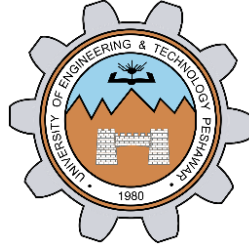


Process Creation, Execution and Termination

LAB # 06



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:

19th May 2023

Department of Computer Systems Engineering
University of Engineering and Technology, Peshawar

OBJECTIVES:

- To understand the concept of process creation, execution, and termination in an operating system.
- To learn how to create child processes and execute commands or programs within them.
- To explore process synchronization and waiting for child processes to complete.
- To demonstrate the creation of multiple child processes and the prevention of orphan processes.

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.

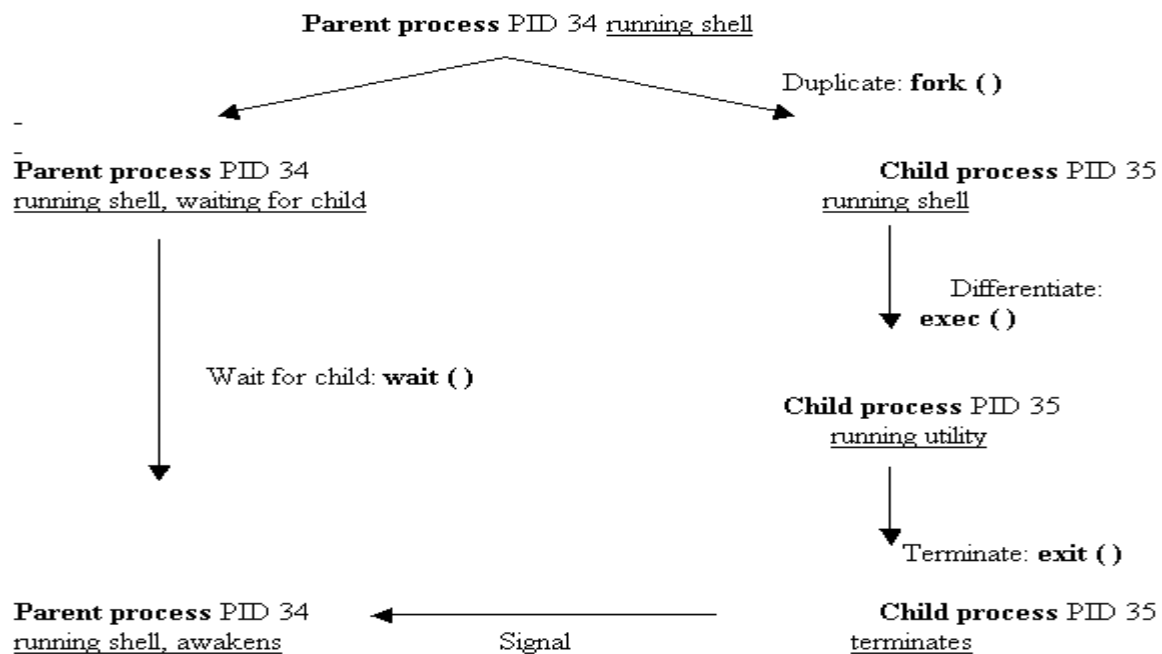
A process that is waiting for its parent to accept its return code is called a **zombie process.**

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

It's 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:



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

PROCESS FAN AND CHAIN:

A process fan and process chain are two different concepts related to the creation and organization of child processes in an operating system. Here's a note explaining each concept:

PROCESS FAN:

- A process fan refers to a pattern where multiple child processes are created from a single parent process.
- In a process fan, the parent process spawns several child processes simultaneously or in a rapid succession, resembling the shape of a fan.
- The child processes created from the parent process in a fan pattern typically have no direct relationship with each other but share a common parent.
- Each child process can perform independent tasks or execute different sections of code.
- The primary purpose of creating a process fan is to achieve parallelism or to distribute workload among multiple processes.
- Examples of scenarios where a process fan may be useful include parallel processing, distributed computing, and implementing concurrent algorithms.

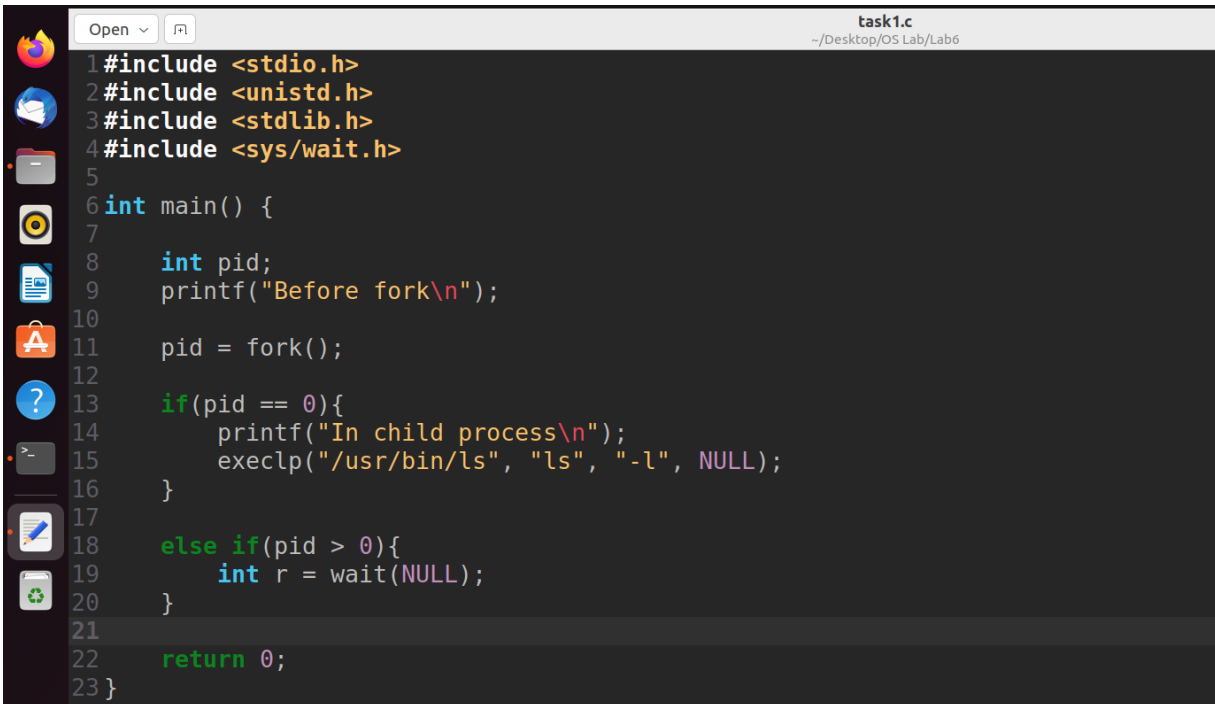
PROCESS CHAIN:

- A process chain refers to a sequential arrangement of child processes, where each child process is created by its immediate parent process.
- In a process chain, the first child process is created by the parent process. Subsequently, each child process creates another child process until a specific condition is met or the desired number of processes is reached.
- The child processes created in a process chain form a linear sequence, with each child process having a direct parent-child relationship with the preceding process in the chain.

- Process chains are often used to establish a logical order or dependency among child processes, where each process relies on the completion of its parent process before it can start its execution.
- Process chains are commonly employed in scenarios such as task scheduling, dependency management, and implementing sequential algorithms.

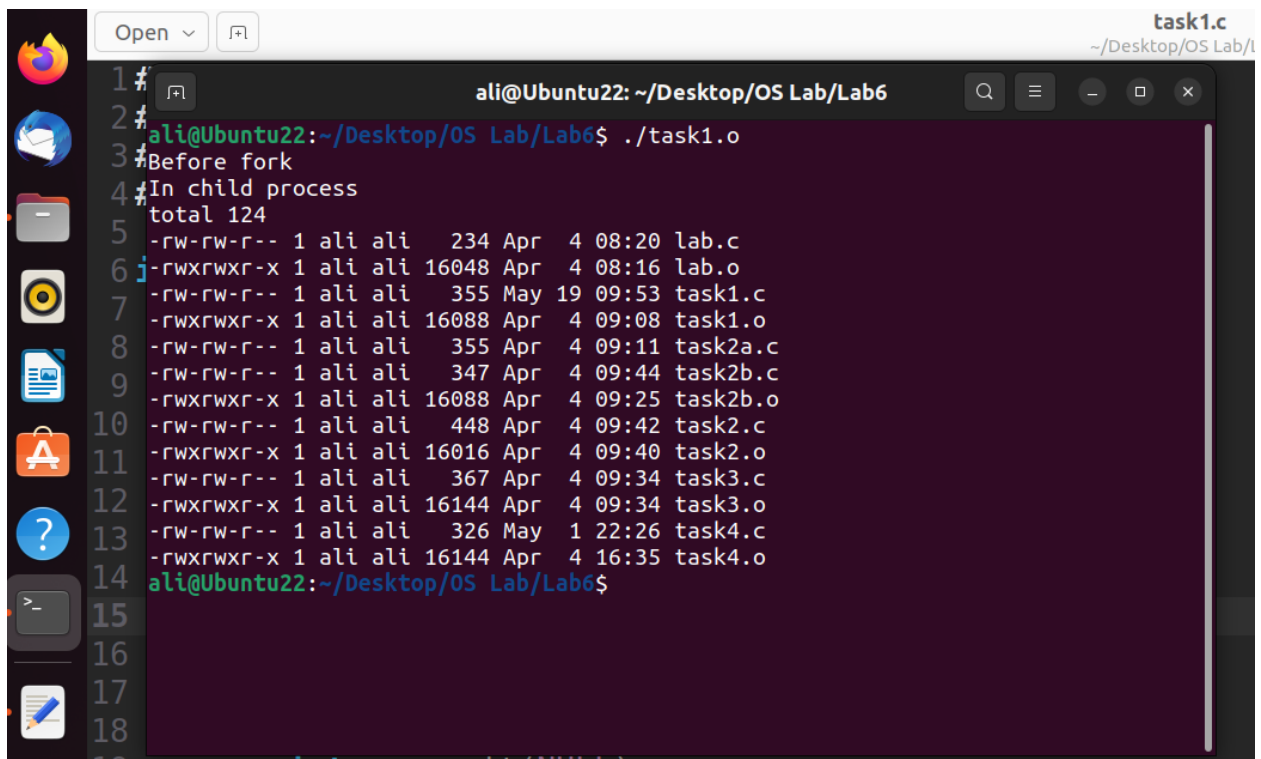
Task1:

Write a C program that executes `ls -l` command in the child process. Parent process shall wait for the child process.



```
task1.c
~/Desktop/OS Lab/Lab6

1#include <stdio.h>
2#include <unistd.h>
3#include <stdlib.h>
4#include <sys/wait.h>
5
6int main() {
7
8    int pid;
9    printf("Before fork\n");
10
11    pid = fork();
12
13    if(pid == 0){
14        printf("In child process\n");
15        execlp("/usr/bin/ls", "ls", "-l", NULL);
16    }
17
18    else if(pid > 0){
19        int r = wait(NULL);
20    }
21
22    return 0;
23}
```

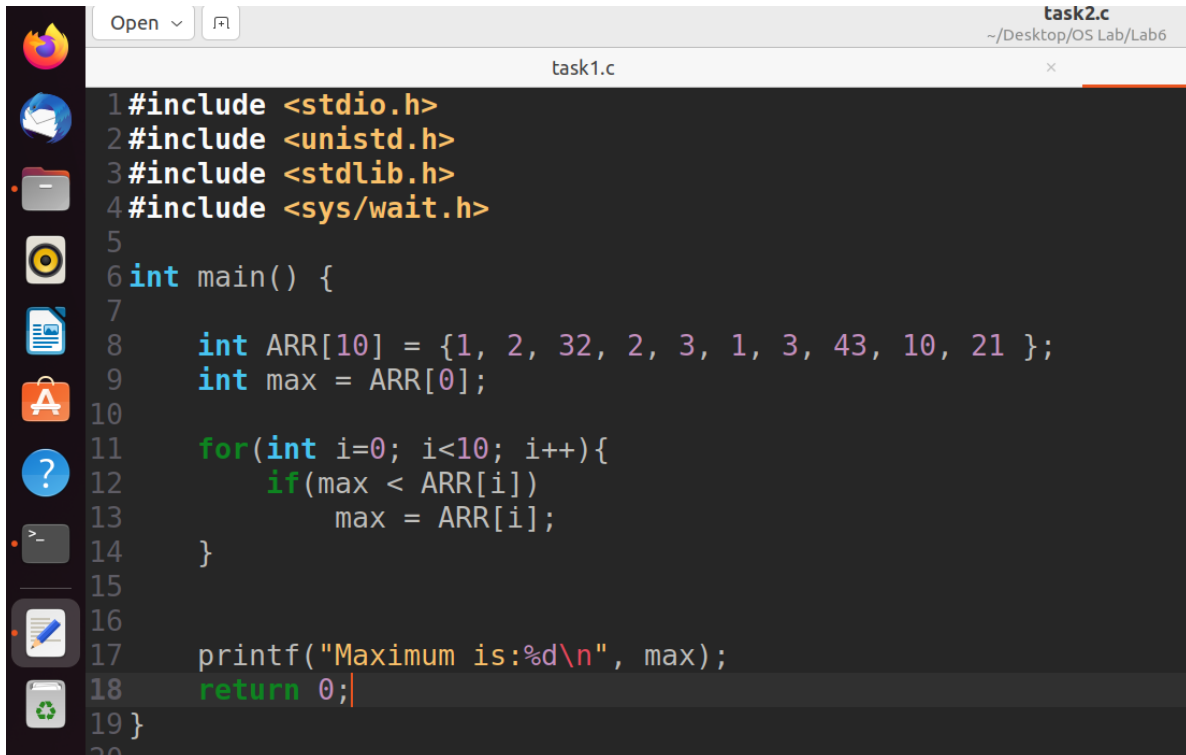


```
task1.c
~/Desktop/OS Lab/Lab6

ali@Ubuntu22: ~/Desktop/OS Lab/Lab6
1#
2# ali@Ubuntu22:~/Desktop/OS Lab/Lab6$ ./task1.o
3#Before fork
4#In child process
5total 124
6-rw-rw-r-- 1 ali ali 234 Apr 4 08:20 lab.c
7-rwxrwxr-x 1 ali ali 16048 Apr 4 08:16 lab.o
8-rw-rw-r-- 1 ali ali 355 May 19 09:53 task1.c
9-rwxrwxr-x 1 ali ali 16088 Apr 4 09:08 task1.o
10-rw-rw-r-- 1 ali ali 355 Apr 4 09:11 task2a.c
11-rw-rw-r-- 1 ali ali 347 Apr 4 09:44 task2b.c
12-rwxrwxr-x 1 ali ali 16088 Apr 4 09:25 task2b.o
13-rw-rw-r-- 1 ali ali 448 Apr 4 09:42 task2.c
14-rwxrwxr-x 1 ali ali 16016 Apr 4 09:40 task2.o
15-rw-rw-r-- 1 ali ali 367 Apr 4 09:34 task3.c
16-rwxrwxr-x 1 ali ali 16144 Apr 4 09:34 task3.o
17-rw-rw-r-- 1 ali ali 326 May 1 22:26 task4.c
18-rwxrwxr-x 1 ali ali 16144 Apr 4 16:35 task4.o
19ali@Ubuntu22:~/Desktop/OS Lab/Lab6$
```

Task2:

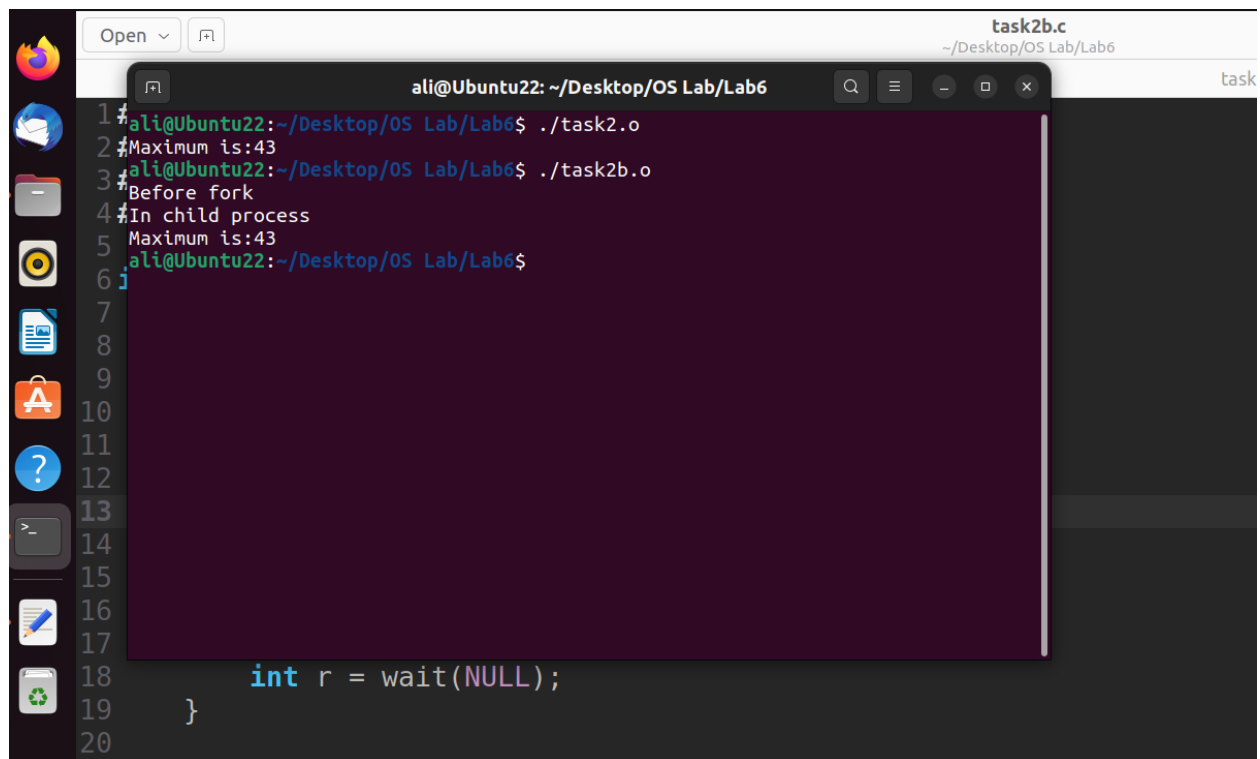
- Write a C program that finds the max of an array.
- Write a C program that creates a child process and executes the above program in child process. Parent shall wait for the child process.

A screenshot of a code editor window titled 'task1.c'. The editor has a dark theme and a sidebar on the left with various icons. The code is a C program to find the maximum value in an array. It includes headers for stdio, unistd, stdlib, and sys/wait. The main function declares an array 'ARR' with 10 elements and a variable 'max' to store the maximum value. It uses a for loop to iterate through the array and update 'max' if a larger value is found. Finally, it prints the maximum value and returns 0.

```
1#include <stdio.h>
2#include <unistd.h>
3#include <stdlib.h>
4#include <sys/wait.h>
5
6int main() {
7
8    int ARR[10] = {1, 2, 32, 2, 3, 1, 3, 43, 10, 21 };
9    int max = ARR[0];
10
11    for(int i=0; i<10; i++){
12        if(max < ARR[i])
13            max = ARR[i];
14    }
15
16    printf("Maximum is:%d\n", max);
17    return 0;
18 }
19
20
```

A screenshot of a code editor window showing three tabs: 'task1.c', 'task2.c', and 'task2a.c'. The 'task2.c' tab is active, displaying a C program that creates a child process to execute 'task2.o'. The parent process prints 'Before fork\n', calls 'fork()' to create a child, and then uses 'execlp' to execute the child process. The parent process then calls 'wait(NULL)' to wait for the child to finish before returning 0.

```
1#include <stdio.h>
2#include <unistd.h>
3#include <stdlib.h>
4#include <sys/wait.h>
5
6int main() {
7
8    int pid;
9    printf("Before fork\n");
10    pid = fork();
11
12    if(pid == 0){
13        printf("In child process\n");
14        execlp("./task2.o", "task2.o", NULL);
15    }
16
17    else if(pid > 0){
18        int r = wait(NULL);
19    }
20
21    return 0;
22 }
```

```
1 #ali@Ubuntu22:~/Desktop/OS Lab/Lab6$ ./task2.o
2 #Maximum is:43
3 #ali@Ubuntu22:~/Desktop/OS Lab/Lab6$ ./task2b.o
4 #Before fork
5 #In child process
6 #Maximum is:43
7
8
9
10
11
12
13
14
15
16
17
18     int r = wait(NULL);
19 }
20
```

Task3:

Create a fan of N processes. Take N as input from the user. Make sure there are no orphan processes.

```
*task3.c
~/Desktop/OS Lab/Lab6

1#include <stdio.h>
2#include <unistd.h>
3#include <stdlib.h>
4#include <sys/wait.h>
5
6int main() {
7
8    int pid,n;
9    printf("Enter n");
10   scanf("%d", &n);e
11
12   printf("Parent Process With ID:%d\n", getpid());
13   for(int i = 1; i<=n; i++){
14       pid = fork();
15
16       if(pid == 0 ){
17           printf("Child %d With ID:%d and Parent ID:%d\n",i, getpid(), getppid());
18           break;
19       }
20   }
21
22   if(pid > 0){
23       for(int i=0; i<n; i++)
24           int r = wait(NULL);
25   }
26   return 0;
27 }
28
```

```
task3
~/Desktop/OS

ali@Ubuntu22: ~/Desktop/OS Lab/Lab6
ali@Ubuntu22:~/Desktop/OS Lab/Lab6$ gedit task3.c&
[1] 27683
ali@Ubuntu22:~/Desktop/OS Lab/Lab6$ gcc task3.c -o task3.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab6$ ./task3.o
Enter n5
Parent Process With ID:27745
Child 1 With ID:27746 and Parent ID:27745
Child 2 With ID:27747 and Parent ID:27745
Child 4 With ID:27749 and Parent ID:27745
Child 5 With ID:27750 and Parent ID:27745
Child 3 With ID:27748 and Parent ID:27745
ali@Ubuntu22:~/Desktop/OS Lab/Lab6$
```

Task4:

Create a chain of N processes. Take N as input from user. Make sure there are no orphan processes.

```
*task4.c
~/Desktop/OS Lab/Lab6

1#include <stdio.h>
2#include <unistd.h>
3#include <stdlib.h>
4#include <sys/wait.h>
5
6int main() {
7
8    int pid,n;
9    printf("Enter n");
10   scanf("%d", &n);
11
12   for(int i = 1; i<=n; i++){
13       pid = fork();
14
15       if(pid > 0){
16           printf("Process:%d With ID:%d and Parent ID:%d\n",i,getpid(), getppid());
17           break;
18       }
19   }
20
21   if(pid > 0){
22       int r = wait(NULL);
23   }
24
25   return 0;
26 }
```

```
ali@Ubuntu22: ~/Desktop/OS Lab/Lab6
1#
2#
3ali@Ubuntu22:~/Desktop/OS Lab/Lab6$ gedit task4.c&
4# [1] 26733
5ali@Ubuntu22:~/Desktop/OS Lab/Lab6$ gcc task4.c -o task4.o
6ali@Ubuntu22:~/Desktop/OS Lab/Lab6$ ./task4.o
7Enter n5
8Process:1 With ID:27184 and Parent ID:26628
9Process:2 With ID:27190 and Parent ID:27184
10Process:3 With ID:27191 and Parent ID:27190
11Process:4 With ID:27192 and Parent ID:27191
12Process:5 With ID:27193 and Parent ID:27192
ali@Ubuntu22:~/Desktop/OS Lab/Lab6$
```