**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.

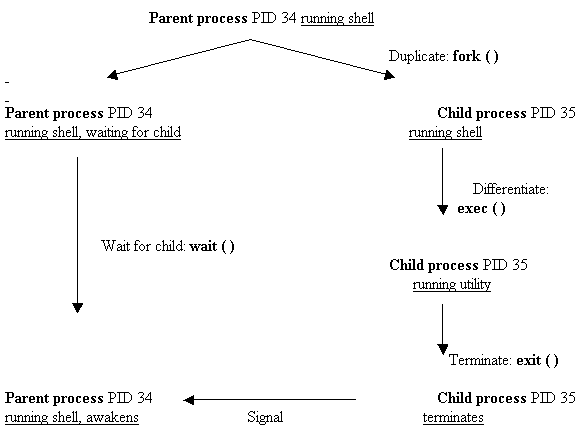
When a child process terminates, its death is communicated to its parent so that the parent may take some appropriate action.

**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.**

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:**



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:**

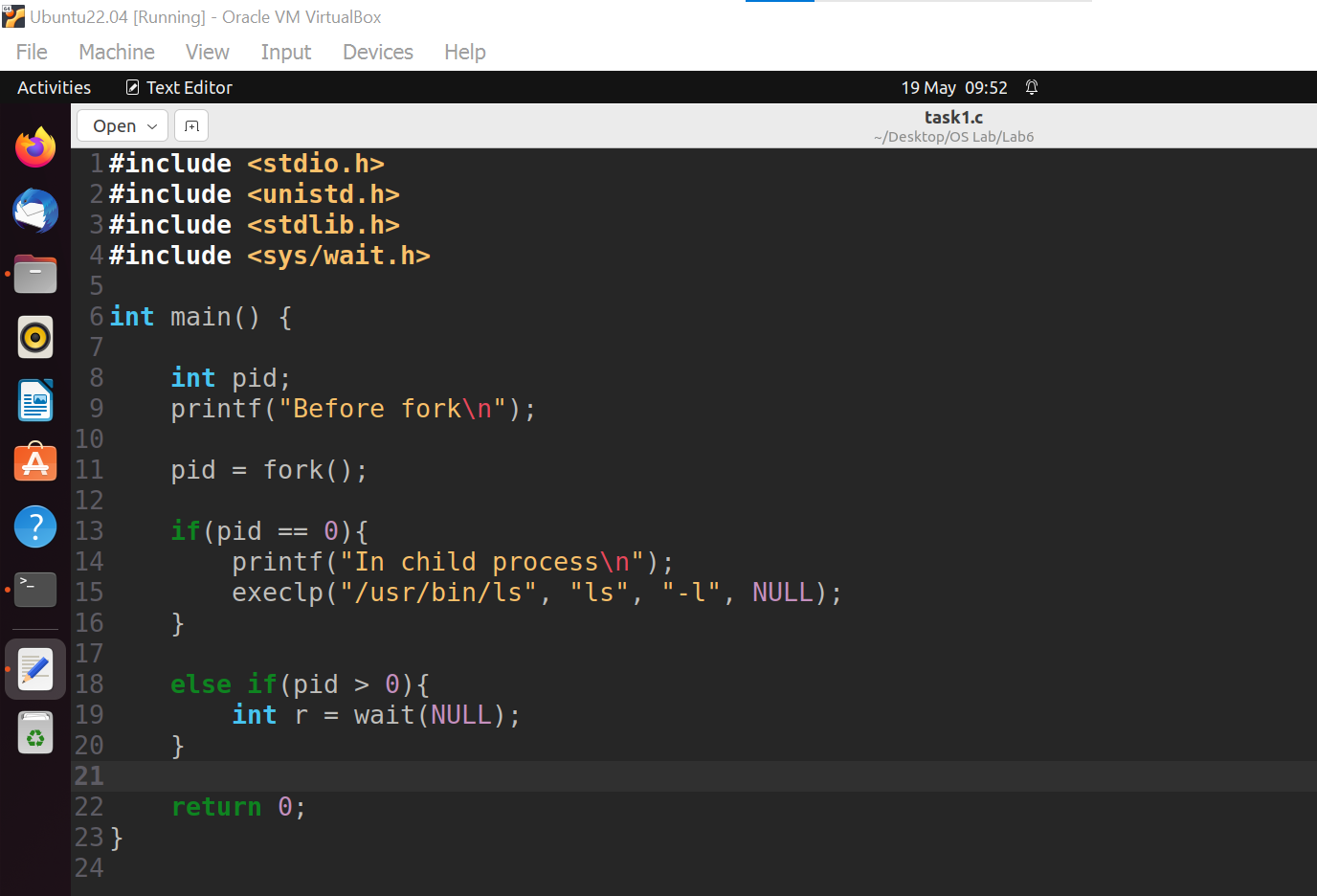
* A process fan refers to a pattern where multiple child processes are created from a single parent process.
* In a process fan, the parent process spawns several child processes simultaneously or in a rapid succession, resembling the shape of a fan.
* The child processes created from the parent process 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.

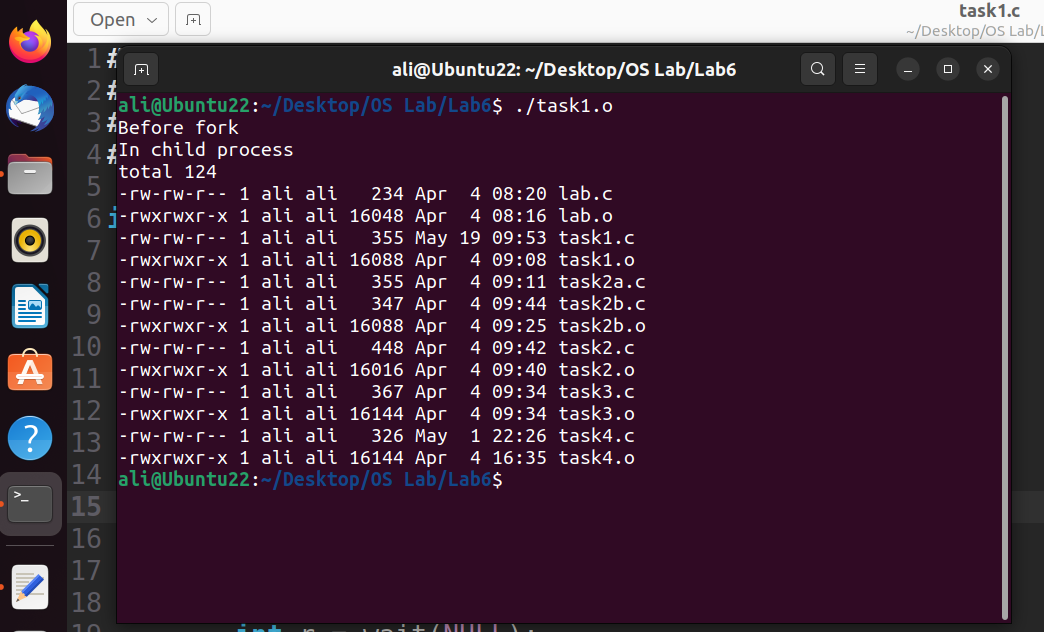
**Process Chain:**

* 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:**

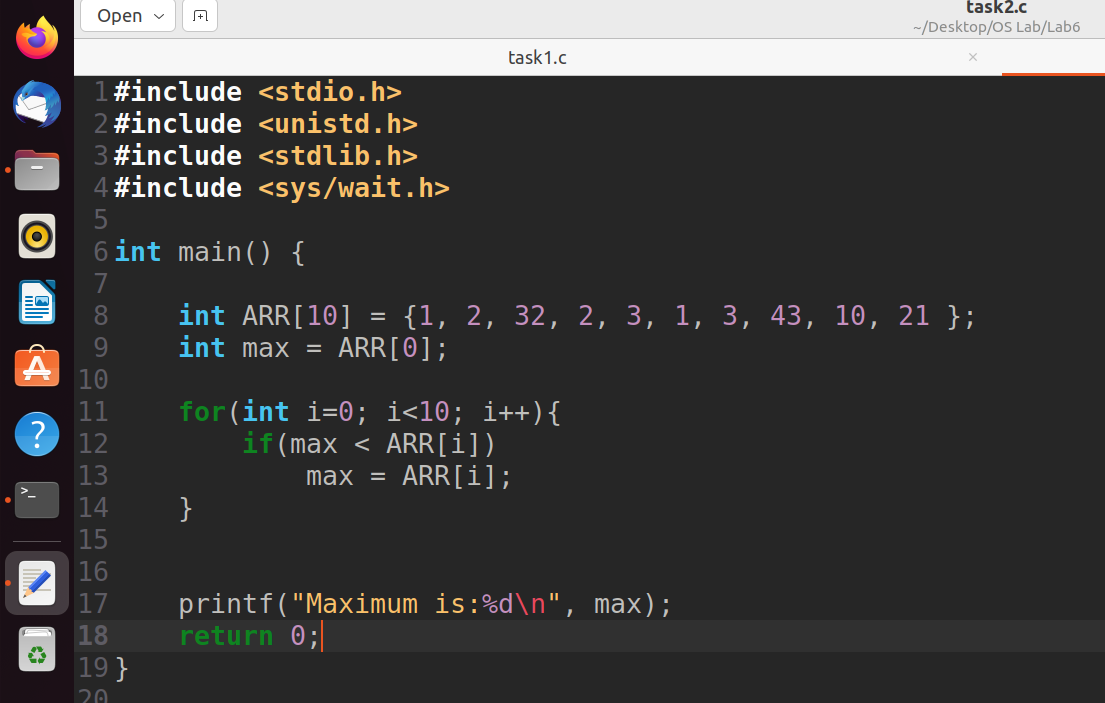
Write a C program that executes ls -l command in the child process. Parent process shall wait for the child process.

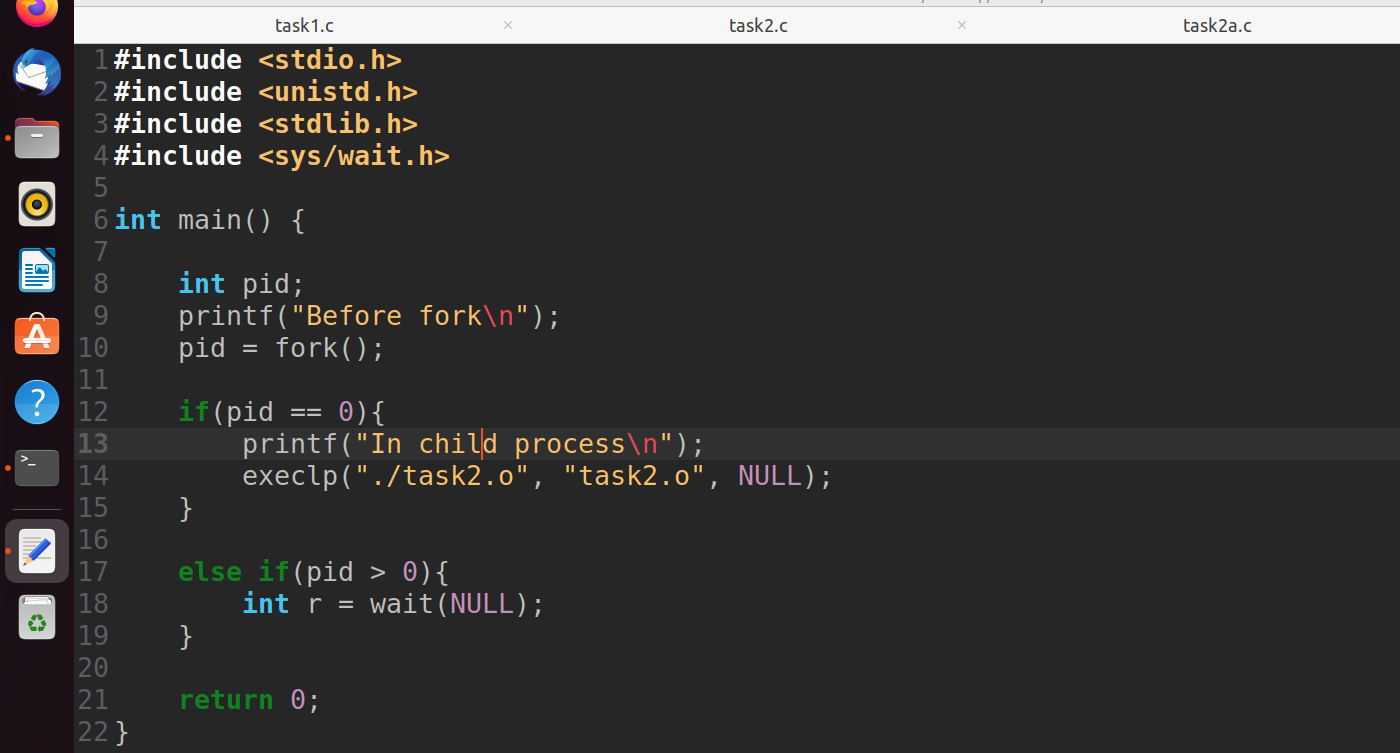


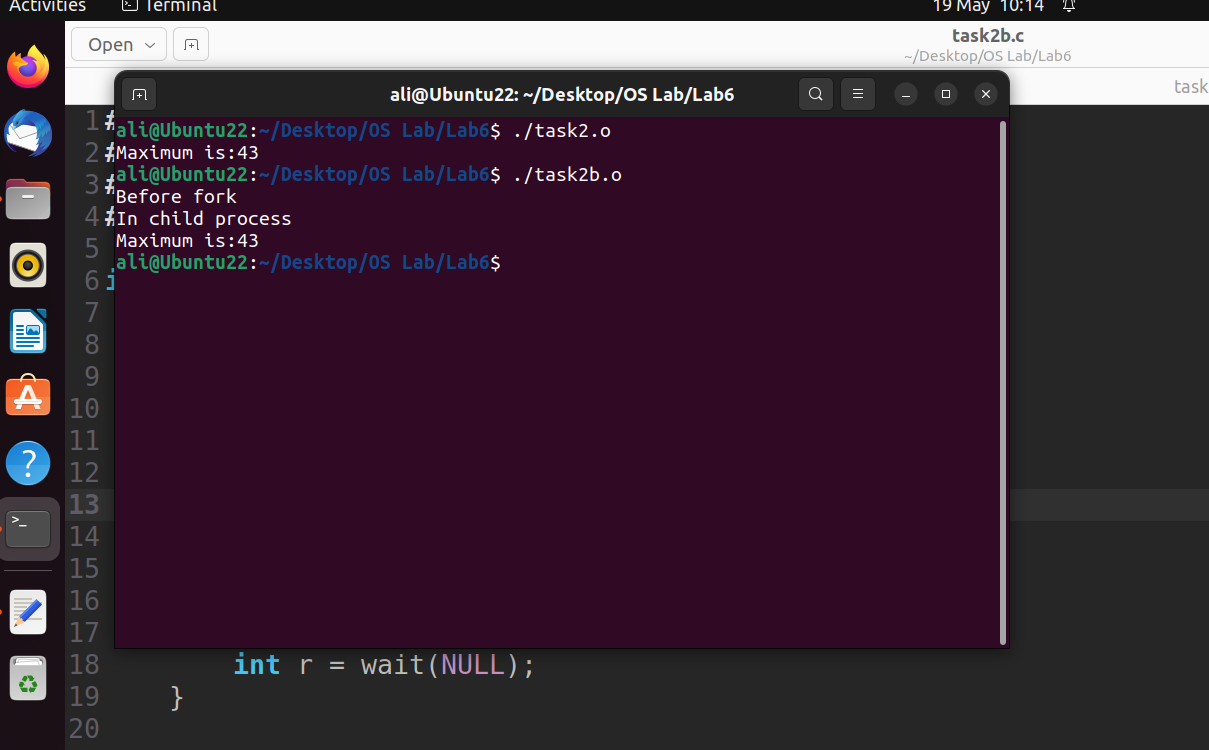


**Task2:**

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

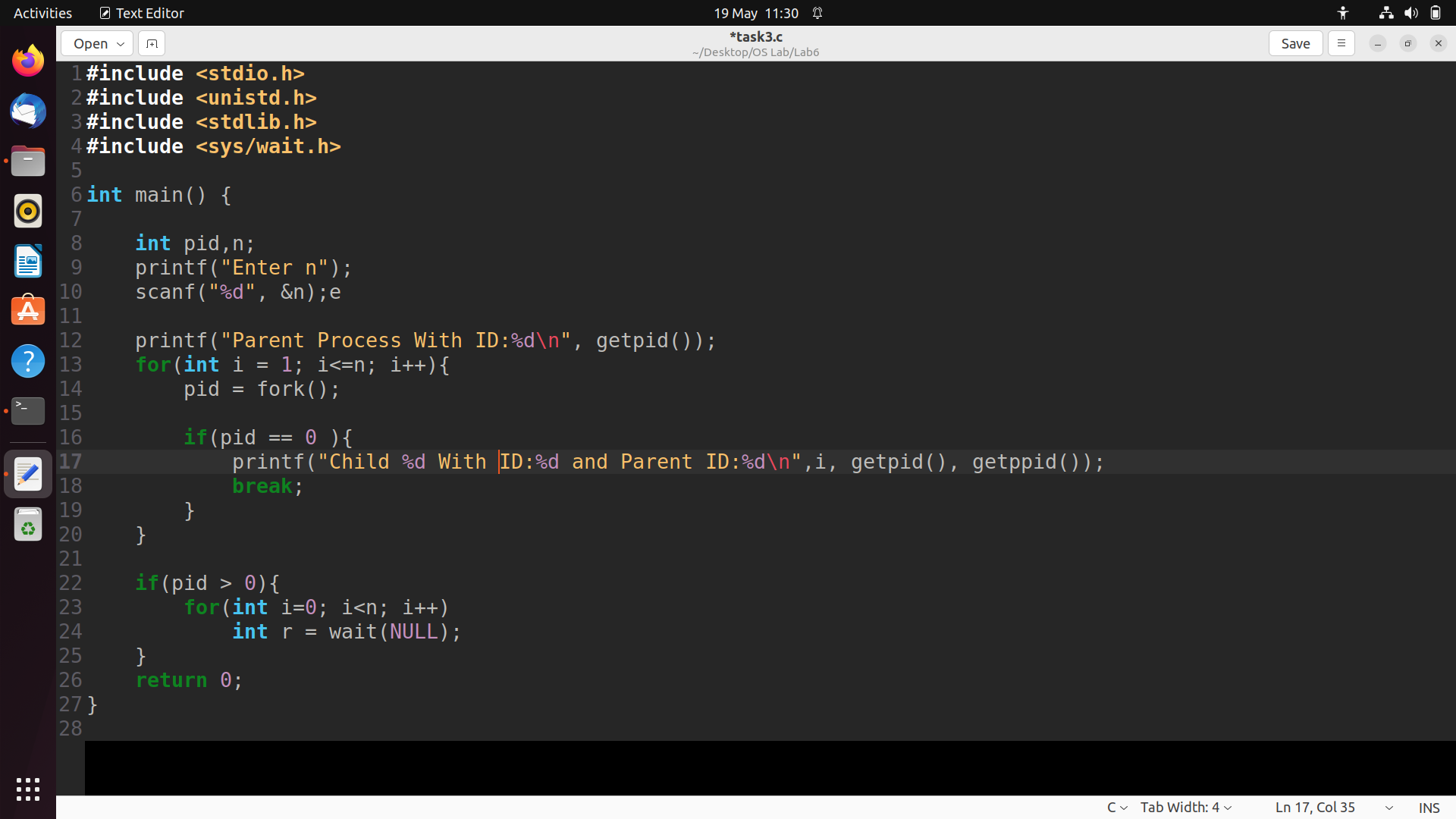


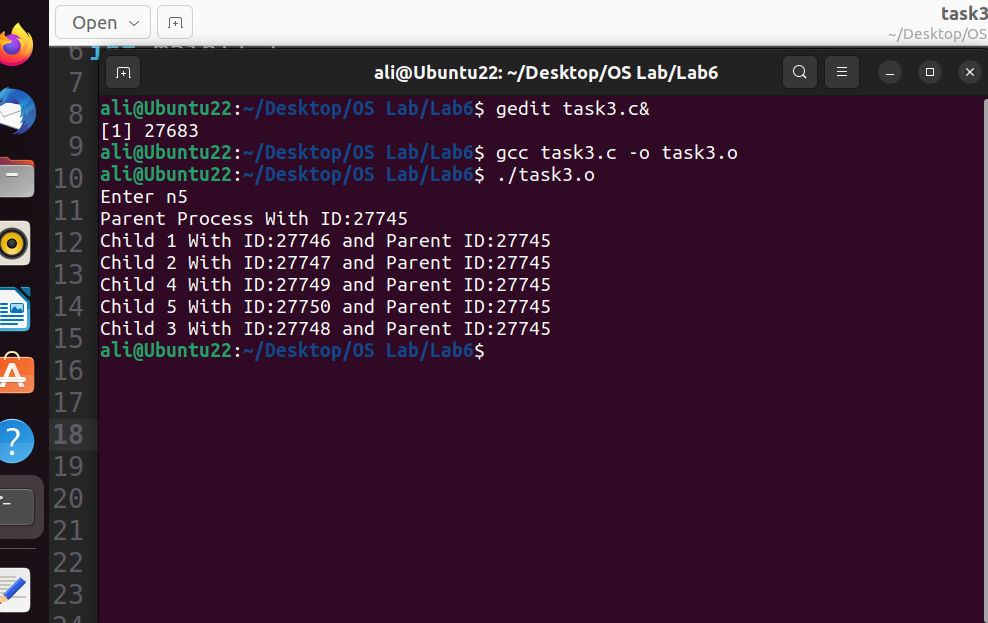
****

****

**Task3:**

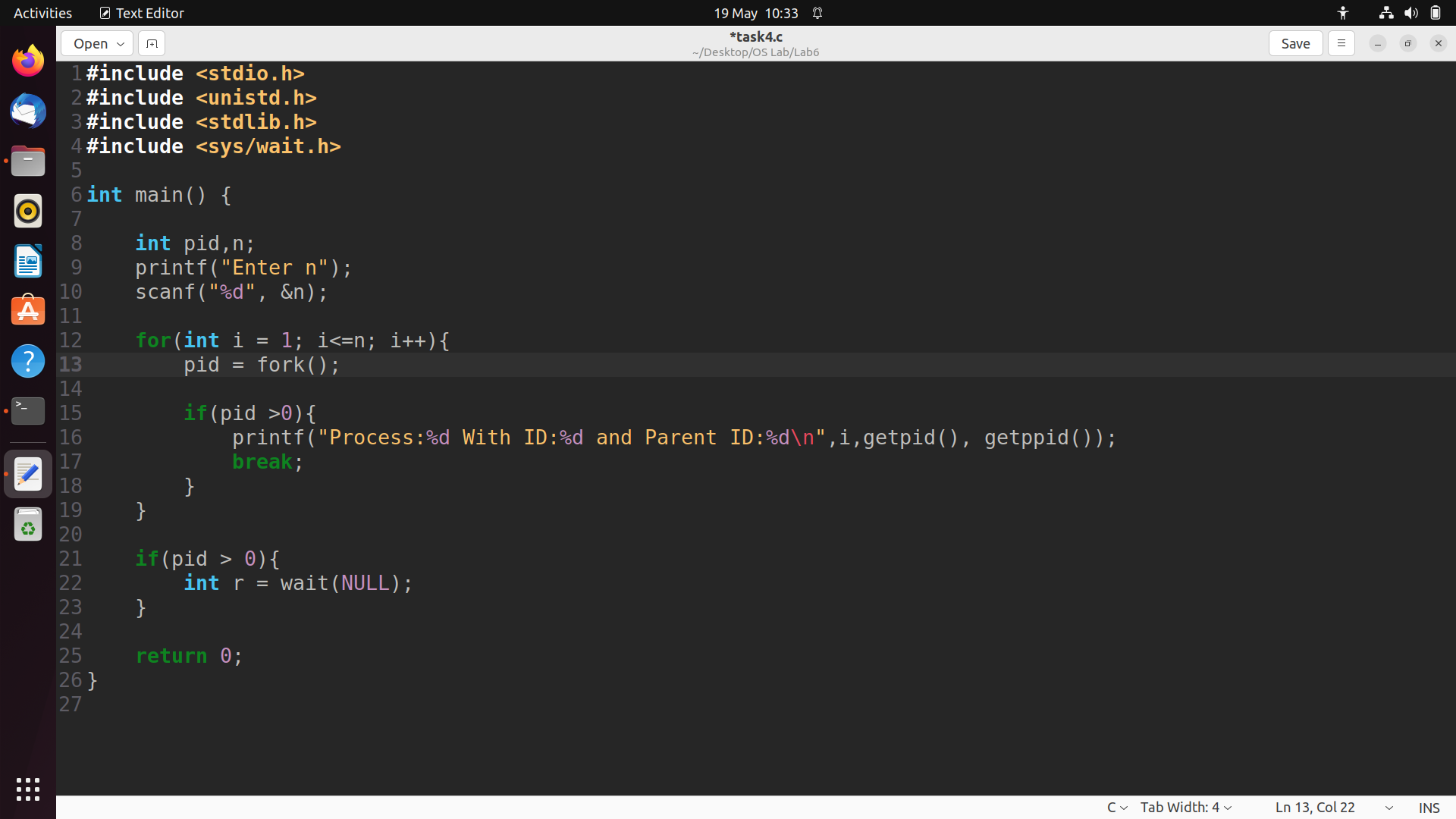
Create a fan of N processes. Take N as input from the user. Make sure there are no orphan processes.

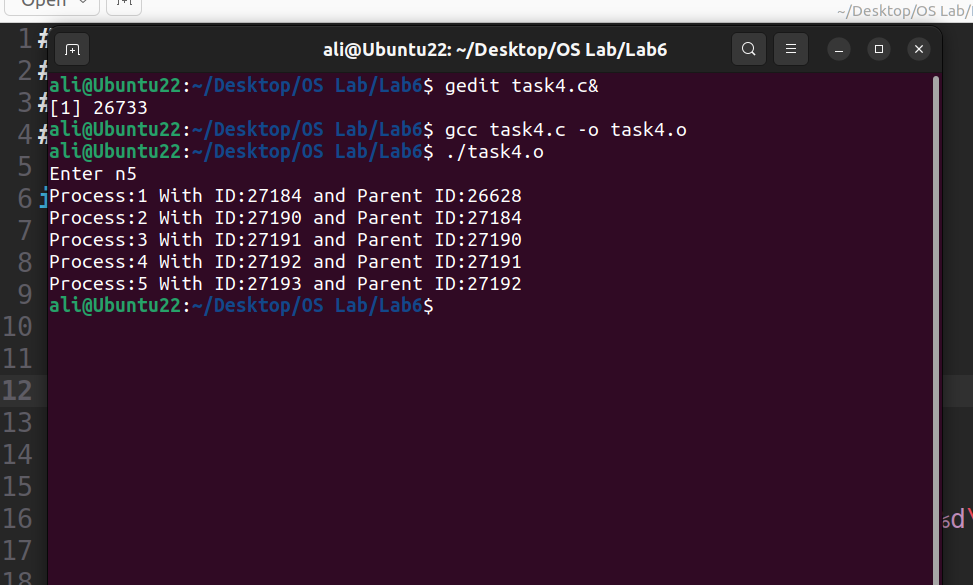


****

**Task4:**

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

****

****