

**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**







**HỆ ĐIỀU HÀNH**

**LỚP: IT007.O18**

**Tên: Lê Minh Nhựt**

**MSSV: 22521060**



**THỰC HÀNH LAB 3**

**Câu 1: Mối quan hệ cha-con giữa các tiến trình.**

1. *Vẽ cây quan hệ parent-child.*

1

1 Init

86 WindowServer

281 iTunes

282 Terminal

287 login

293 -bash

461 firefox-bin

751 Aquamacs

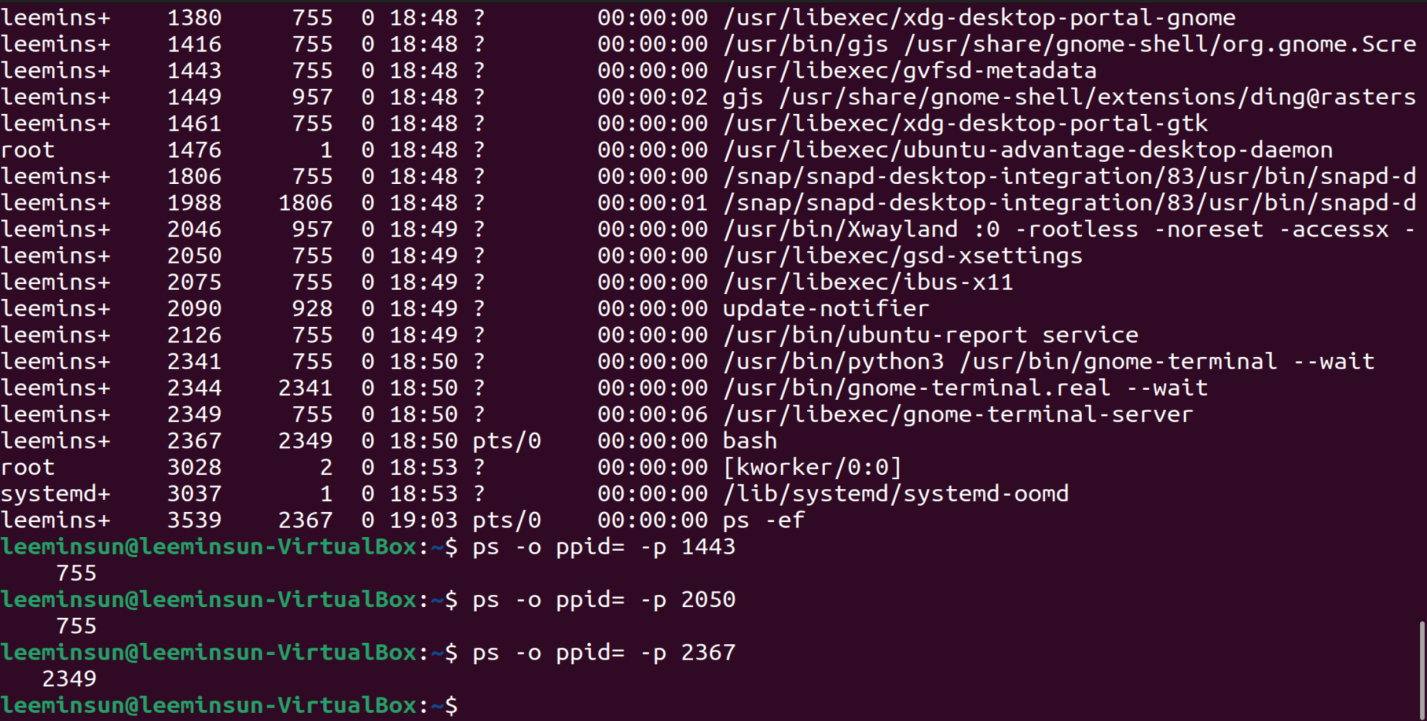
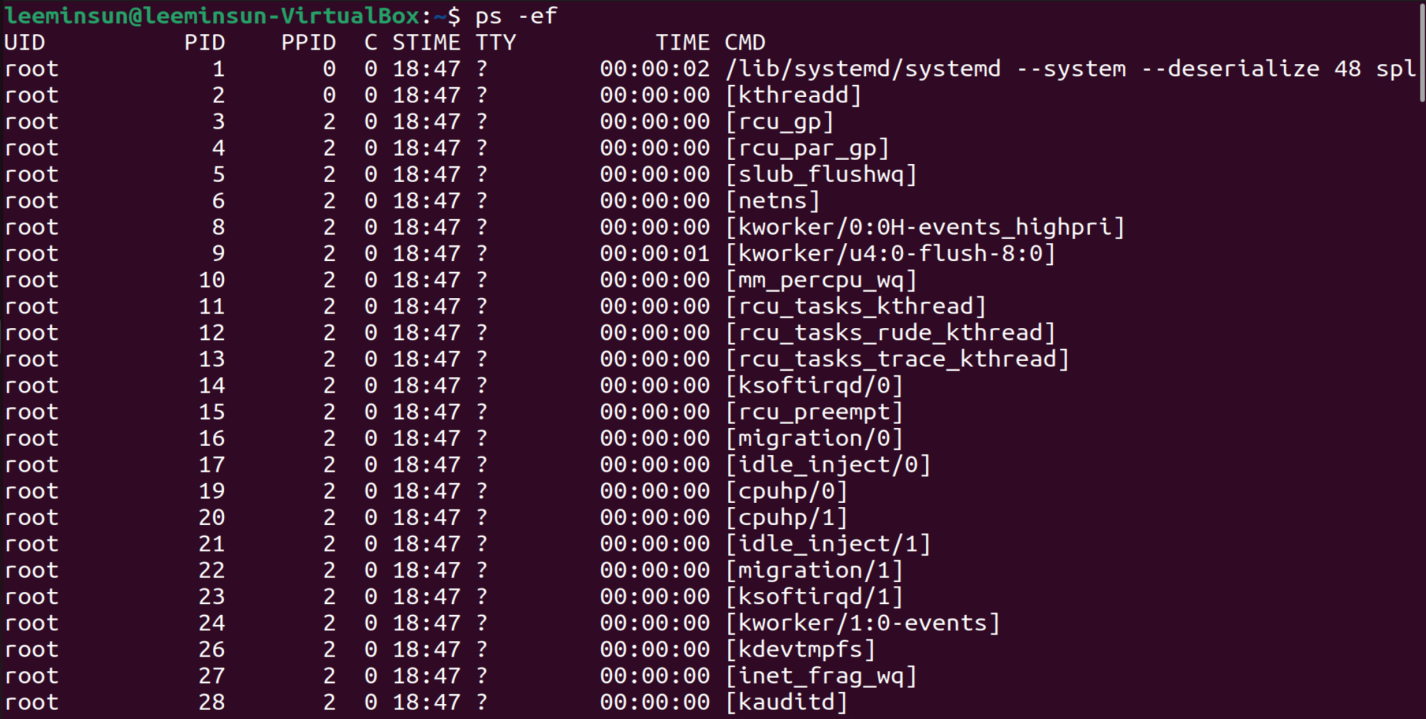
531 Safari

726 Mail

1. *Trình bày cách sử dụng lệnh ps để tìm tiến trình cha của một tiến trình dựa vào PID của nó.*

Đầu tiên ta dùng lệnh **ps -ef** để hiển thị tất cả các tiến trình đang chạy (chứa pid và các thông tin khác).

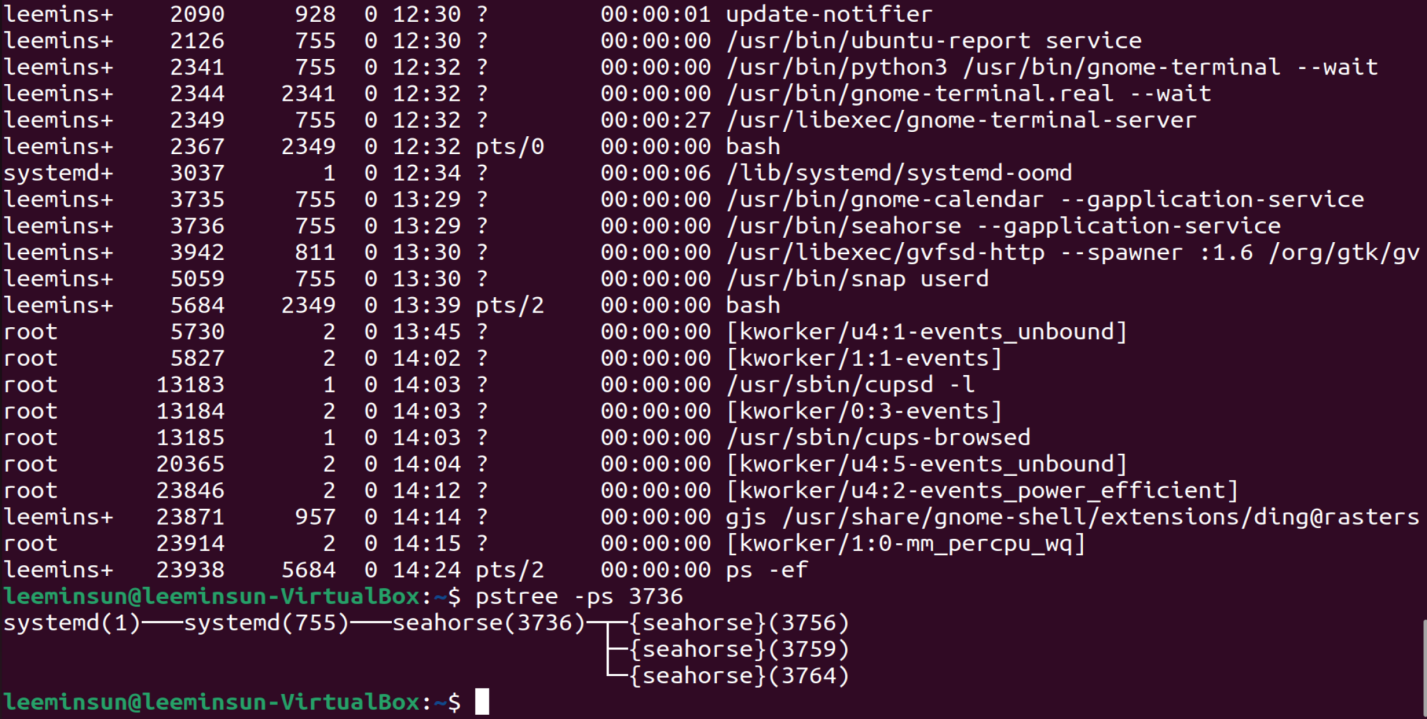
Sau đó dùng lệnh **ps -o ppid= <PID>** để tìm tiến trình cha của nó.



1. *Tìm hiểu và cài đặt lệnh pstree (nếu chưa được cài đặt), sau đó trình bày cách sử dụng lệnh này để tìm tiến trình cha của một tiến trình dựa vào PID của nó.*

Sử dụng lệnh **pstree -ps <PID>** để hiển thị các tiến trình cha và con của PID chỉ định. Trong đó **-s** cho phép xem các tiến trình có mối liên hệ với tiến trình có PID chỉ định, **-p** để hiển thị PID của các tiến trình.

Trong ảnh bên dưới các tiến trình *bên trái* là cha của tiến trình có PID = 3736.



**Câu 2:**

Đoạn chương trình trên sử dụng hàm **fork()** để tạo 1 tiến trình con:

- Trong phần tiến trình cha:

+ Biến **pid** được khai báo để lưu giá trị ID của tiến trình con khi sử dụng hàm **fork()**. Biến **num\_coconuts** được khởi tạo với giá trị 17.

+ Hàm **fork()** được gọi để tạo 1 tiến trình con. Khi gọi **fork()**, tiến trình cha sẽ tạo 1 bản sao của chính nó và trả về ID của tiến trình con cho tiến trình cha và trả về 0 cho tiến trình con.

+ Trong tiến trình cha, kiểm tra giá trị **pid**. Nếu **pid** là 0, nghĩa là đang thực thi trong tiến trình con, không phải tiến trình cha. Do đó, biến **num\_coconuts** được gán giá trị 42 và tiến trình con kết thúc với lệnh **exit(0)**.

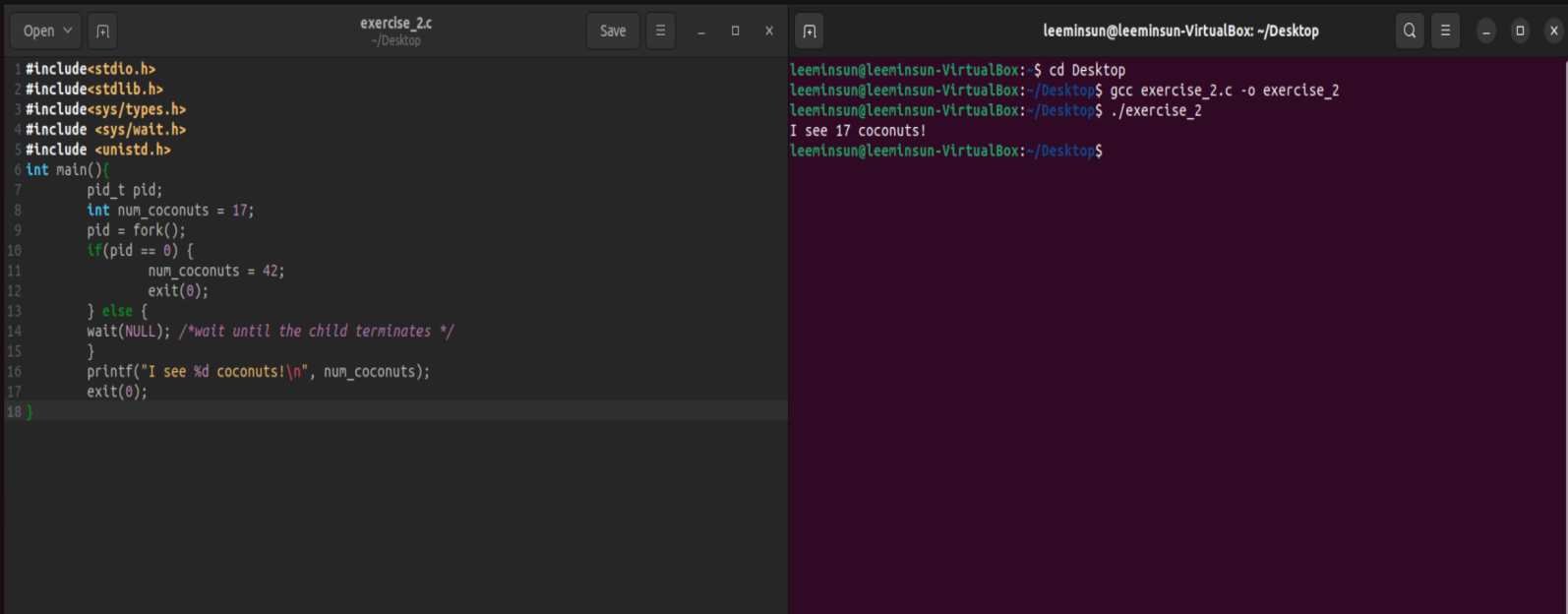
+ Trong tiến trình cha, chờ cho tiến trình con kết thúc bằng cách sử dụng hàm **wait(NULL)**. Hàm này đảm bảo rằng tiến trình cha không kết thúc trước khi tiến trình con hoàn thành.

+ Cuối cùng in ra giá trị của biến **num\_coconuts(17)** bằng câu lệnh **printf**.

- Trong phần của tiến trình con:

+ Nếu giá trị **pid** trả về từ hàm **fork()** là 0, đó là tiến trình con. Trong trường hợp này, biến **num\_coconuts** được gán giá trị 42.  
+ Tiến trình con kết thúc với lệnh **exit(0)**.

Kết quả chạy được chương tình sẽ in ra dòng **“I see 17 coconuts!”** vì khi gọi hàm **fork()**, child process sẽ sao chép toàn bộ bộ nhớ của parent process, bao gồm cả các biến và giá trị của chúng. Tuy nhiên, khi child process thay đổi giá trị của biến **num\_coconuts**, nó chỉ thay đổi trong không gian bộ nhớ của nó, không ảnh hưởng đến parent process. Do đó, parent process vẫn giữ nguyên giá trị ban đầu của biến **num\_coconuts** là 17.



**Câu 3: Trong phần thực hành, các ví dụ chỉ sử dụng thuộc tính mặc định của pthread, hãy tìm hiểu POSIX thread và trình bày tất cả các hàm được sử dụng để làm thay đổi thuộc tính của pthread, sau đó viết các chương trình minh họa tác động của các thuộc tính này và chú thích đầy đủ cách sử dụng hàm này trong chương trình. (Gợi ý các hàm liên quan đến thuộc tính của pthread đều bắt đầu bởi: pthread\_attr\_\*)**

**\*Các hàm được sử dụng để làm thay đổi thuộc tính của pthread:**

• **pthread\_attr\_init** (pthread\_attr\_t \*attr): để khởi tạo đối tượng thuộc tính luồng với các giá trị mặc định.

• **pthread\_attr\_destroy** (pthread\_attr\_t \*attr): để xóa bộ nhớ cấp phát cho đối tượng thuộc tính luồng và làm cho nó không hợp lệ.

• **pthread\_attr\_setdetachstate** (pthread\_attr\_t \*attr, int detachstate): để đặt trạng thái detach của luồng (joinable hoặc detached) cho đối tượng thuộc tính luồng.

• **pthread\_attr\_getdetachstate** (const pthread\_attr\_t \*attr, int \*detachstate): để lấy trạng thái detach của luồng từ đối tượng thuộc tính luồng.

• **pthread\_attr\_setscope** (pthread\_attr\_t \*attr, int scope): để đặt phạm vi tranh chấp của luồng (process hoặc system) cho đối tượng thuộc tính luồng.

• **pthread\_attr\_getscope** (const pthread\_attr\_t \*attr, int \*scope): để lấy phạm vi tranh chấp của luồng từ đối tượng thuộc tính luồng.

• **pthread\_attr\_setstackaddr** (pthread\_attr\_t \*attr, void \*stackaddr): để đặt địa chỉ ngăn xếp của luồng cho đối tượng thuộc tính luồng.

• **pthread\_attr\_getstackaddr** (const pthread\_attr\_t \*attr, void \*\*stackaddr): để lấy địa chỉ ngăn xếp của luồng từ đối tượng thuộc tính luồng.

• **pthread\_attr\_setstacksize** (pthread\_attr\_t \*attr, size\_t stacksize): để đặt kích thước ngăn xếp của luồng cho đối tượng thuộc tính luồng.

• **pthread\_attr\_getstacksize** (const pthread\_attr\_t \*attr, size\_t \*stacksize): để lấy kích thước ngăn xếp của luồng từ đối tượng thuộc tính luồng.

• **pthread\_attr\_setschedpolicy** (pthread\_attr\_t \*attr, int policy): để đặt chính sách lập lịch của luồng cho đối tượng thuộc tính luồng.

• **pthread\_attr\_getschedpolicy** (const pthread\_attr\_t \*restrict attr, int \*restrict policy): để lấy chính sách lập lịch của luồng từ đối tượng thuộc tính luồng.

• **pthread\_attr\_setguardsize** (pthread\_attr\_t \*attr, size\_t guardsize): để đặt kích thước bảo vệ ngăn xếp của luồng cho đối tượng thuộc tính luồng.• **pthread\_attr\_getguardsize** (const pthread\_attr\_t \*attr, size\_t \*guardsize): để lấy kích thước bảo vệ ngăn xếp của luồng từ đối tượng thuộc tính luồng.

**Một vài chương trình minh họa:**

**pthread\_attr\_init**

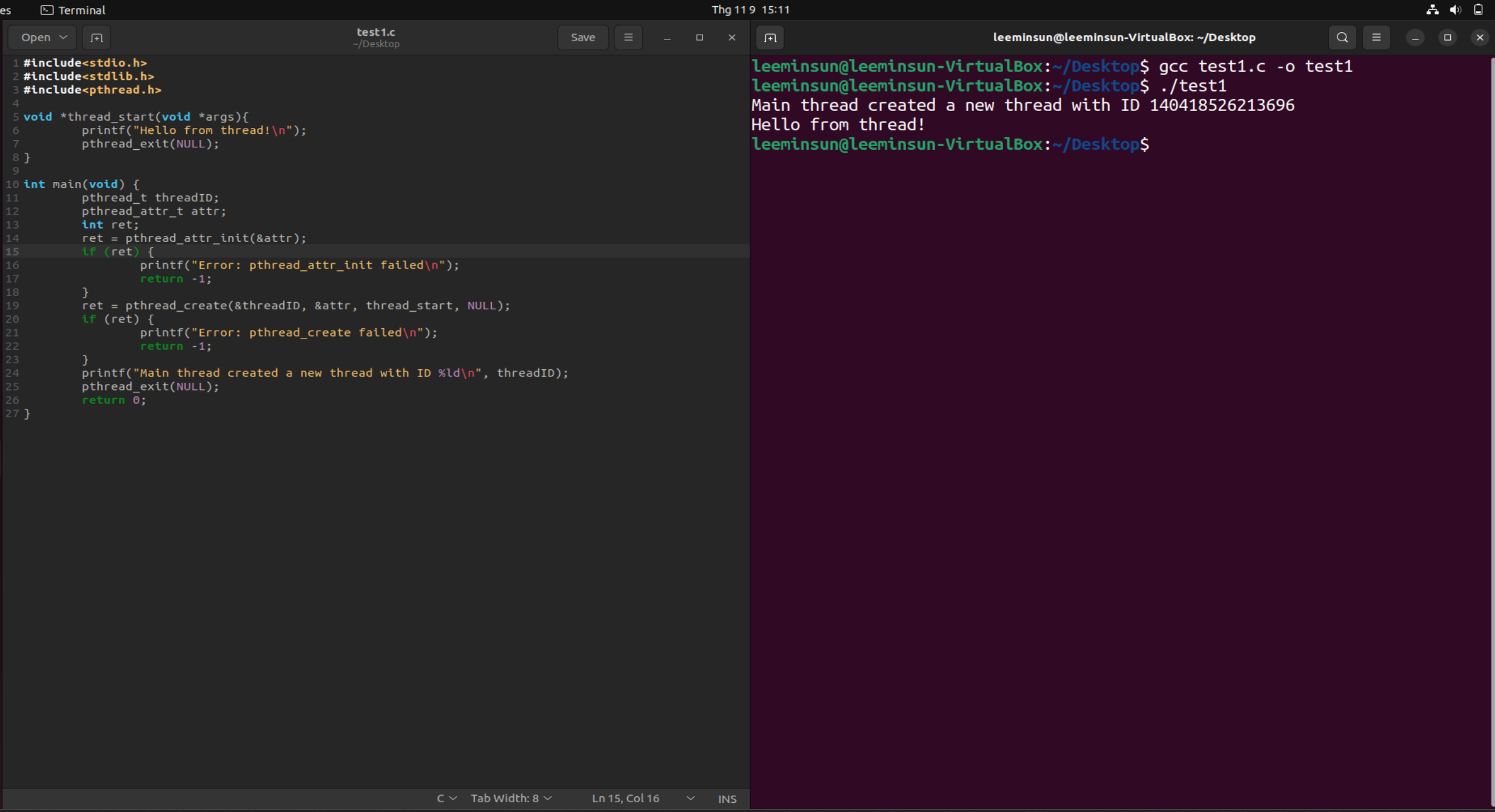
Hàm pthread\_attr\_init có prototype như sau:

**#include <pthread.h>**

**int pthread\_attr\_init(pthread\_attr\_t \*attr);**

Trong đó, attr là con trỏ trỏ đến đối tượng thuộc tính luồng cần khởi tạo. Hàm này trả về 0 nếu thành công, hoặc một số dướng là mã lỗi nếu có lỗi.

**Code và kết quả:**



**Công dụng trong chương trình**: Tác dụng của hàm **pthread\_attr\_init** trong ví dụ trên là để khởi tạo một đối tượng thuộc tính luồng với các giá trị mặc định cho các thuộc tính như trạng thái detach, kích thước ngăn xếp, chính sách lập lịch, v.v. Điều này có nghĩa là khi tạo luồng mới với đối tượng thuộc tính luồng này, luồng sẽ có các thuộc tính theo giá trị mặc định của hệ thống. Đây là một cách để thiết lập các thuộc tính cho các luồng mới một cách nhanh chóng và tiện lợi.

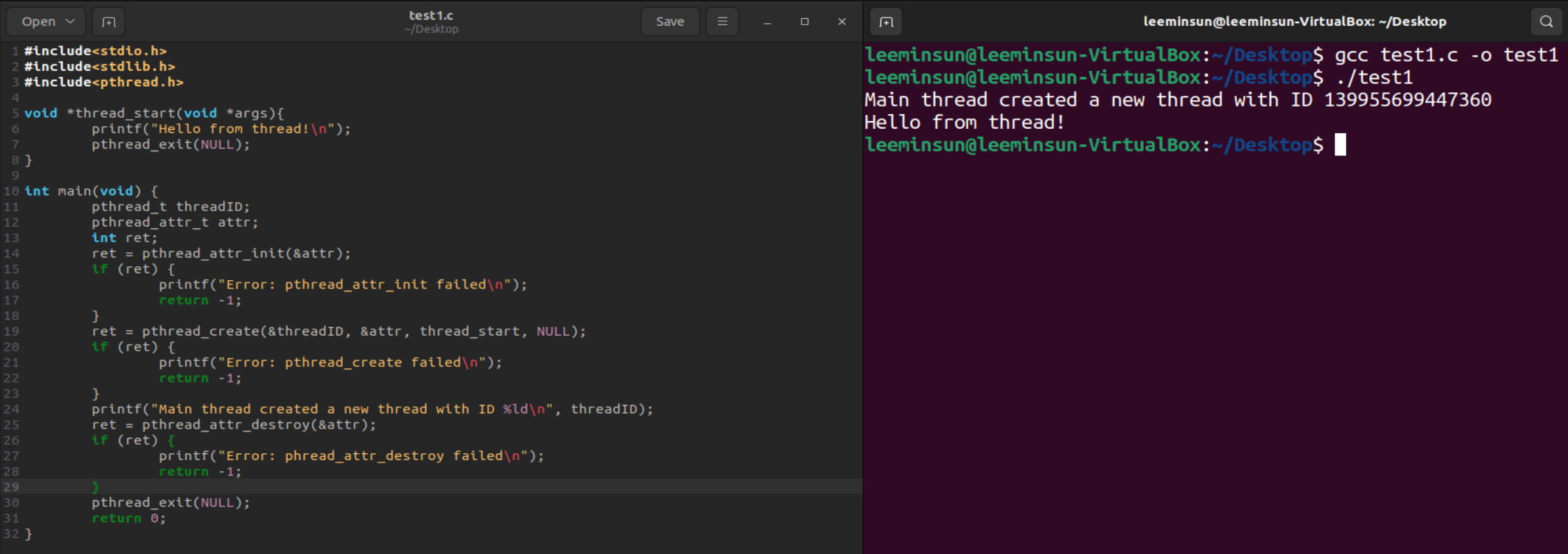
**pthread\_attr\_destroy**

Hàm **pthread\_attr\_destroy** có prototype như sau:

**#include <pthread.h>   
int pthread\_attr\_destroy(pthread\_attr\_t \*attr);**

Trong đó, **attr** là con trỏ trỏ đến đối tượng thuộc tính luồng cần xóa. Hàm này trả về 0 nếu thành công, hoặc một số dướng là mã lỗi nếu có lỗi.

**Code và kết quả:**

Cl

**Công dụng trong chương trình:** Tác dụng của hàm **pthread\_attr\_destroy** trong chướng trình trên là để xóa bộ nhớ cấp phát cho đối tượng thuộc tính luồng và làm cho nó không hợp lệ. Điều này có nghĩa là đối tượng thuộc tính luồng này không thể được sử dụng để tạo các luồng mới. Đây là một cách để tránh rò rỉ bộ nhớ và giảm thiểu lãng phí tài nguyên của tiến trình.

**Câu 4:**

\*Code:

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

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

**#include <signal.h>**

**void signalHandle(int signal){**

**printf("\nYou are pressed CTRL+C! Goodbye!\n");**

**exit(0);**

**}**

**int main(){**

**//a. In ra dòng chữ: “Welcome to IT007, I am <your\_Student\_ID>!”**

**printf("Welcome to IT007, I am 22521060!\n");**

**//b. Mở tệp abcd.txt bằng vim editor**

**pid\_t pid = fork();**

**if (pid == -1) {**

**printf("Error creating child process.\n");**

**exit(1);**

**}**

**if (pid == 0) {**

**system("gedit abcd.txt");**

**exit(1);**

**}**

**//c +d. Tắt vim editor khi người dùng nhấn CRTL+C**

**signal(SIGINT, signalHandle);**

**// Chờ đợi người dùng nhấn CTRL+C**

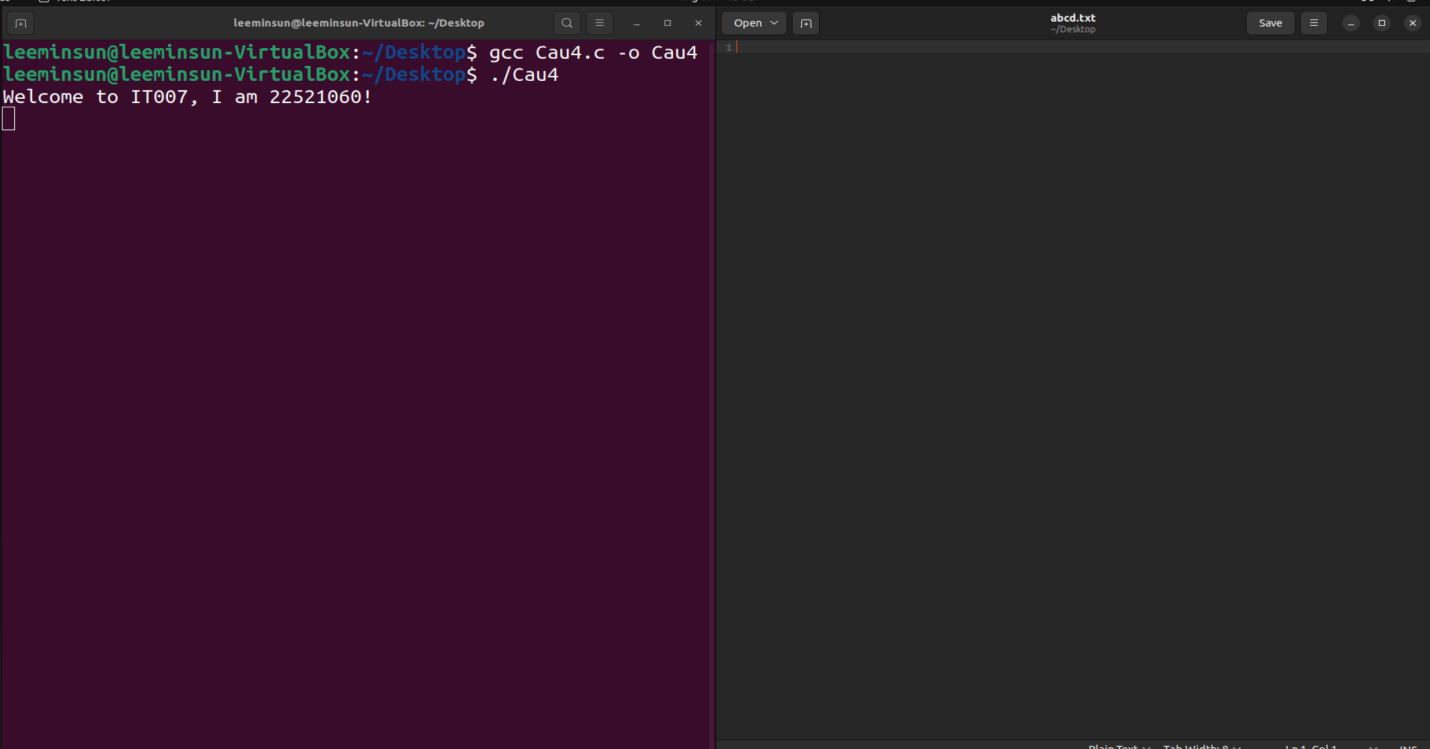
**while(1) {**

**sleep(1);**

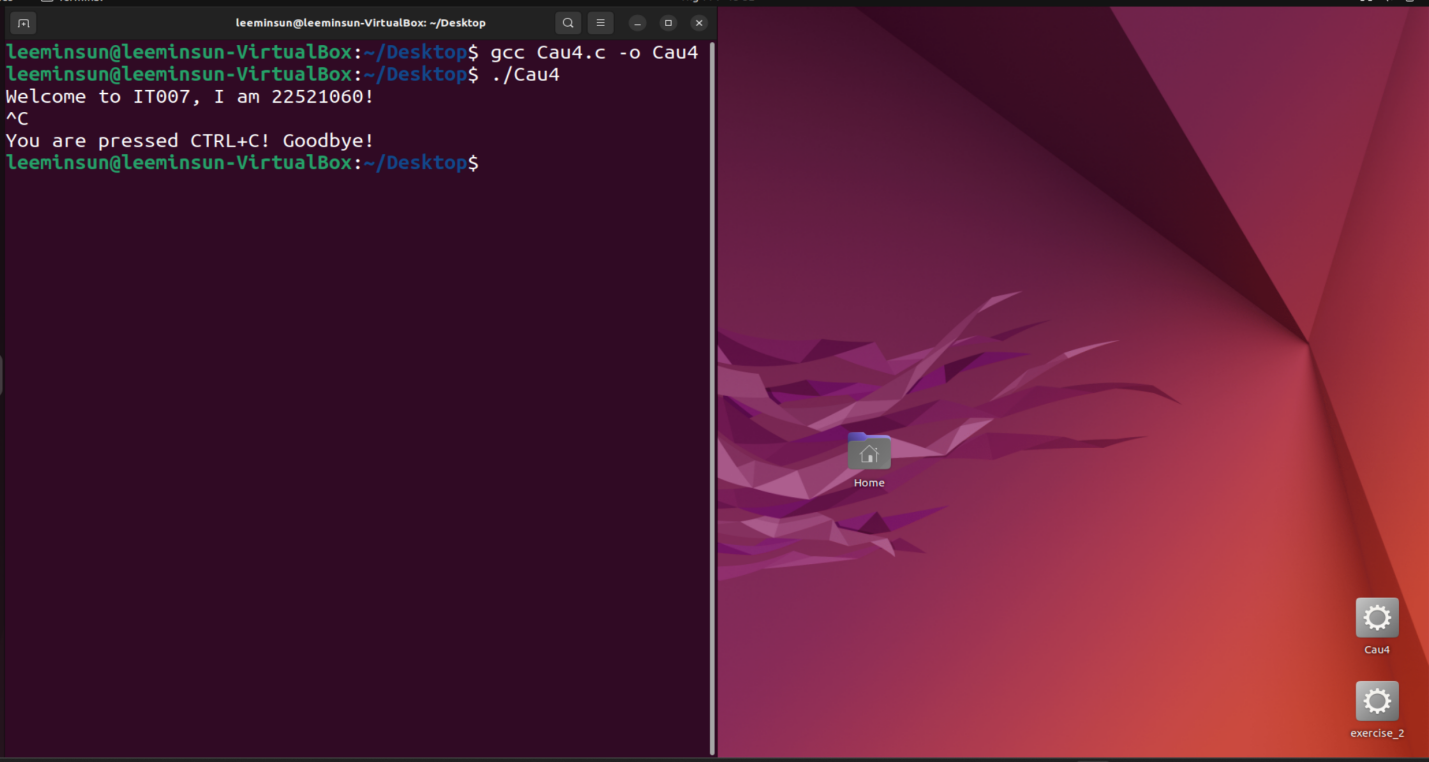
**}**

**return 0;**

**}**



*Ảnh 4.1: In ra màn hình dòng chữ “Welcome to IT007, I am <your\_Student\_ID>!” và mở tệp abcd.txt bằng vim editor.*



*Ảnh 4.2: Tắt vim editor khi nhấn Ctrl + C và hiện dòng chữ “You are presse CTRL+C! Goodbye!”*