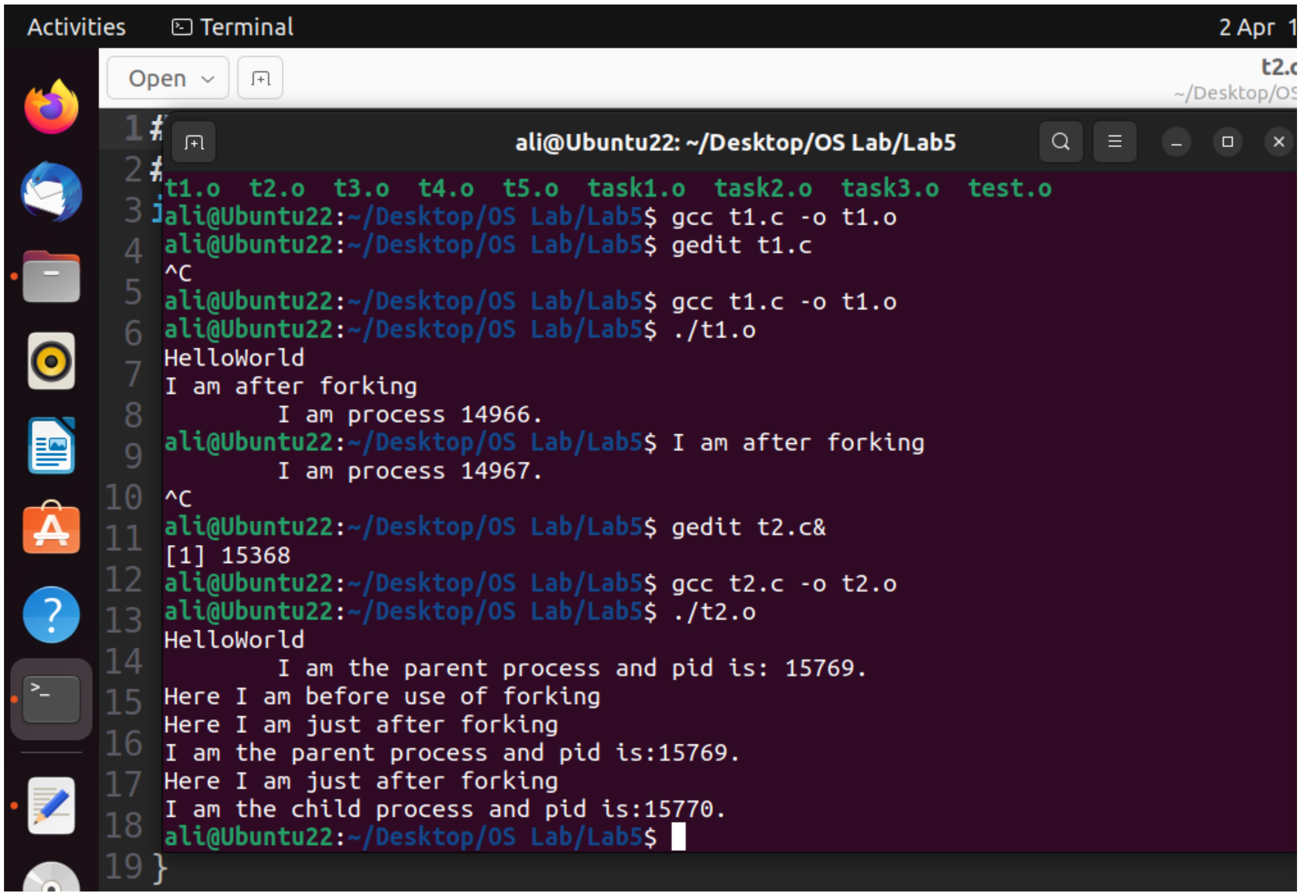
When this program is executed, it first prints Hello World! . When the fork is executed, an identical process called the child is created. Then both the parent and the child process begin execution at the next statement. Note the following:

When a fork is executed, everything in the parent process is copied to the child process. This includes variable values, code, and file descriptors.
Following the fork, the child and parent processes are completely independent.
There is no guarantee which process will print I am a process first.
The child process begins execution at the statement immediately after the fork, not at the beginning of the program.
A parent process can be distinguished from the child process by examining the return value of the fork call. Fork returns a zero to the child process and the process id of the child process to the parent.
A process can execute as many forks as desired. However, be wary of infinite loops of forks (there is a maximum number of processes allowed for a single user).

Example Lab5_2.c :

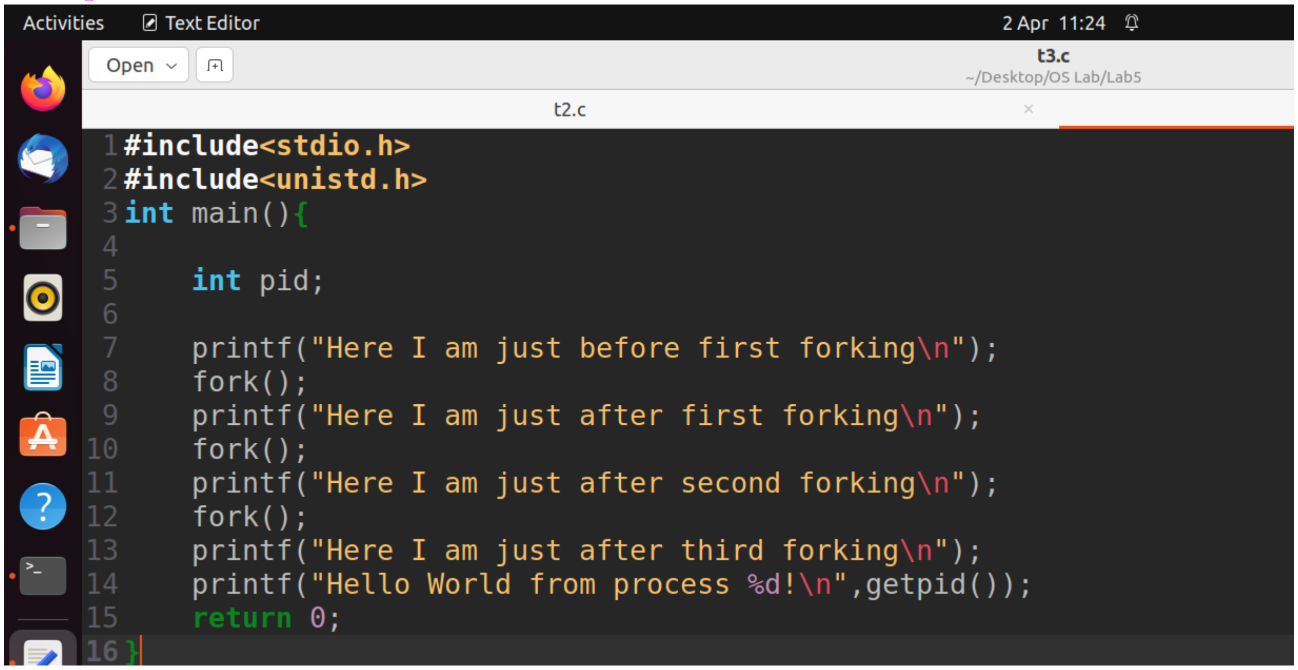


```c
#include<stdio.h>
#include<unistd.h>
int main(){

    int pid;
    printf("HelloWorld\n");

    printf("\tI am the parent process and pid is: %d.\n",getpid());
    printf("Here I am before use of forking\n");
    pid = fork();
    printf("Here I am just after forking\n");

    if(pid==0)
        printf("I am the child process and pid is:%d.\n",getpid());
    else
        printf("I am the parent process and pid is:%d.\n",getpid());

    return 0;
}
```

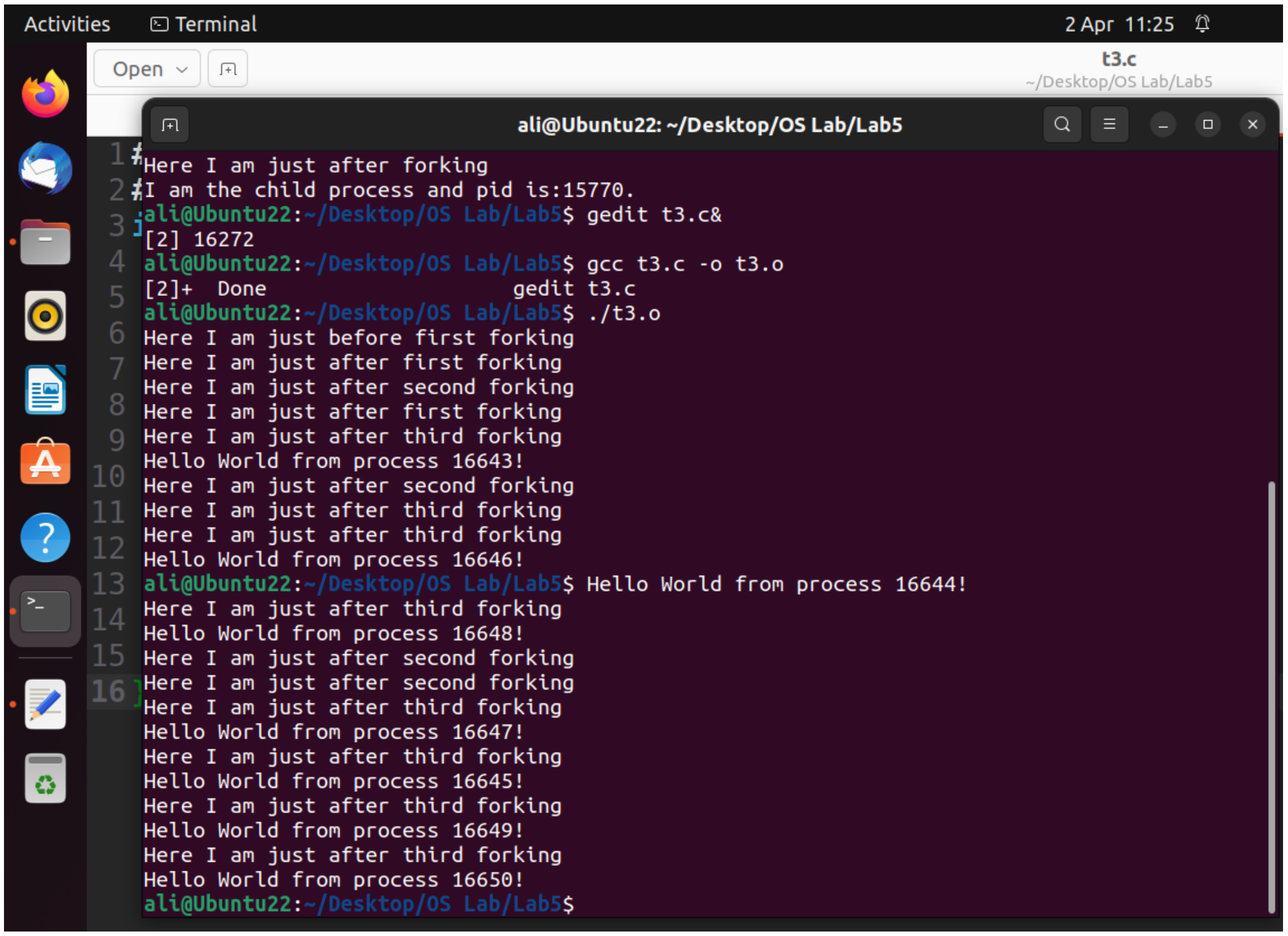


```
t1.o  t2.o  t3.o  t4.o  t5.o  task1.o  task2.o  task3.o  test.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gcc t1.c -o t1.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gedit t1.c
^C
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gcc t1.c -o t1.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ./t1.o
HelloWorld
I am after forking
        I am process 14966.
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ I am after forking
        I am process 14967.
^C
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gedit t2.c&
[1] 15368
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gcc t2.c -o t2.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ./t2.o
HelloWorld
        I am the parent process and pid is: 15769.
Here I am before use of forking
Here I am just after forking
I am the parent process and pid is:15769.
Here I am just after forking
I am the child process and pid is:15770.
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$
```

Example Lab5_3.c :



```c
#include<stdio.h>
#include<unistd.h>
int main(){

    int pid;

    printf("Here I am just before first forking\n");
    fork();
    printf("Here I am just after first forking\n");
    fork();
    printf("Here I am just after second forking\n");
    fork();
    printf("Here I am just after third forking\n");
    printf("Hello World from process %d!\n",getpid());
    return 0;
}
```



```
Here I am just after forking
I am the child process and pid is:15770.
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gedit t3.c&
[2] 16272
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gcc t3.c -o t3.o
[2]+  Done                    gedit t3.c
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ./t3.o
Here I am just before first forking
Here I am just after first forking
Here I am just after second forking
Here I am just after first forking
Here I am just after third forking
Hello World from process 16643!
Here I am just after second forking
Here I am just after third forking
Here I am just after third forking
Hello World from process 16646!
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ Hello World from process 16644!
Here I am just after third forking
Hello World from process 16648!
Here I am just after second forking
Here I am just after second forking
Here I am just after third forking
Hello World from process 16647!
Here I am just after third forking
Hello World from process 16645!
Here I am just after third forking
Hello World from process 16649!
Here I am just after third forking
Hello World from process 16650!
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$
```

Function: void exit (int status)

exit ( ) terminates the process which calls this function and returns the exit **status** value. Both UNIX and C (forked) programs can read the status value.

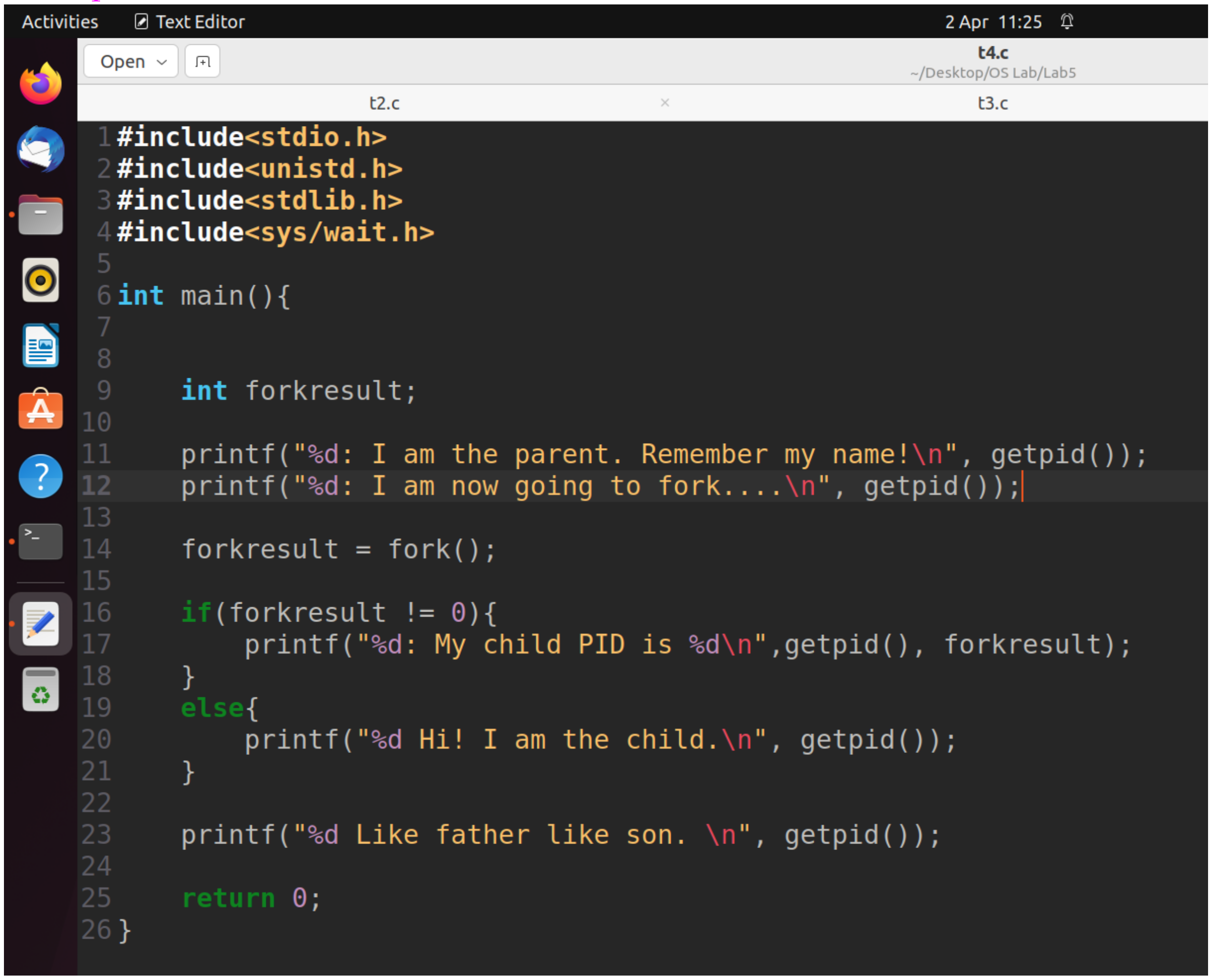
By convention, **a status of 0 means normal termination any other value indicates an error or unusual occurrence.** Many standard library calls have errors defined in the sys/stat.h header file. We can easily derive our own conventions.

sleep

A process may **suspend** for a period of time using the **sleep** command

Function: unsigned int sleep (seconds)

Example Lab5_4.c :

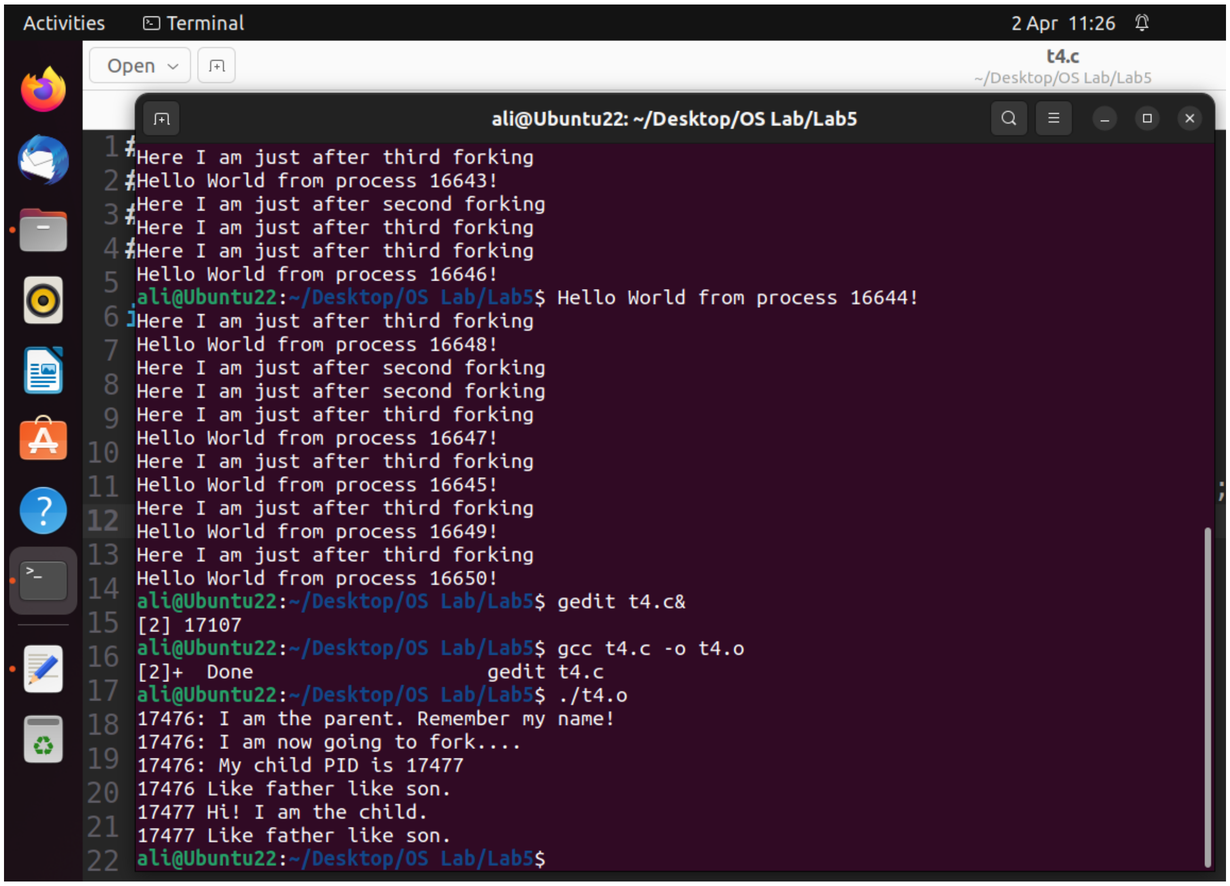


Activities Text Editor 2 Apr 11:25

Open t4.c ~/Desktop/OS Lab/Lab5

t2.c t3.c

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

int main(){


    int forkresult;

    printf("%d: I am the parent. Remember my name!\n", getpid());
    printf("%d: I am now going to fork....\n", getpid());

    forkresult = fork();

    if(forkresult != 0){
        printf("%d: My child PID is %d\n",getpid(), forkresult);
    }
    else{
        printf("%d Hi! I am the child.\n", getpid());
    }

    printf("%d Like father like son. \n", getpid());

    return 0;
}
```



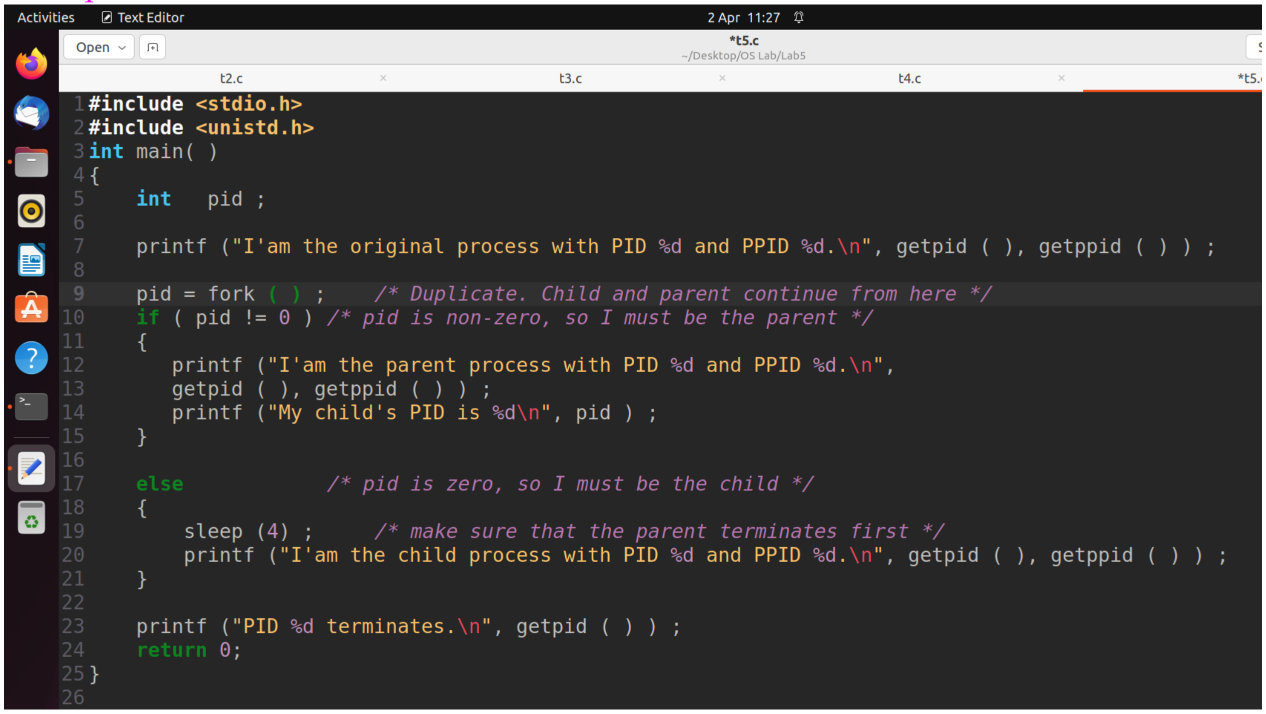
```
Here I am just after third forking
Hello World from process 16643!
Here I am just after second forking
Here I am just after third forking
Here I am just after third forking
Hello World from process 16646!
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ Hello World from process 16644!
Here I am just after third forking
Hello World from process 16648!
Here I am just after second forking
Here I am just after second forking
Here I am just after third forking
Hello World from process 16647!
Here I am just after third forking
Hello World from process 16645!
Here I am just after third forking
Hello World from process 16649!
Here I am just after third forking
Hello World from process 16650!
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gedit t4.c&
[2] 17107
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gcc t4.c -o t4.o
[2]+  Done                    gedit t4.c
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ./t4.o
17476: I am the parent. Remember my name!
17476: I am now going to fork....
17476: My child PID is 17477
17476 Like father like son.
17477 Hi! I am the child.
17477 Like father like son.
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$
```

____

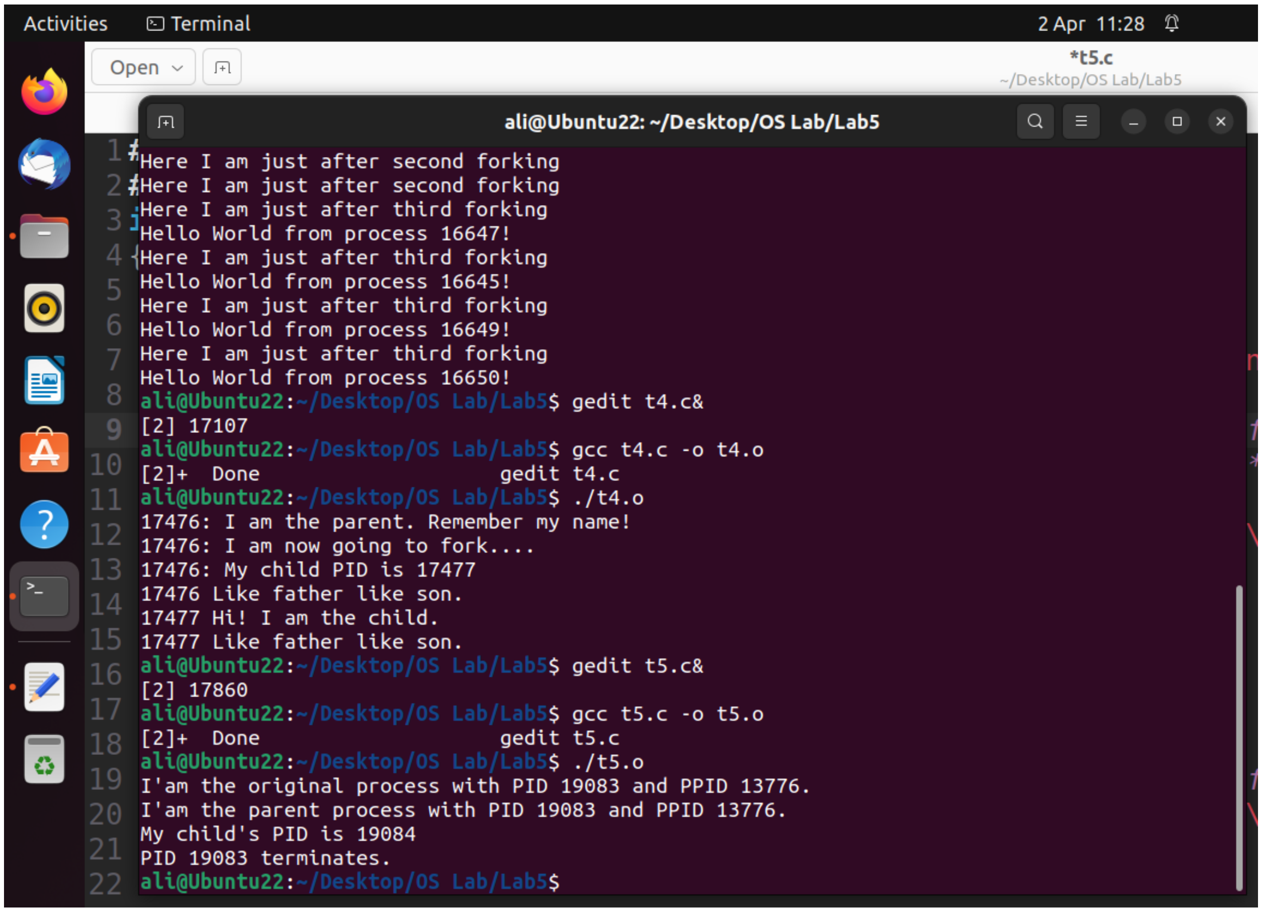
Orphan processes :

When a **parent dies before its child**, the child is automatically adopted by the original "**init**" process whose **PID** is **1**. To, illustrate this insert a **sleep** statement into the child's code. This ensured that the parent process terminated before its child.

Example Lab5_5.c :



```c
#include <stdio.h>
#include <unistd.h>
int main( )
{
    int   pid ;

    printf ("I'am the original process with PID %d and PPID %d.\n", getpid ( ), getppid ( ) ) ;

    pid = fork ( ) ;    /* Duplicate. Child and parent continue from here */
    if ( pid != 0 ) /* pid is non-zero, so I must be the parent */
    {
       printf ("I'am the parent process with PID %d and PPID %d.\n",
       getpid ( ), getppid ( ) ) ;
       printf ("My child's PID is %d\n", pid ) ;
    }

    else             /* pid is zero, so I must be the child */
    {
        sleep (4) ;     /* make sure that the parent terminates first */
        printf ("I'am the child process with PID %d and PPID %d.\n", getpid ( ), getppid ( ) ) ;
    }

    printf ("PID %d terminates.\n", getpid ( ) ) ;
    return 0;
}
```



```
Here I am just after second forking
Here I am just after second forking
Here I am just after third forking
Hello World from process 16647!
Here I am just after third forking
Hello World from process 16645!
Here I am just after third forking
Hello World from process 16649!
Here I am just after third forking
Hello World from process 16650!
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gedit t4.c&
[2] 17107
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gcc t4.c -o t4.o
[2]+  Done                    gedit t4.c
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ./t4.o
17476: I am the parent. Remember my name!
17476: I am now going to fork....
17476: My child PID is 17477
17476 Like father like son.
17477 Hi! I am the child.
17477 Like father like son.
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gedit t5.c&
[2] 17860
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gcc t5.c -o t5.o
[2]+  Done                    gedit t5.c
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ./t5.o
I'am the original process with PID 19083 and PPID 13776.
I'am the parent process with PID 19083 and PPID 13776.
My child's PID is 19084
PID 19083 terminates.
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$
```

Important points to note:

1. **The Shell acts as the parent process**. All the processes started by the user are treated as the children of **shell**.

2. The **status of a UNIX process** is shown as the **second column** of the process table when viewed by the execution of the **ps** command. Some of the states are:

R: ***running***, **O**: ***orphan***, **S**: ***sleeping***, **Z**: ***zombie***.

3. **The child process is given the time slice before the parent process**. This is quite logical. For example, we do not want the process started by us to wait until its parent, which is the UNIX shell finishes. This will explain the order in which the print statement is executed by the parent and the children.

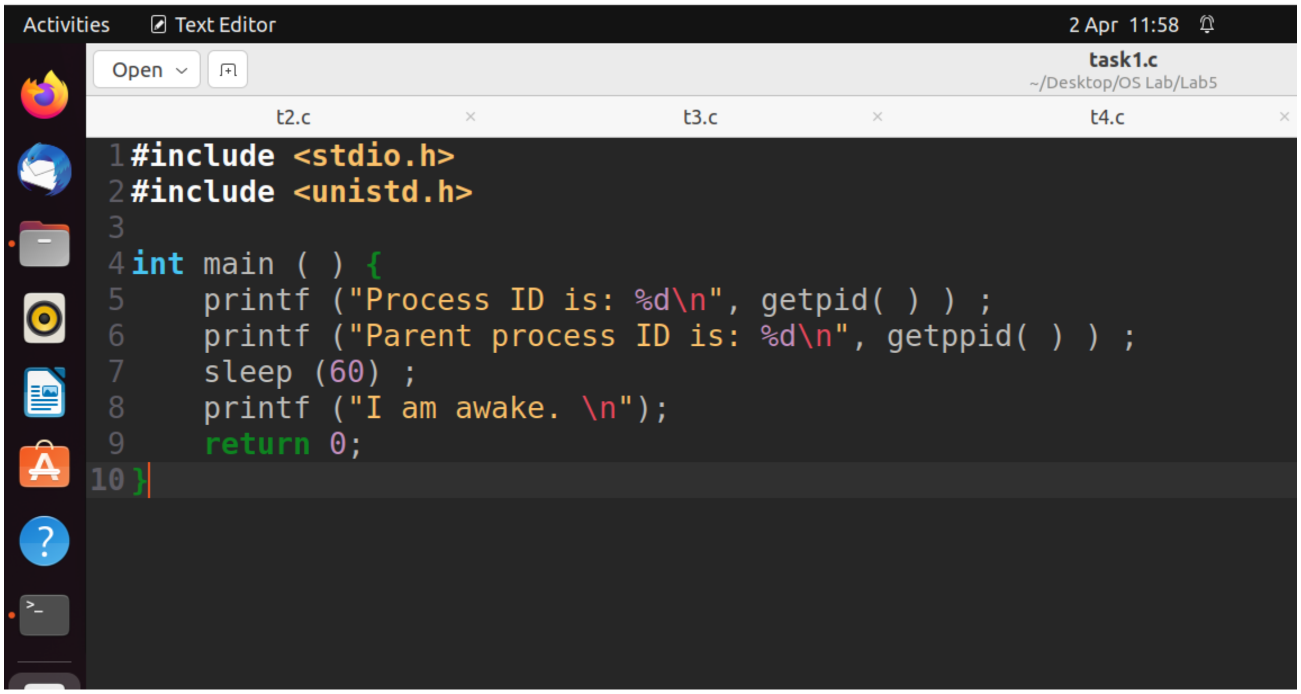
TASKS:

Execute the C programs given in the following problems. ***Observe*** and **Interpret** the results. You will learn about ***child*** and ***parent*** processes, and much more about UNIX processes in general by performing the suggested experiments. UNIX Calls used in the following problems:

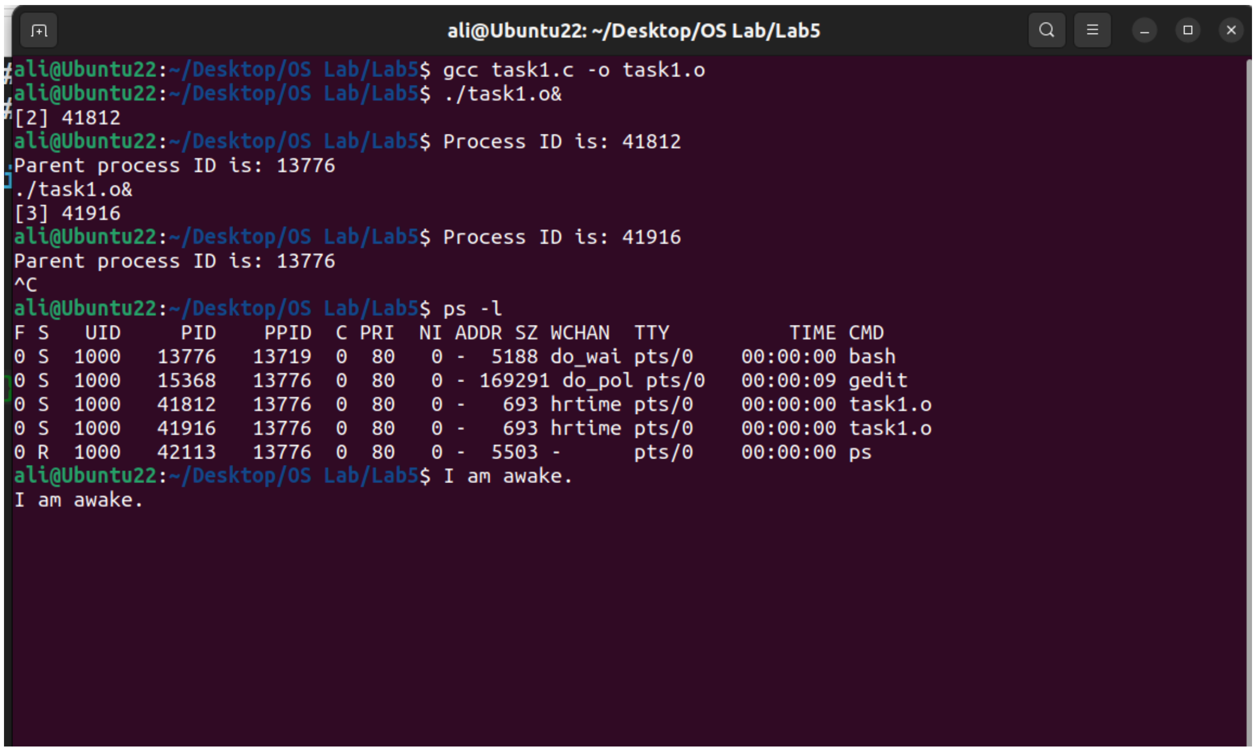
getpid( ), getppid( ), sleep( ) and ***fork( )***.

1) Run the following program twice. Both times as a background process, i.e., suffix it with an ampersand "**&**". Once both processes are running as background processes, view the *process table* using **ps -l** UNIX command. Observe the *process state*, *PID (process ID)* etc. Repeat this experiment to observe the changes, if any. Write your observation about the Process ID and state of the process.

```
main ( ) {
   printf ("Process ID is: %d\n",
getpid( ) ) ;
   printf ("Parent process ID is:
%d\n", getppid( ) ) ;
   sleep (60) ;
   printf ("I am awake. \n");
}
```



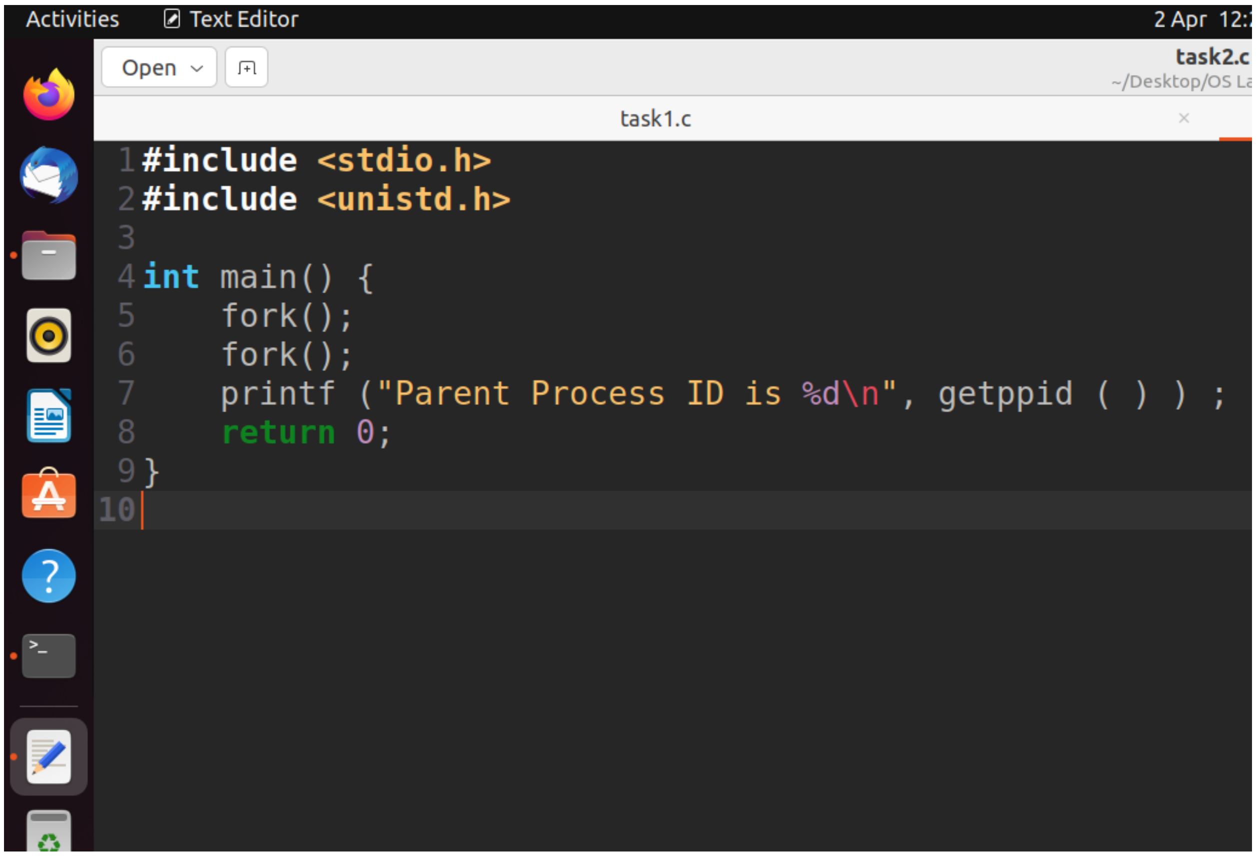
```
#include <stdio.h>
#include <unistd.h>

int main ( ) {
    printf ("Process ID is: %d\n", getpid( ) ) ;
    printf ("Parent process ID is: %d\n", getppid( ) ) ;
    sleep (60) ;
    printf ("I am awake. \n");
    return 0;
}
```



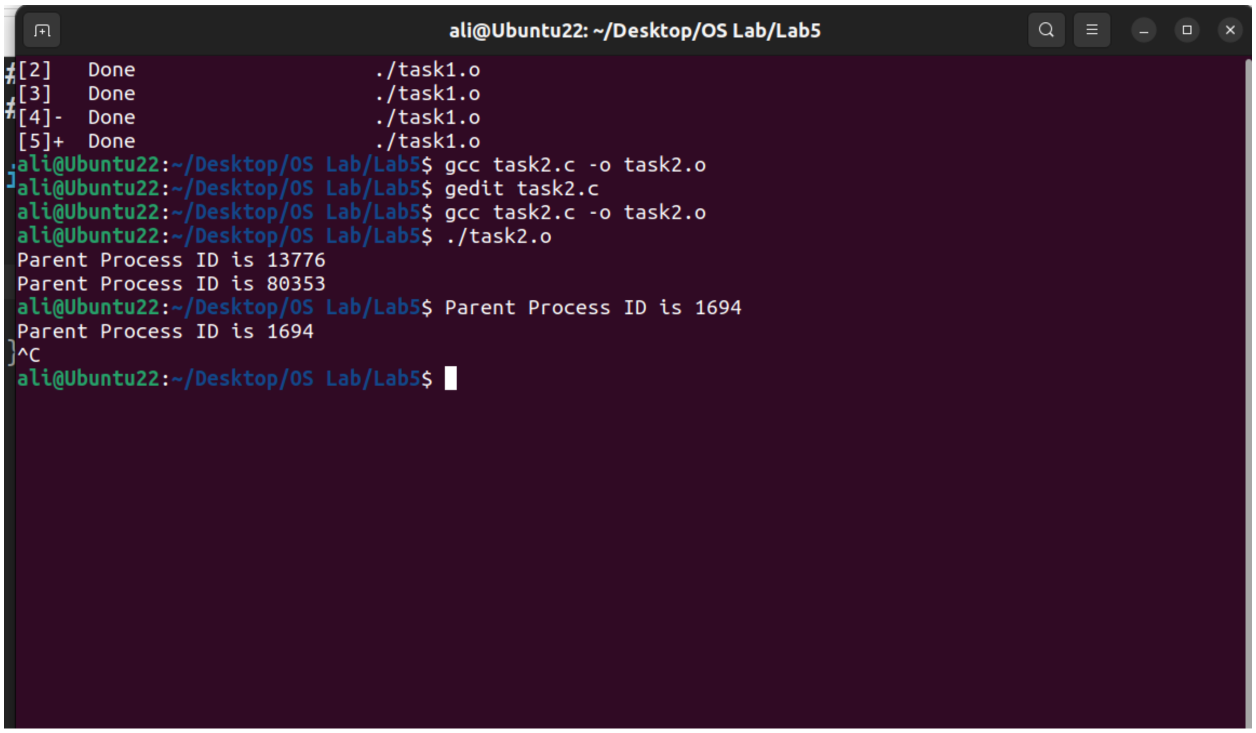
```
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gcc task1.c -o task1.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ./task1.o&
[2] 41812
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ Process ID is: 41812
Parent process ID is: 13776
./task1.o&
[3] 41916
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ Process ID is: 41916
Parent process ID is: 13776
^C
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   13776   13719  0  80   0 -  5188 do_wai pts/0    00:00:00 bash
0 S  1000   15368   13776  0  80   0 - 169291 do_pol pts/0   00:00:09 gedit
0 S  1000   41812   13776  0  80   0 -   693 hrtime pts/0    00:00:00 task1.o
0 S  1000   41916   13776  0  80   0 -   693 hrtime pts/0    00:00:00 task1.o
0 R  1000   42113   13776  0  80   0 -  5503 -      pts/0    00:00:00 ps
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ I am awake.
I am awake.
```

2) Run the following program and observe the *number of times* and the *order* in which the print statement is executed. The **fork( )** creates a child that is a duplicate of the parent process. The child process begins from the **fork( )**. All the statements after the call to **fork ( )** are executed by the parent process and also by the child process. Draw a family tree of processes and explain the results you observed.

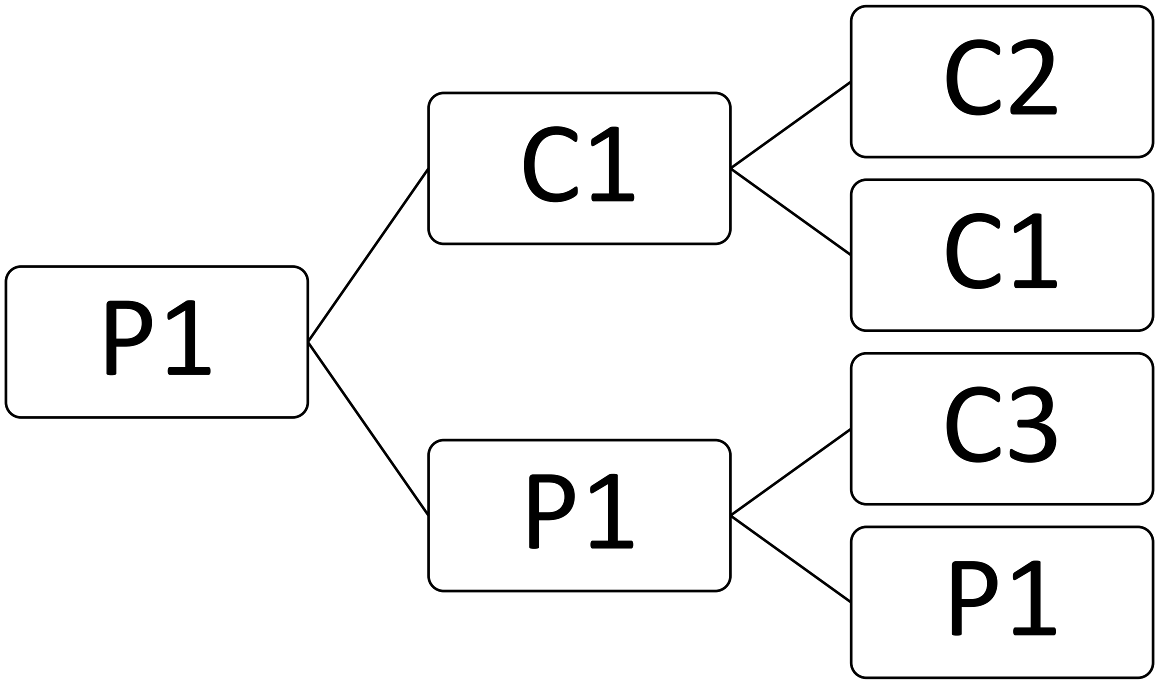
```
main ( ) {
   fork ( ) ;
   fork ( ) ;
   printf ("Parent Process ID is
%d\n", getppid ( ) ) ;
}
```



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

int main() {
    fork();
    fork();
    printf ("Parent Process ID is %d\n", getppid ( ) ) ;
    return 0;
}
```



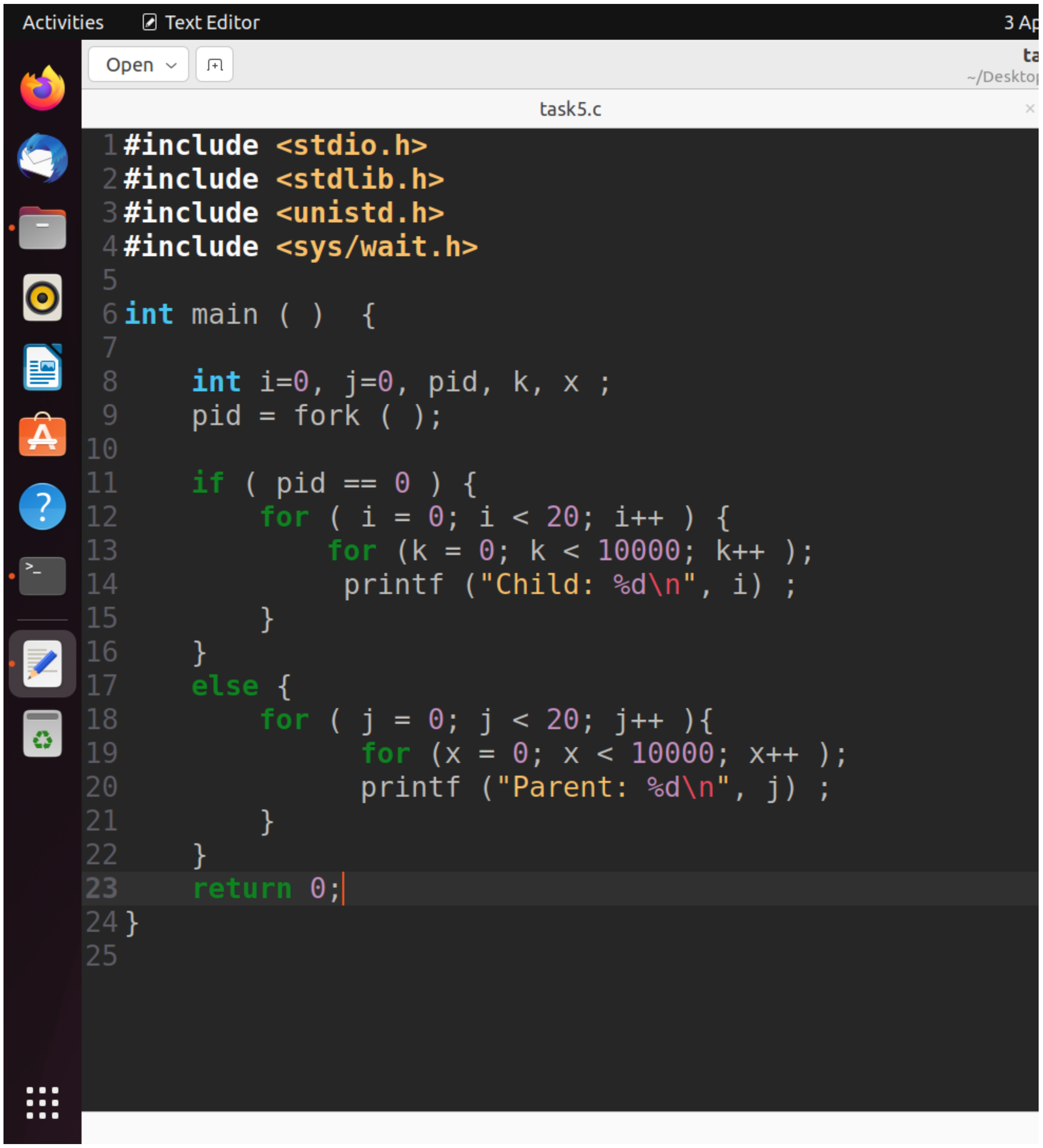
```
[2]   Done                    ./task1.o
[3]   Done                    ./task1.o
[4]-  Done                    ./task1.o
[5]+  Done                    ./task1.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gcc task2.c -o task2.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gedit task2.c
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gcc task2.c -o task2.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ./task2.o
Parent Process ID is 13776
Parent Process ID is 80353
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ Parent Process ID is 1694
Parent Process ID is 1694
^C
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$
```



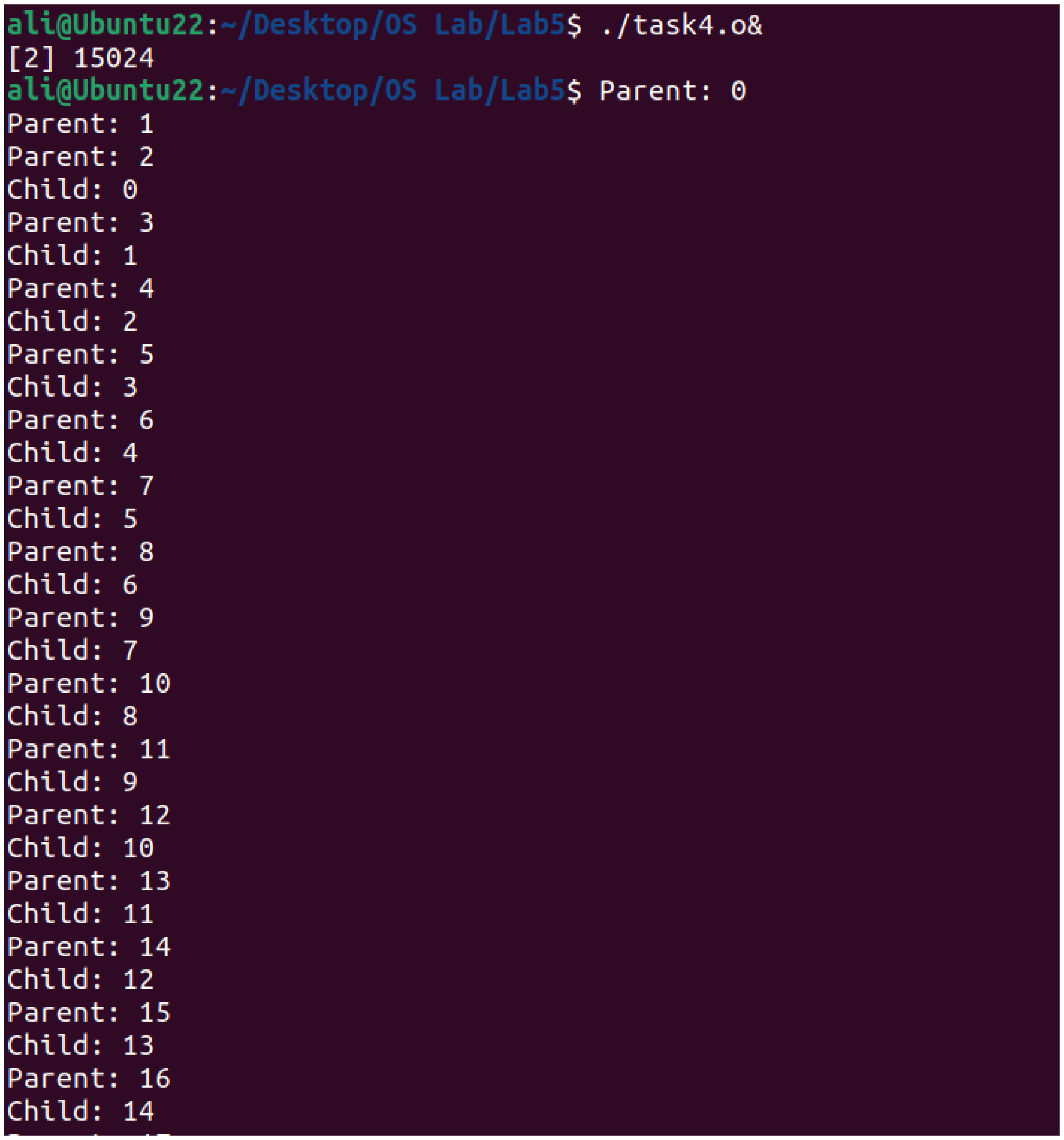
3) Run the following program and observe the result of **time slicing** used by UNIX.

```
main ( )  {
   int i=0, j=0, pid, k, x ;
   pid = fork ( );
   if ( pid == 0 ) {
      for ( i = 0; i < 20; i++ ) {
         for (k = 0; k < 10000; k++ );
          printf ("Child: %d\n", i) ;
      }
   }
   else {
      for ( j = 0; j < 20; j++ ){
           for (x = 0; x < 10000; x++ );
```

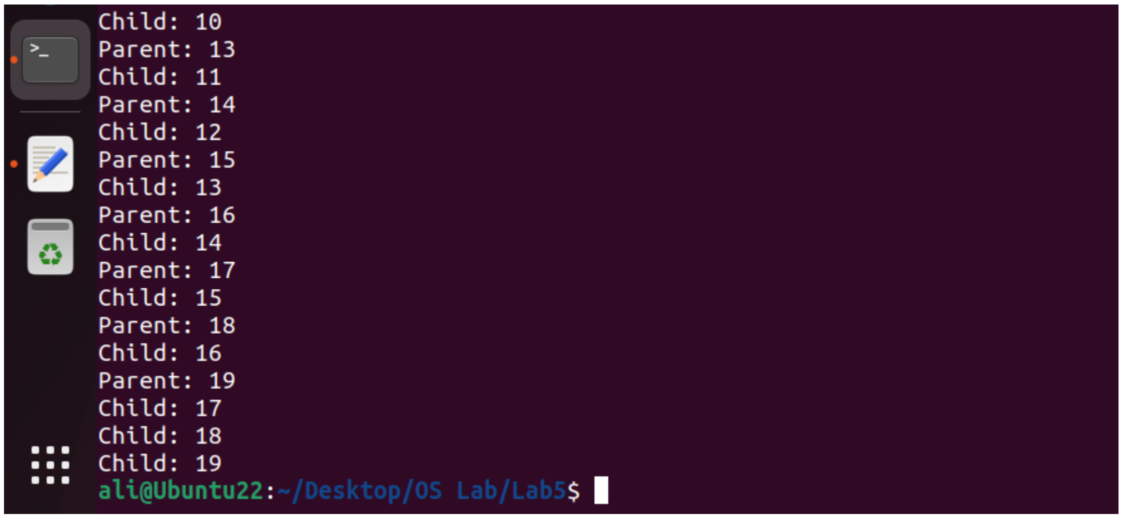
```
            printf ("Parent: %d\n", j) ;
        }
    }
}
```



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

int main ( )  {

    int i=0, j=0, pid, k, x ;
    pid = fork ( );

    if ( pid == 0 ) {
        for ( i = 0; i < 20; i++ ) {
            for (k = 0; k < 10000; k++ );
             printf ("Child: %d\n", i) ;
        }
    }
    else {
        for ( j = 0; j < 20; j++ ){
              for (x = 0; x < 10000; x++ );
              printf ("Parent: %d\n", j) ;
        }
    }
    return 0;
}
```

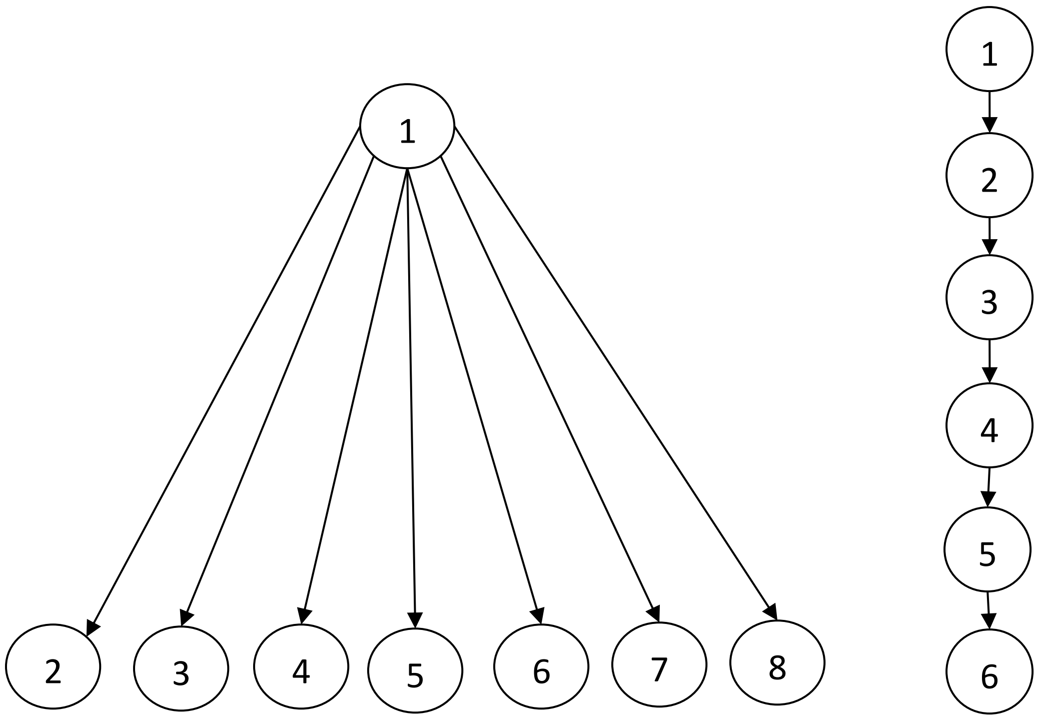


```
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ./task4.o&
[2] 15024
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ Parent: 0
Parent: 1
Parent: 2
Child: 0
Parent: 3
Child: 1
Parent: 4
Child: 2
Parent: 5
Child: 3
Parent: 6
Child: 4
Parent: 7
Child: 5
Parent: 8
Child: 6
Parent: 9
Child: 7
Parent: 10
Child: 8
Parent: 11
Child: 9
Parent: 12
Child: 10
Parent: 13
Child: 11
Parent: 14
Child: 12
Parent: 15
Child: 13
Parent: 16
Child: 14
```



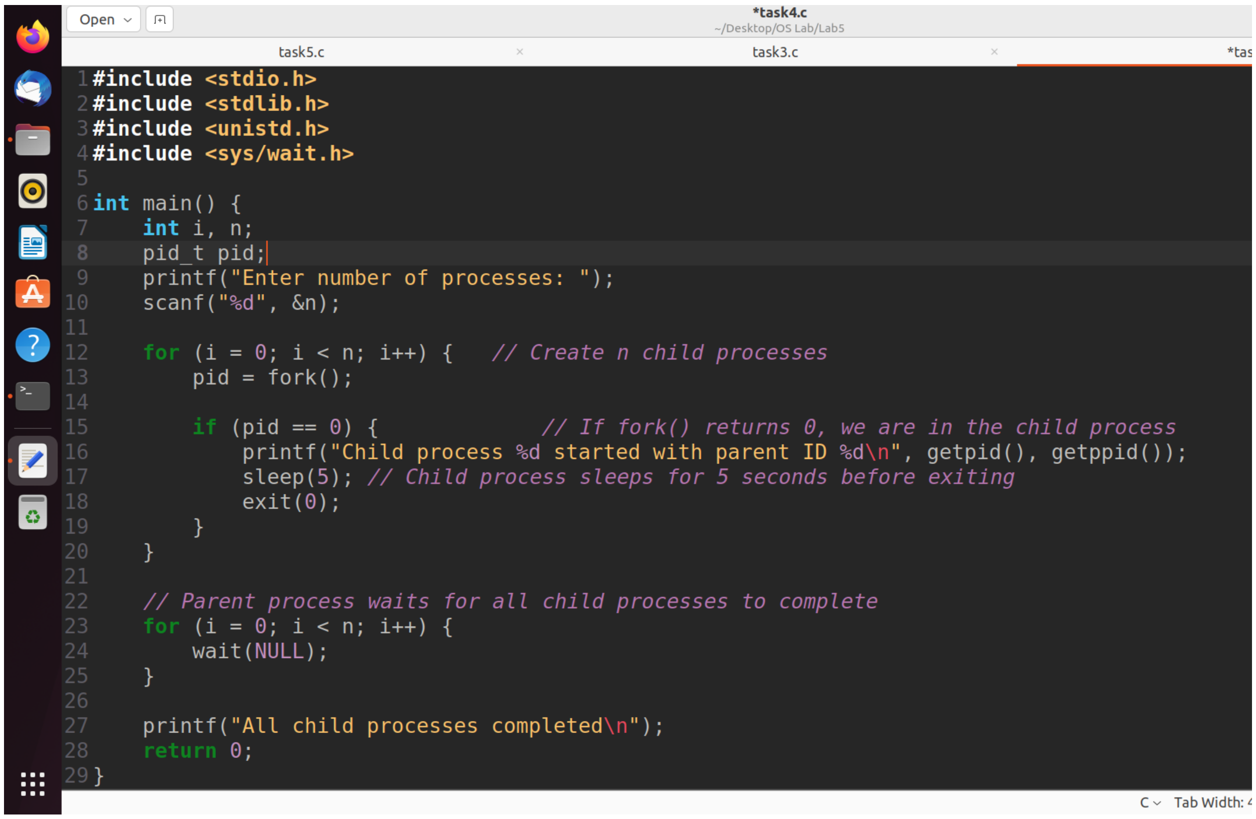
```
Child: 10
Parent: 13
Child: 11
Parent: 14
Child: 12
Parent: 15
Child: 13
Parent: 16
Child: 14
Parent: 17
Child: 15
Parent: 18
Child: 16
Parent: 19
Child: 17
Child: 18
Child: 19
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$
```

4) Create process fan as shown in figure 1 (a) and fill the figure 1 (a) with actual IDs.



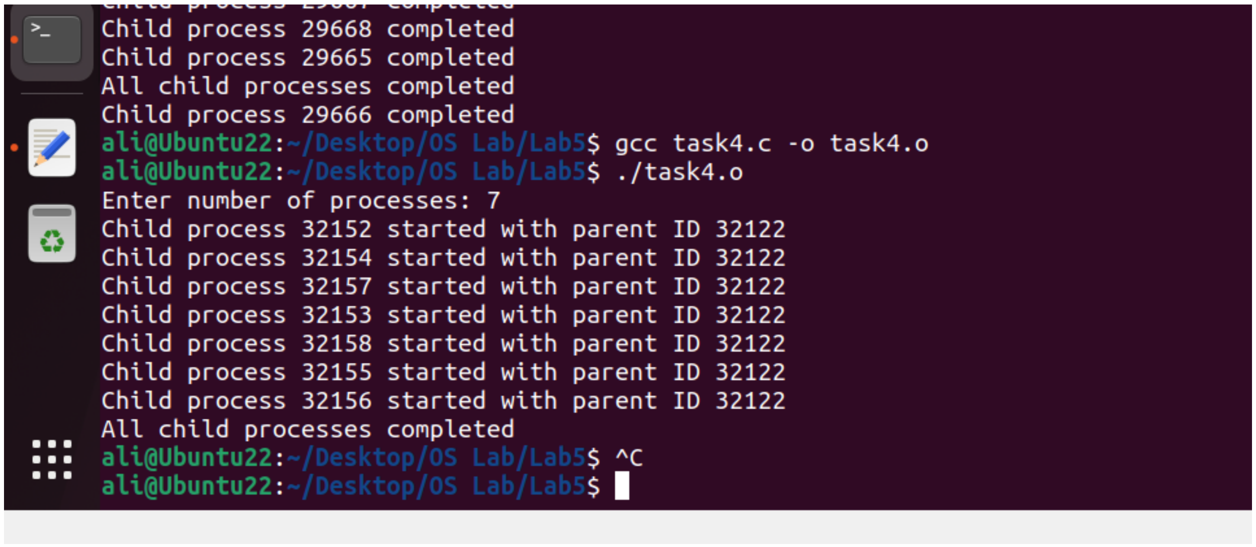
(a) (b)

Figure 1 Multiple Processes (a) Process Fan (b) Process Chain

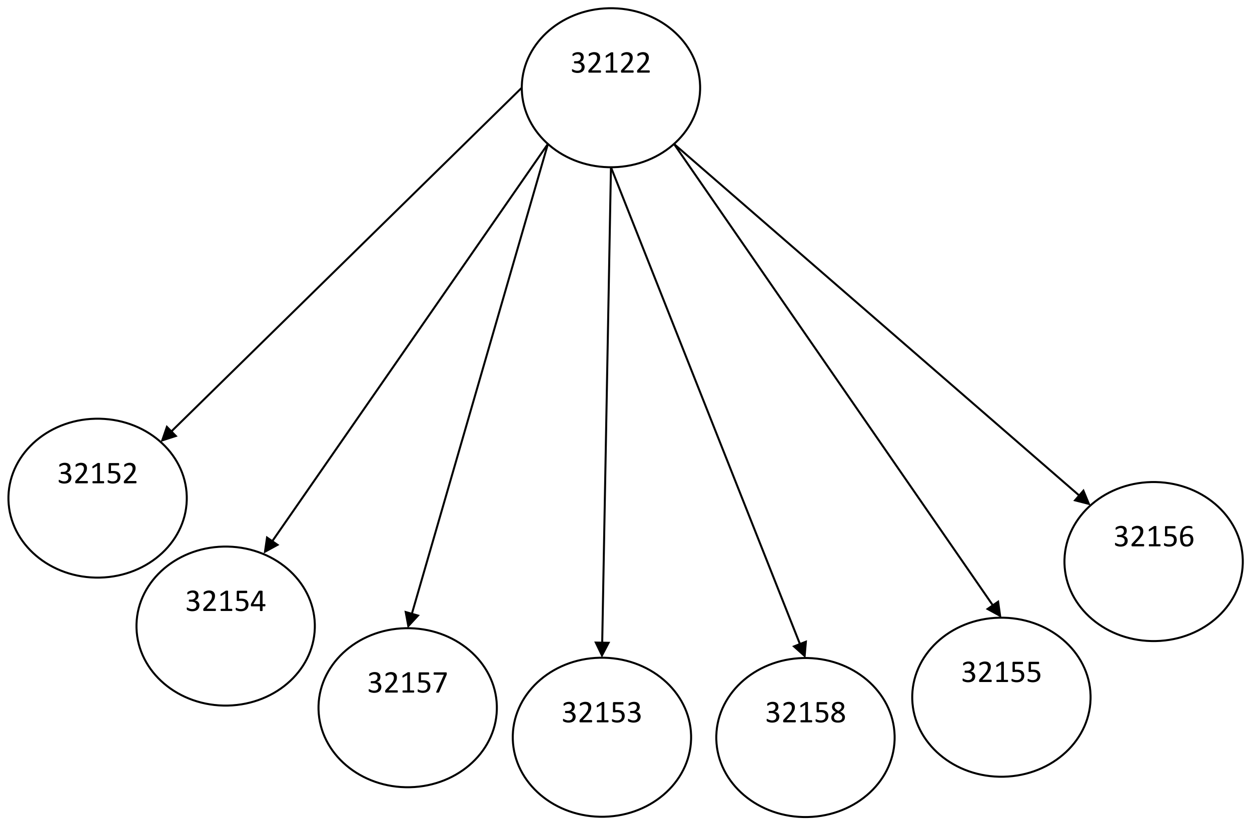

Open | *task4.c ~/Desktop/OS Lab/Lab5 | task5.c | task3.c | *tas

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

int main() {
    int i, n;
    pid_t pid;
    printf("Enter number of processes: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {   // Create n child processes
        pid = fork();

        if (pid == 0) {              // If fork() returns 0, we are in the child process
            printf("Child process %d started with parent ID %d\n", getpid(), getppid());
            sleep(5); // Child process sleeps for 5 seconds before exiting
            exit(0);
        }
    }

    // Parent process waits for all child processes to complete
    for (i = 0; i < n; i++) {
        wait(NULL);
    }

    printf("All child processes completed\n");
    return 0;
}
```

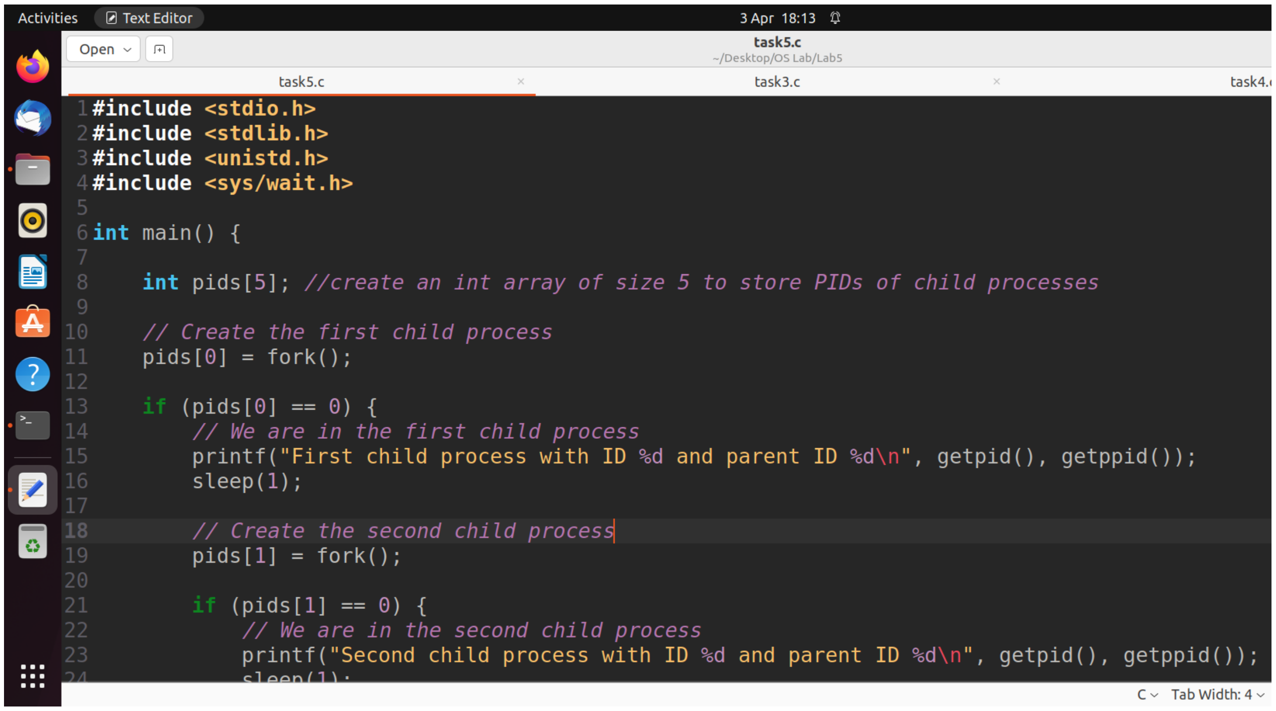
C Tab Width: 4



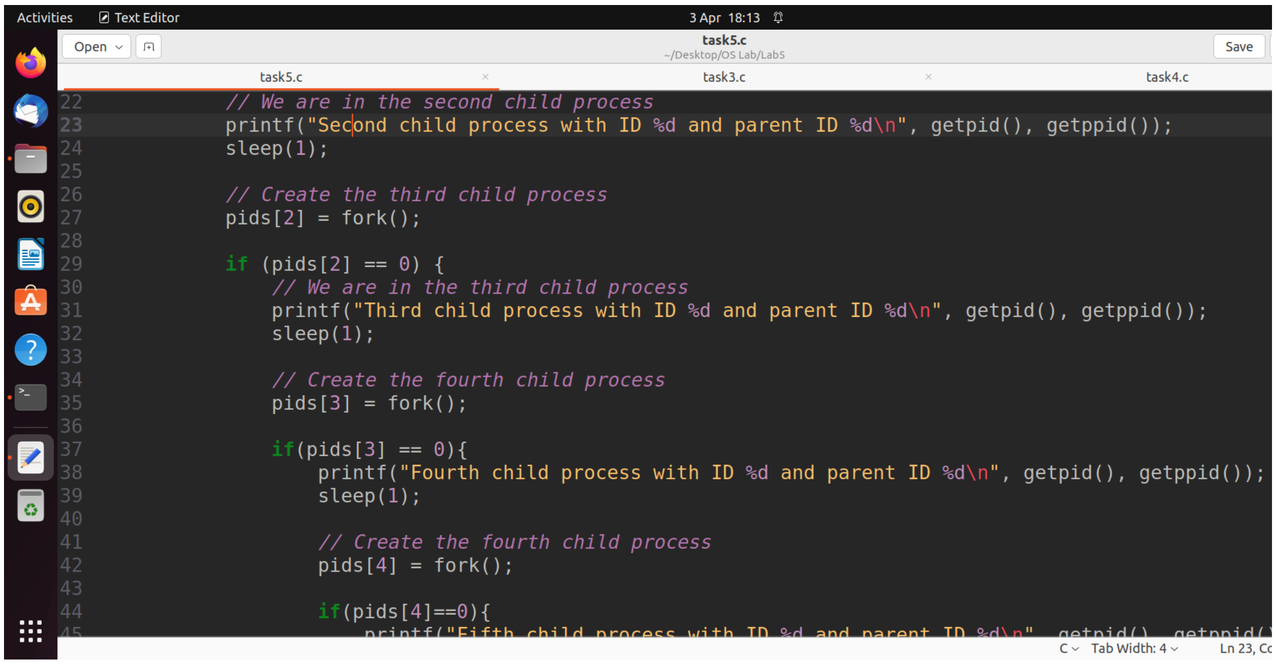
```
Child process 29668 completed
Child process 29665 completed
All child processes completed
Child process 29666 completed
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gcc task4.c -o task4.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ./task4.o
Enter number of processes: 7
Child process 32152 started with parent ID 32122
Child process 32154 started with parent ID 32122
Child process 32157 started with parent ID 32122
Child process 32153 started with parent ID 32122
Child process 32158 started with parent ID 32122
Child process 32155 started with parent ID 32122
Child process 32156 started with parent ID 32122
All child processes completed
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ^C
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$
```


32122
32152
32154
32157
32153
32158
32155
32156

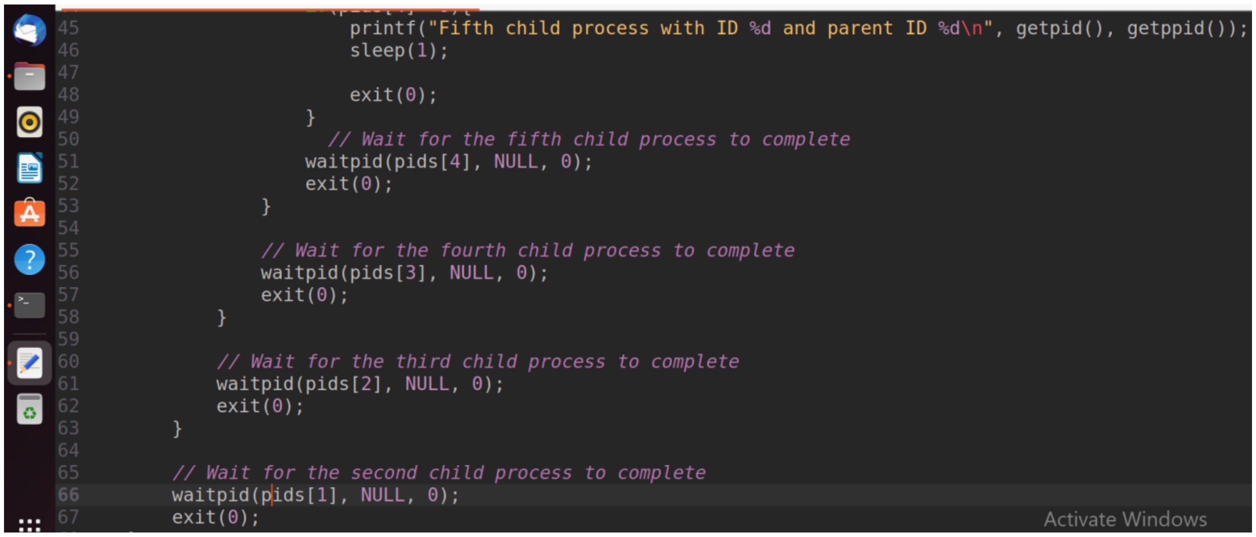
5) Create process chain as shown in figure 1(b) and fill the figure 1 (b) with actual IDs.

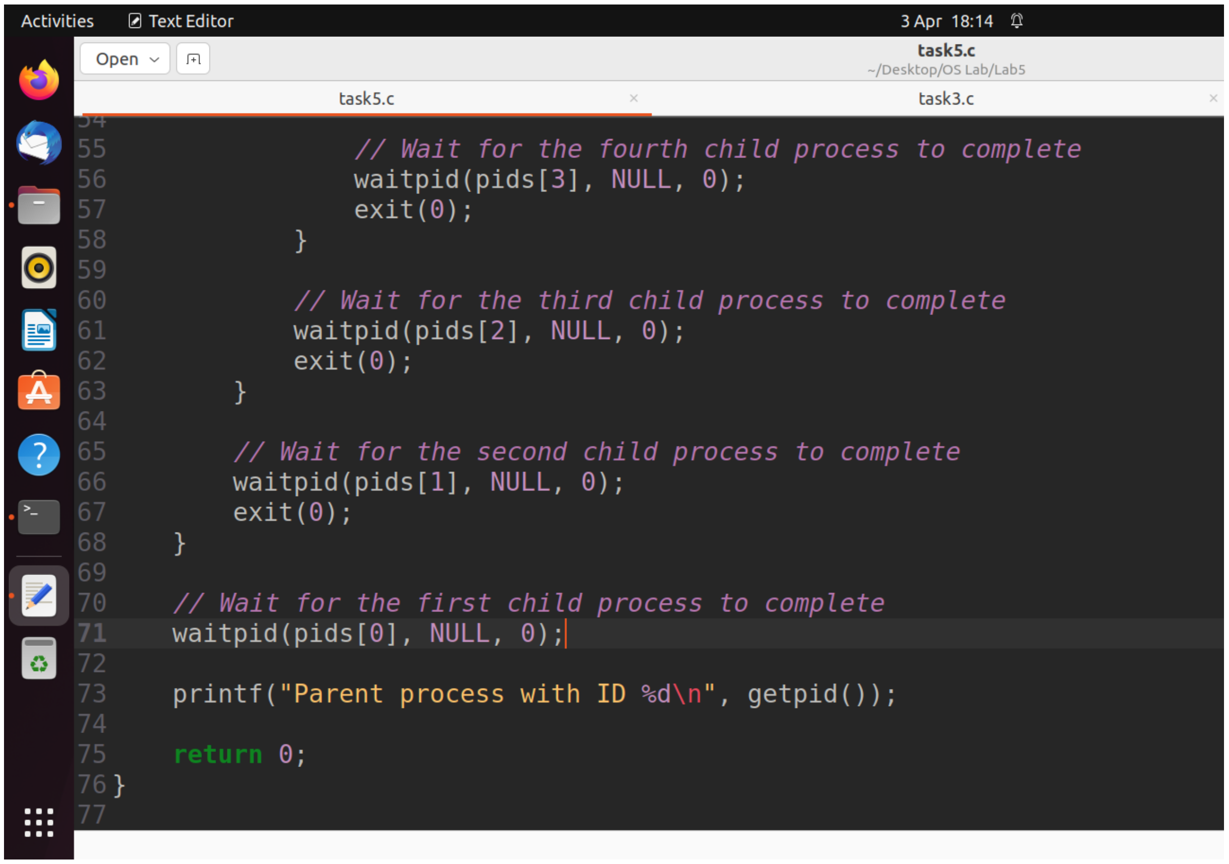


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {

    int pids[5]; //create an int array of size 5 to store PIDs of child processes

    // Create the first child process
    pids[0] = fork();

    if (pids[0] == 0) {
        // We are in the first child process
        printf("First child process with ID %d and parent ID %d\n", getpid(), getppid());
        sleep(1);

        // Create the second child process
        pids[1] = fork();

        if (pids[1] == 0) {
            // We are in the second child process
            printf("Second child process with ID %d and parent ID %d\n", getpid(), getppid());
            sleep(1);
```

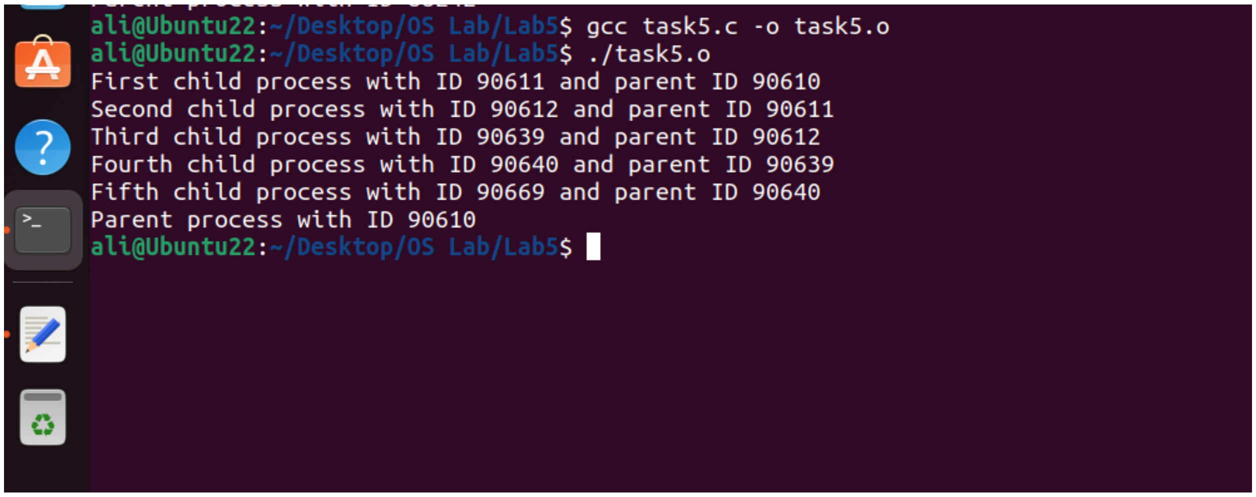


```c
            // We are in the second child process
            printf("Second child process with ID %d and parent ID %d\n", getpid(), getppid());
            sleep(1);

            // Create the third child process
            pids[2] = fork();

            if (pids[2] == 0) {
                // We are in the third child process
                printf("Third child process with ID %d and parent ID %d\n", getpid(), getppid());
                sleep(1);

                // Create the fourth child process
                pids[3] = fork();

                if(pids[3] == 0){
                    printf("Fourth child process with ID %d and parent ID %d\n", getpid(), getppid());
                    sleep(1);

                    // Create the fourth child process
                    pids[4] = fork();

                    if(pids[4]==0){
                        printf("Fifth child process with ID %d and parent ID %d\n", getpid(), getppid(
```

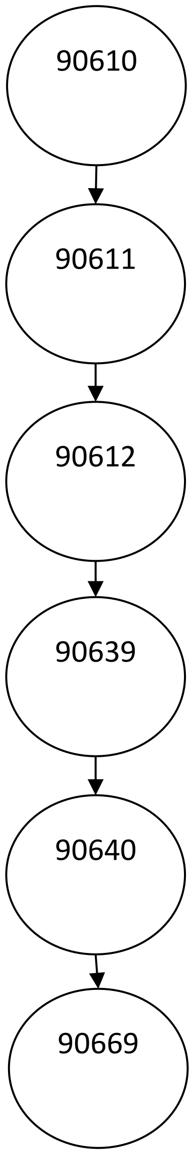

```
                        printf("Fifth child process with ID %d and parent ID %d\n", getpid(), getppid());
                        sleep(1);

                        exit(0);
                    }
                    // Wait for the fifth child process to complete
                    waitpid(pids[4], NULL, 0);
                    exit(0);
                }

                // Wait for the fourth child process to complete
                waitpid(pids[3], NULL, 0);
                exit(0);
            }

            // Wait for the third child process to complete
            waitpid(pids[2], NULL, 0);
            exit(0);
        }

        // Wait for the second child process to complete
        waitpid(pids[1], NULL, 0);
        exit(0);
```
Activate Windows



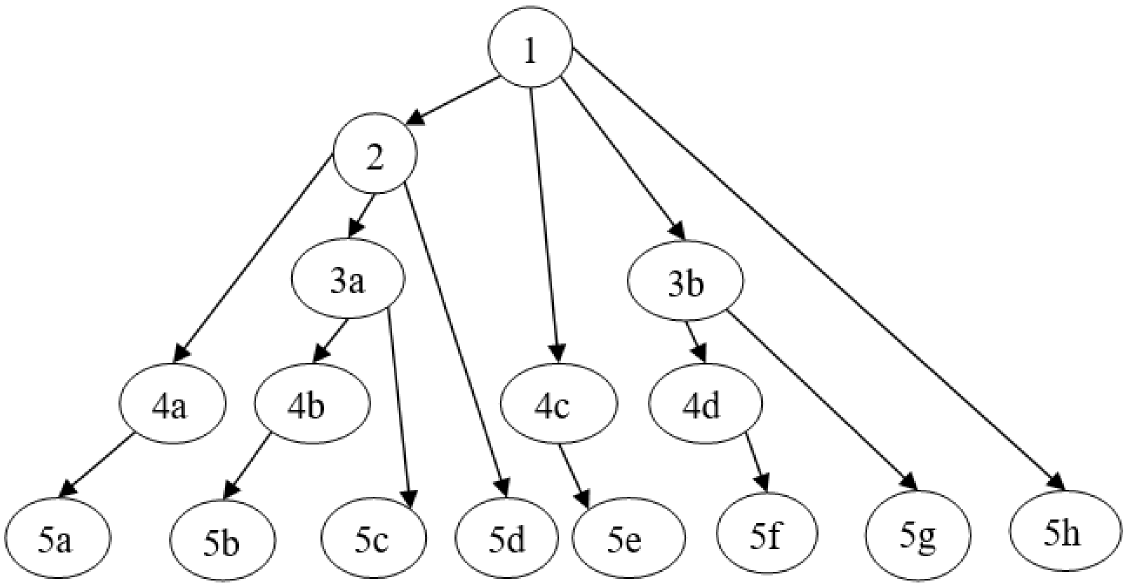
```
                // Wait for the fourth child process to complete
                waitpid(pids[3], NULL, 0);
                exit(0);
            }

            // Wait for the third child process to complete
            waitpid(pids[2], NULL, 0);
            exit(0);
        }

        // Wait for the second child process to complete
        waitpid(pids[1], NULL, 0);
        exit(0);
    }

    // Wait for the first child process to complete
    waitpid(pids[0], NULL, 0);

    printf("Parent process with ID %d\n", getpid());

    return 0;
}
```

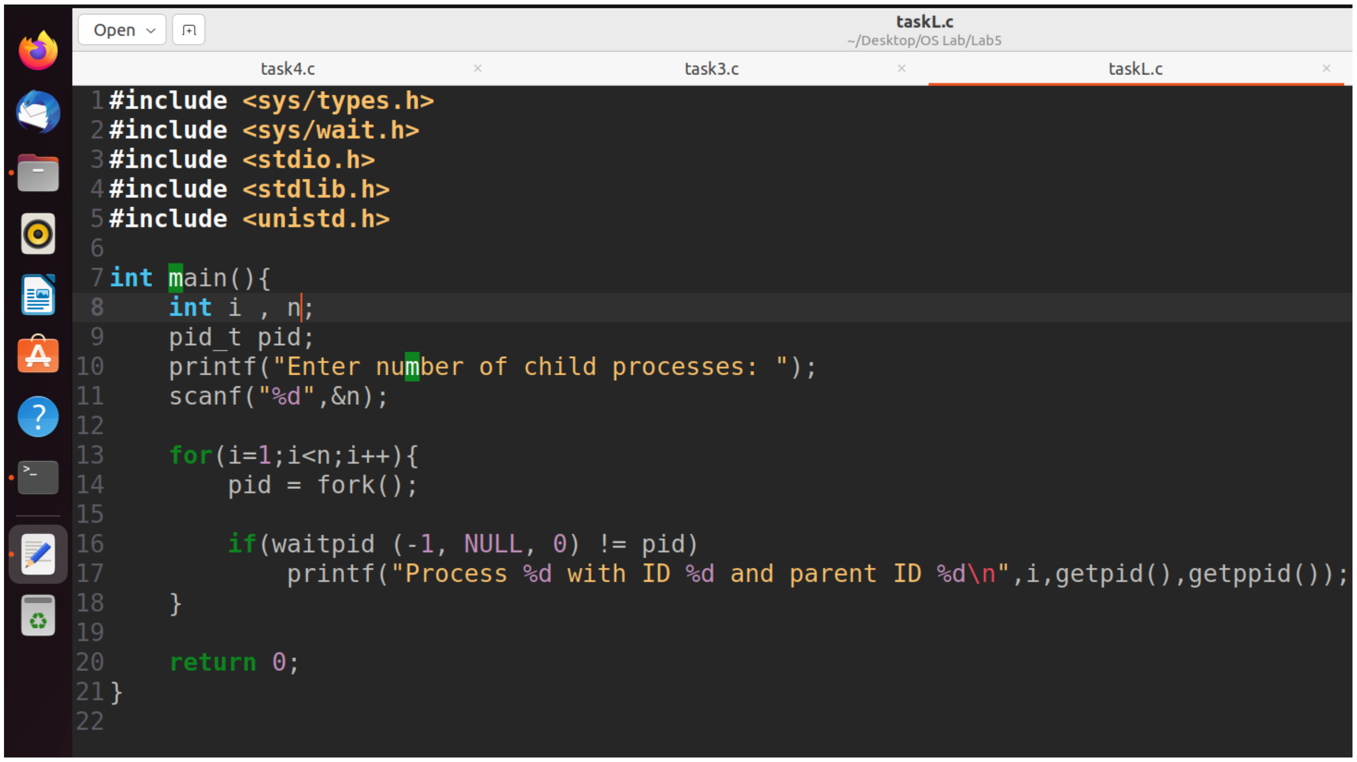


```
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gcc task5.c -o task5.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ./task5.o
First child process with ID 90611 and parent ID 90610
Second child process with ID 90612 and parent ID 90611
Third child process with ID 90639 and parent ID 90612
Fourth child process with ID 90640 and parent ID 90639
Fifth child process with ID 90669 and parent ID 90640
Parent process with ID 90610
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$
```

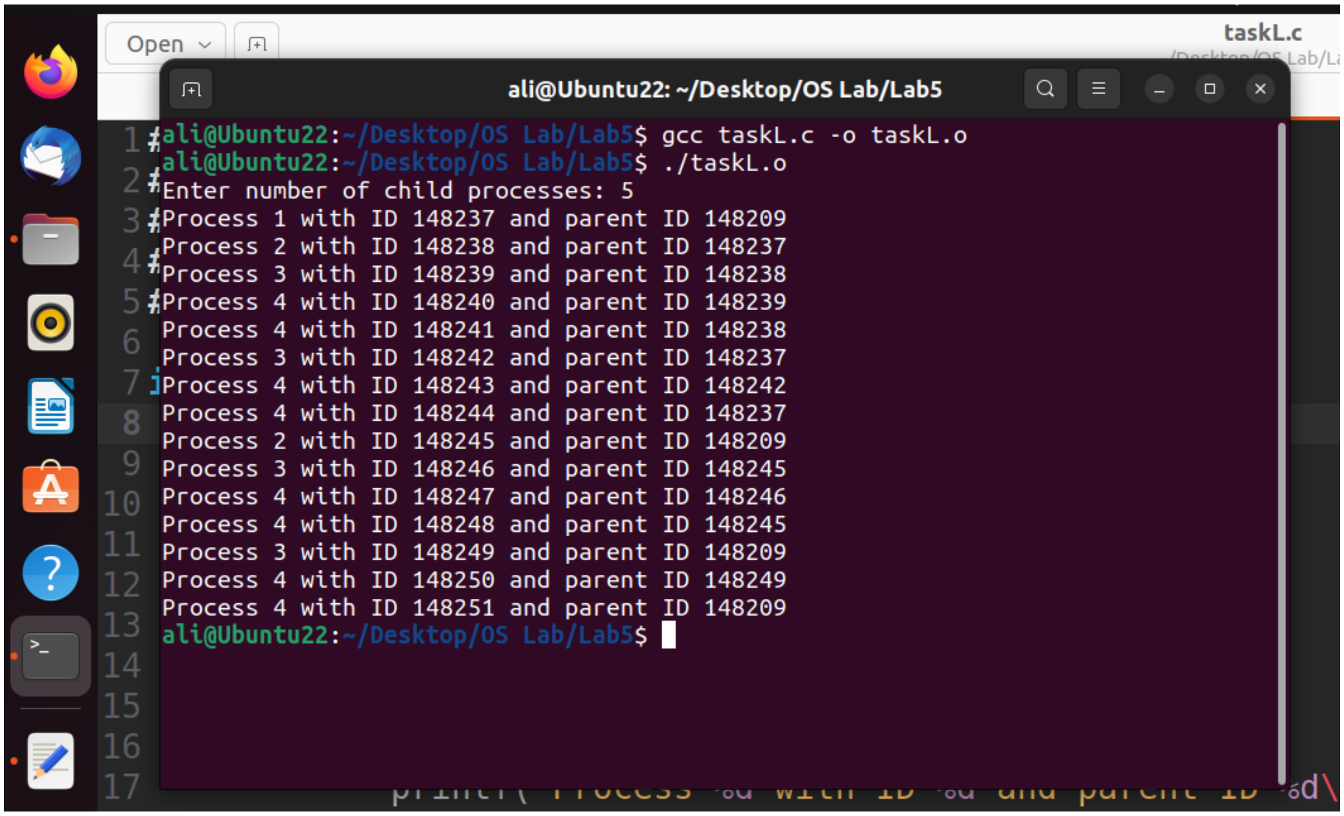


6) Create process tree as shown in figure 2 and fill the figure 2 with actual IDs.

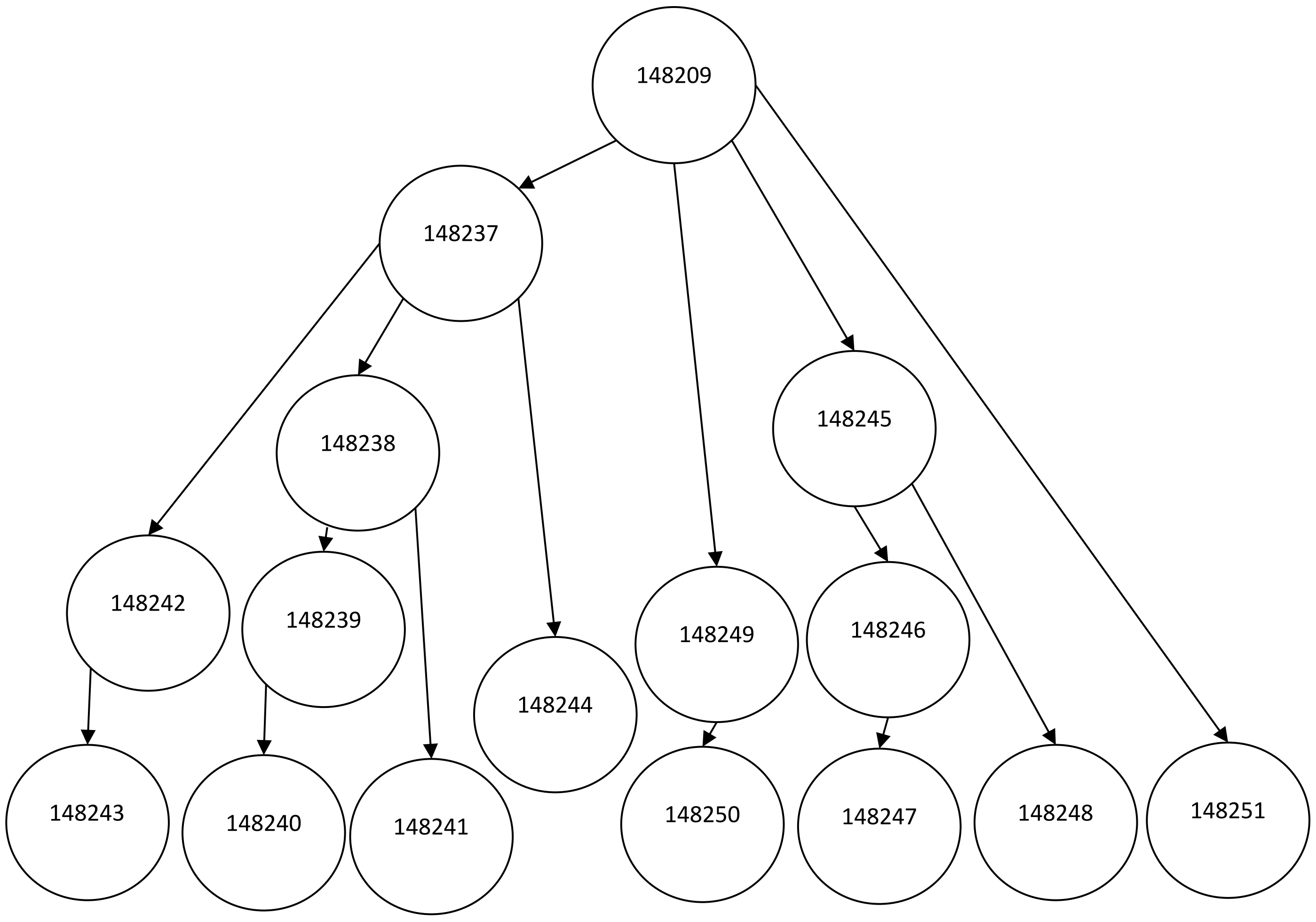




```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(){
    int i , n;
    pid_t pid;
    printf("Enter number of child processes: ");
    scanf("%d",&n);

    for(i=1;i<n;i++){
        pid = fork();

        if(waitpid (-1, NULL, 0) != pid)
            printf("Process %d with ID %d and parent ID %d\n",i,getpid(),getppid());
    }

    return 0;
}
```



```
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ gcc taskL.c -o taskL.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$ ./taskL.o
Enter number of child processes: 5
Process 1 with ID 148237 and parent ID 148209
Process 2 with ID 148238 and parent ID 148237
Process 3 with ID 148239 and parent ID 148238
Process 4 with ID 148240 and parent ID 148239
Process 4 with ID 148241 and parent ID 148238
Process 3 with ID 148242 and parent ID 148237
Process 4 with ID 148243 and parent ID 148242
Process 4 with ID 148244 and parent ID 148237
Process 2 with ID 148245 and parent ID 148209
Process 3 with ID 148246 and parent ID 148245
Process 4 with ID 148247 and parent ID 148246
Process 4 with ID 148248 and parent ID 148245
Process 3 with ID 148249 and parent ID 148209
Process 4 with ID 148250 and parent ID 148249
Process 4 with ID 148251 and parent ID 148209
ali@Ubuntu22:~/Desktop/OS Lab/Lab5$
```


148209
148237
148238
148242
148239
148244
148249
148245
148246
148243
148240
148241
148250
148247
148248
148251