Name: Nguyễn Khoa Nguyên

ID: 24521190

Class: IT007.Q15.1

OPERATING SYSTEM

LAB 3'S REPORT

SUMMARY

Task		Status	Page
Section 3.5	Ex1	Done	2,3,4,5,6,7,8,9,10
	Ex2	Done	11,12
	Ex3	Done	12,13,14
	Ex4	Done	15,16,17

Self-scrores:

Note: Export file to **PDF and name the file by following format:*

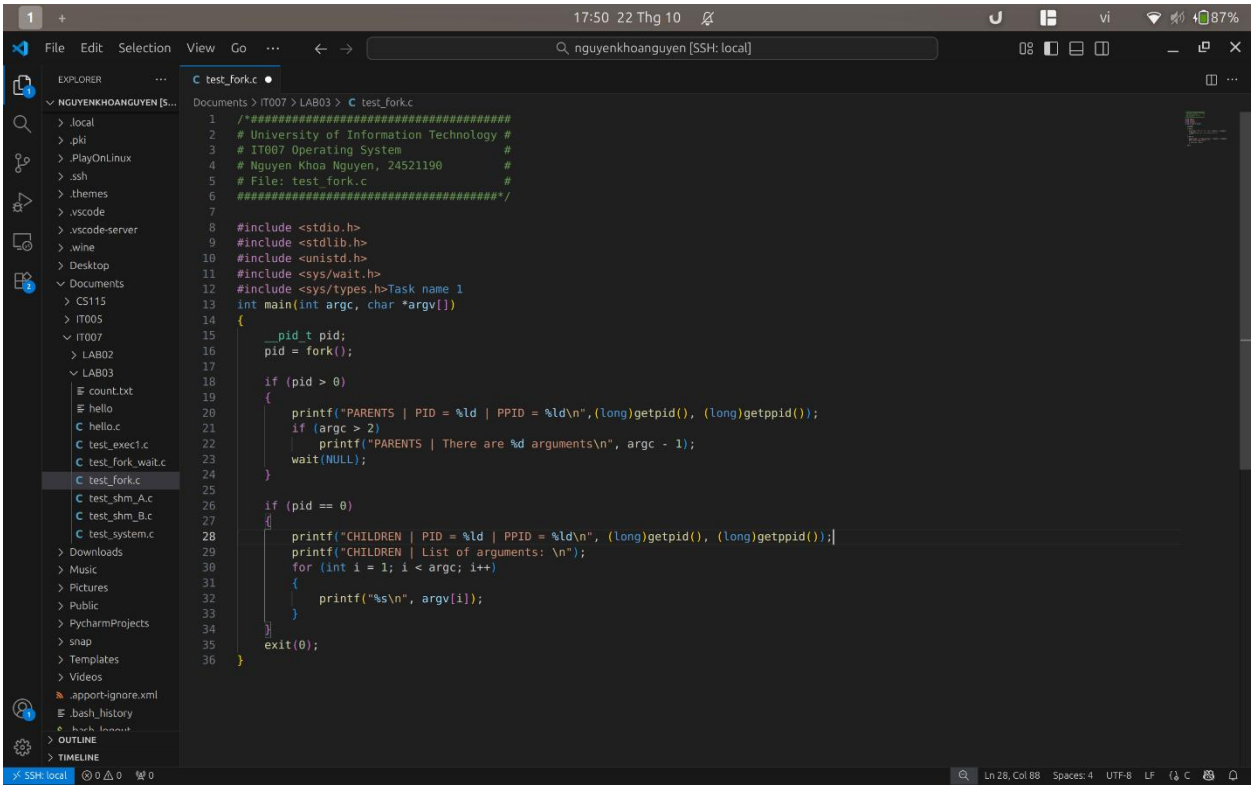
Student ID_LABx.pdf

Section 3.5

1. Thực hiện ví dụ 3-1, Ví dụ 3-2, Ví dụ 3-3, Ví dụ 3-4 giải thích code và kết quả nhận được ?

- Ví dụ 3-1:

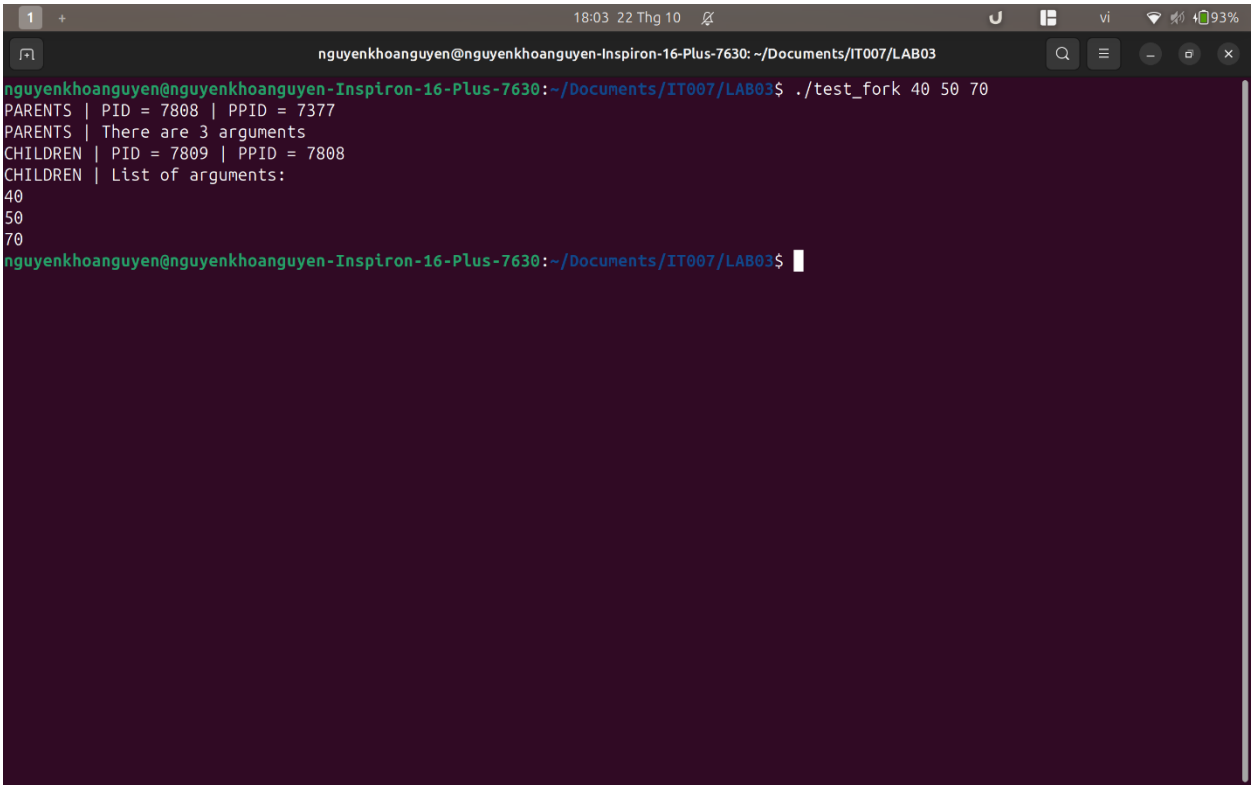
- Có thể tạo tiến trình bằng hàm fork (). Hàm fork() tạo ra một tiến trình mới bằng cách gọi một bản sao của nó. Tiến trình gọi hàm fork() được gọi là tiến trình cha, tiến trình mới được tạo ra là tiến trình con. Tiến trình cha quay lại việc thực thi và tiến trình con bắt đầu thực thi tại cùng một nơi mà fork() trả về. Nếu kết quả của hàm fork() là 0 thì ta đang ở trong tiến trình con, ngược lại nếu kết quả trả về lớn hơn 0 thì ta đang trong tiến trình cha và kết quả trả về chính là PID của tiến trình con, và nếu trả về -1 thì hàm fork() không thực thi được. Tiến trình cha và con được phân biệt bởi PID của chúng, PID của tiến trình cha sẽ được gán cho PPID của tiến trình con.



Hình 1 Nội dung code file test_fork.c

- __pid_t pid và pid = fork(): chương trình khai báo một biến pid kiểu __pid_t (kiểu dữ liệu đặc biệt dùng để lưu trữ ID của tiến trình), sau đó gọi hàm fork() tại tiến trình vừa được khai báo. Lúc này hàm fork() sẽ tạo ra một tiến trình con mới chính là bản sao của tiến trình hiện tại

- if (pid >0): Nếu pid lớn hơn 0 thì đây là tiến trình cha, và trong phần điều kiện này chương trình sẽ tiến hành in ra ID của tiến trình cha. Nếu có nhiều hơn 2 đối số (argc > 2) được truyền vào chương trình thì chương trình cũng in ra số lượng đối số. Và hàm wait(NULL) được gọi để tiến trình cha chờ tiến trình con thực hiện xong sau đó mới tiếp tục.
- if (pid == 0): Nếu pid bằng 0 thì đây là tiến trình con. Trong phần điều kiện này thì chương trình sẽ in ra ID của tiến trình con và cũng in ra các đối số được truyền vào chương trình.
- exit(0): Chương trình kết thúc với mã trạng thái 0 cho biết chương trình đã được thực thi thành công
- Kết quả thực thi code:



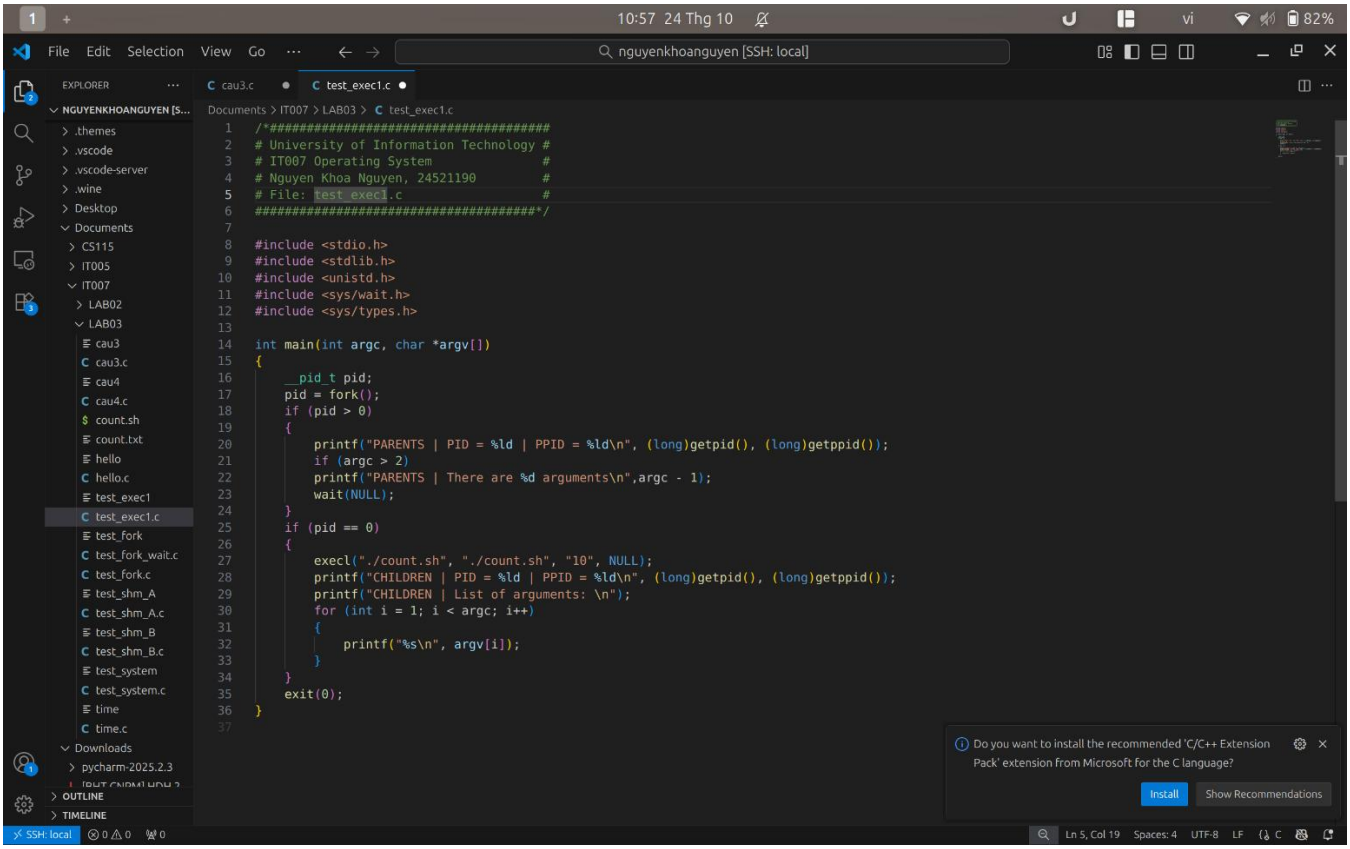
Hình 2 Kết quả thực thi file test_fork.c

- PARENTS | PID = 7808 | PPID = 7377: Dòng này in ra thông tin về tiến trình cha. PID = 7808 chính là ID của tiến trình cha và PPID = 7377 chính là ID tiến trình cha của tiến trình cha
- PARENTS | There are 3 arguments: Dòng này in ra số lượng đối số được truyền vào chương trình từ dòng lệnh. Trong trường hợp này, có 3 đối số là 50 70 100
- CHILDREN | PID = 7809 | PPID = 7808: Dòng này in ra thông tin về tiến trình con, PID = 7809 chính là ID của tiến trình con và PPID = 7808 chính là ID tiến trình cha của tiến trình con này

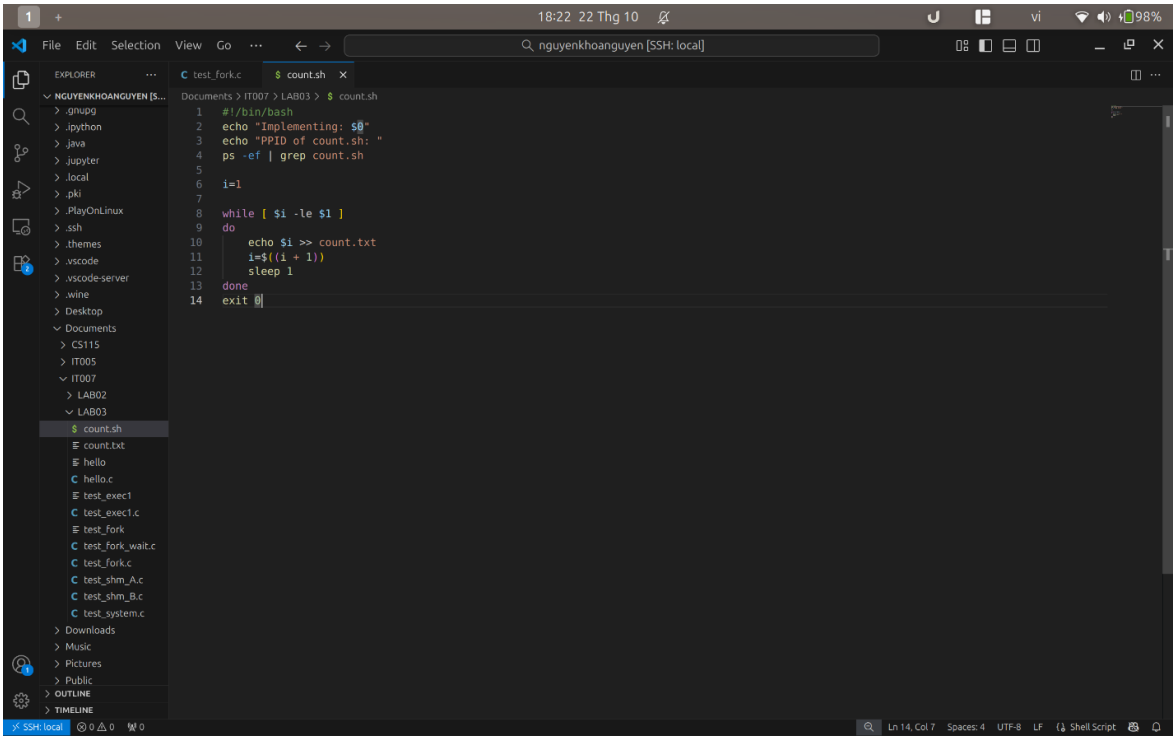
- CHILDREN | List of arguments: 40 50 70: Dòng này in ra danh sách các đối số được truyền vào chương trình

- Ví dụ 3-2:

- Một cách khác để tạo tiến trình là sử dụng họ hàm exec(), họ hàm exec() được sử dụng để thay thế tiến trình hiện tại bằng cách nạp chương trình được chỉ định tới địa chỉ của nó, sau đó tiến trình gọi hàm exec sẽ tự hủy. Nghĩa là số lượng tiến trình sẽ giữ nguyên, nhưng tiến trình mới sẽ thay thế tiến trình cũ tại địa chỉ đã cấp phát và PID không đổi.

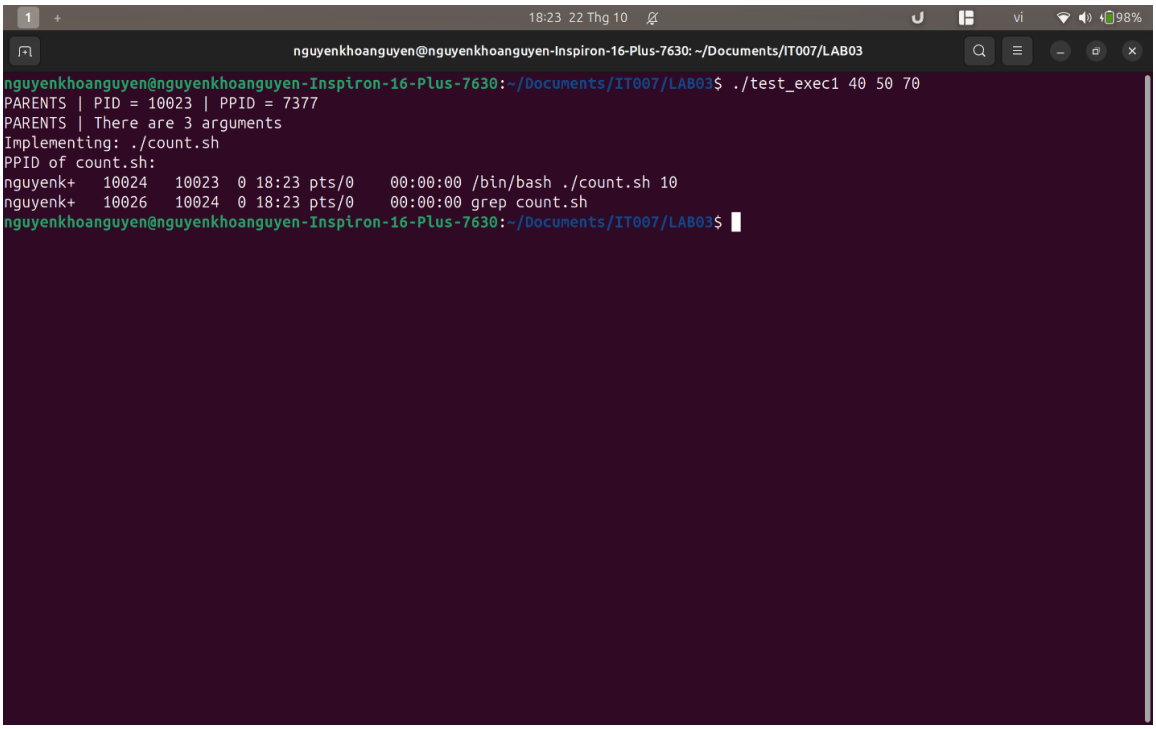


Hình 3 Nội dung code file test_exec1.c



Hình 4 Nội dung file count.sh

- Cách hoạt động tương tự như ví dụ trước đó. Trước khi in ra danh sách truyền đối số vào thì tiến trình con thực thi tập lệnh count.sh bằng hàm execl
- Kết quả

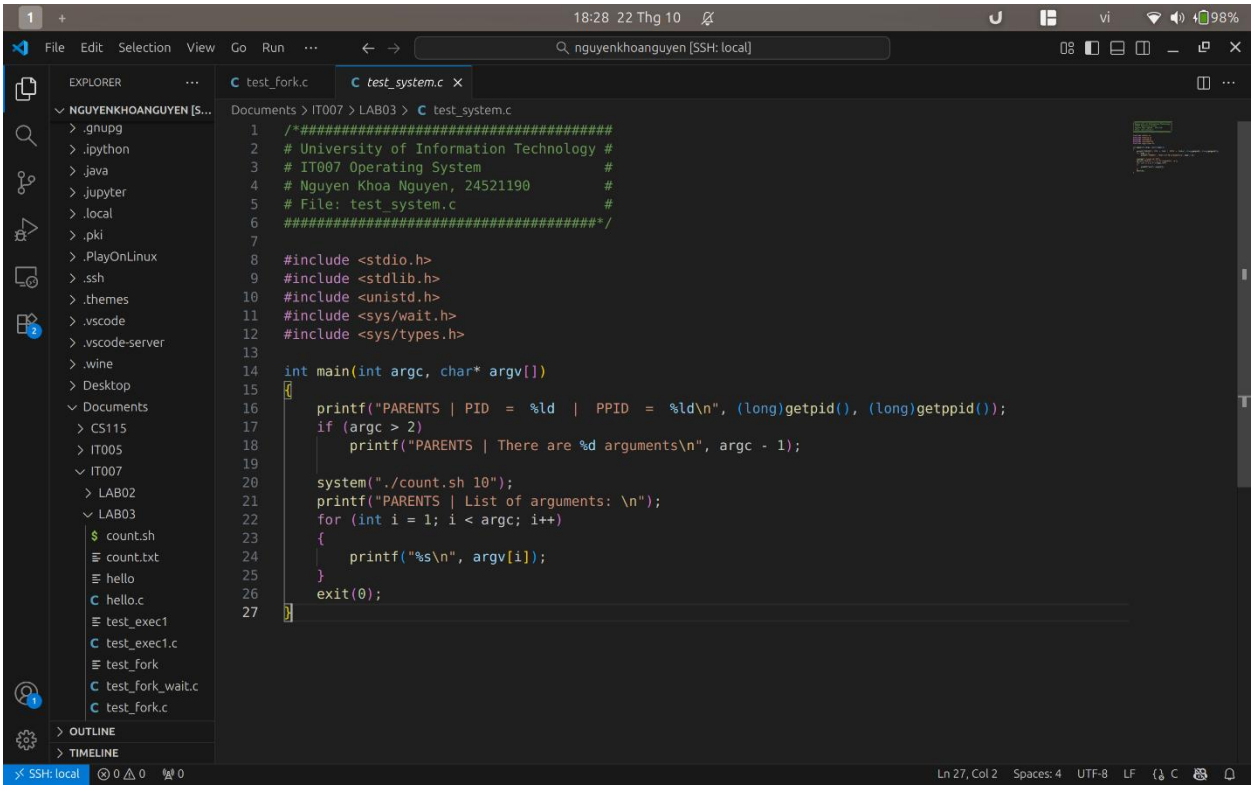


Hình 5 Kết quả thực thi file test_ecec1.c

- 10024: PID của tiến trình đang chạy file count.sh
- 10023: PPID = 10023 đúng với PID của process cha (test_exec1). Điều này xác nhận rằng process con đã được tạo và thực thi count.sh thành công

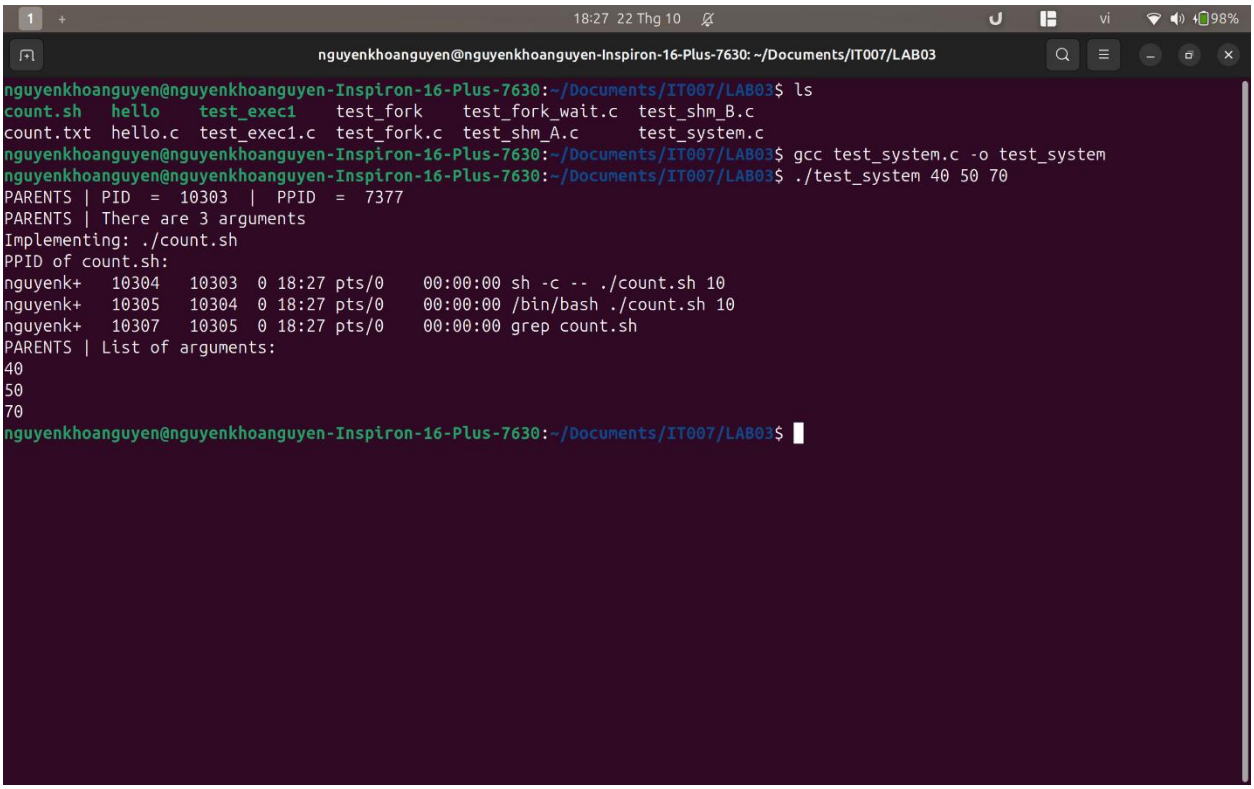
- Ví dụ 3-3:

- Cách hoạt động tương tự như với ví dụ 3-2 khi hàm system() giúp tiến trình con thực thi tập lệnh count.sh trước khi in ra danh sách đối số như lệnh execl



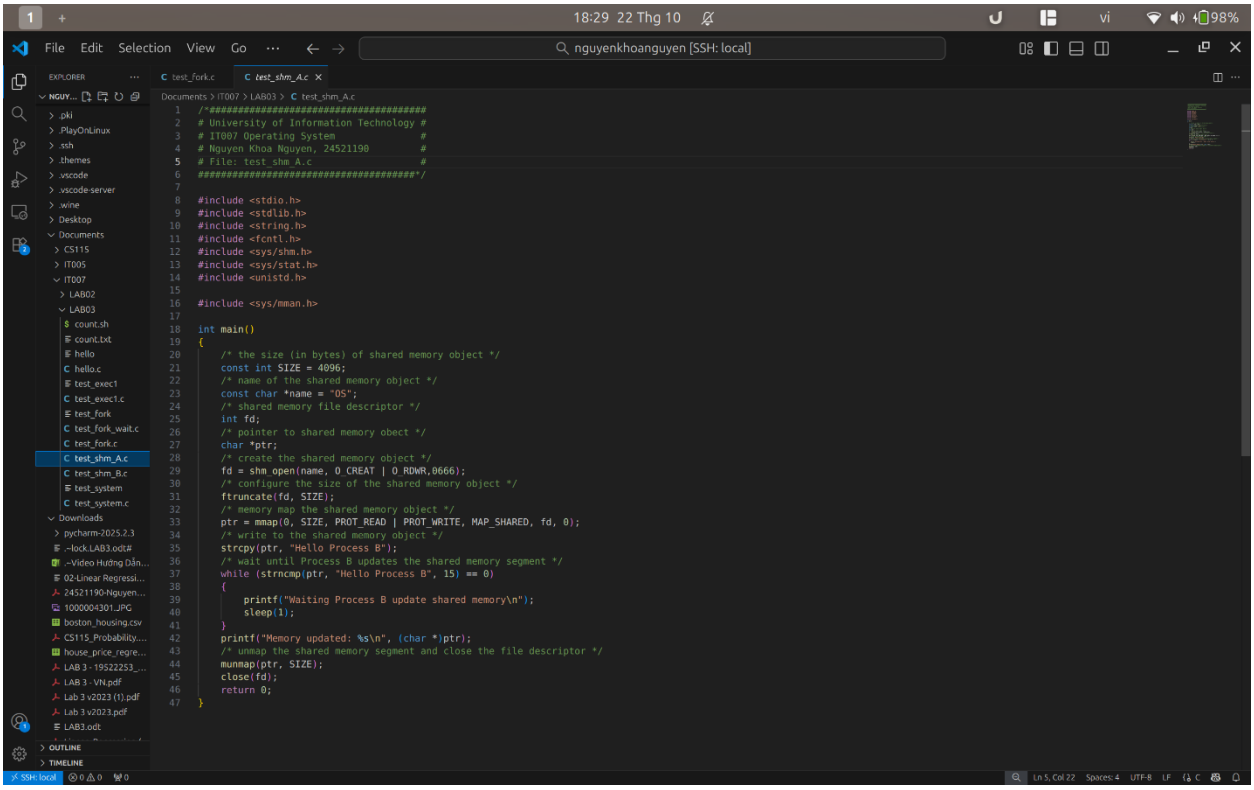

```c
/*###################################
# University of Information Technology #
# IT007 Operating System               #
# Nguyen Khoa Nguyen, 24521190         #
# File: test_system.c                  #
###################################*/

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

int main(int argc, char* argv[])
{
    printf("PARENTS | PID  =  %ld  |  PPID  =  %ld\n", (long)getpid(), (long)getppid());
    if (argc > 2)
        printf("PARENTS | There are %d arguments\n", argc - 1);

    system("./count.sh 10");
    printf("PARENTS | List of arguments: \n");
    for (int i = 1; i < argc; i++)
    {
        printf("%s\n", argv[i]);
    }
    exit(0);
}
```

Hình 6 Nội dung file test_system.c



Hình 7 Kết quả thực thi file test_system.c

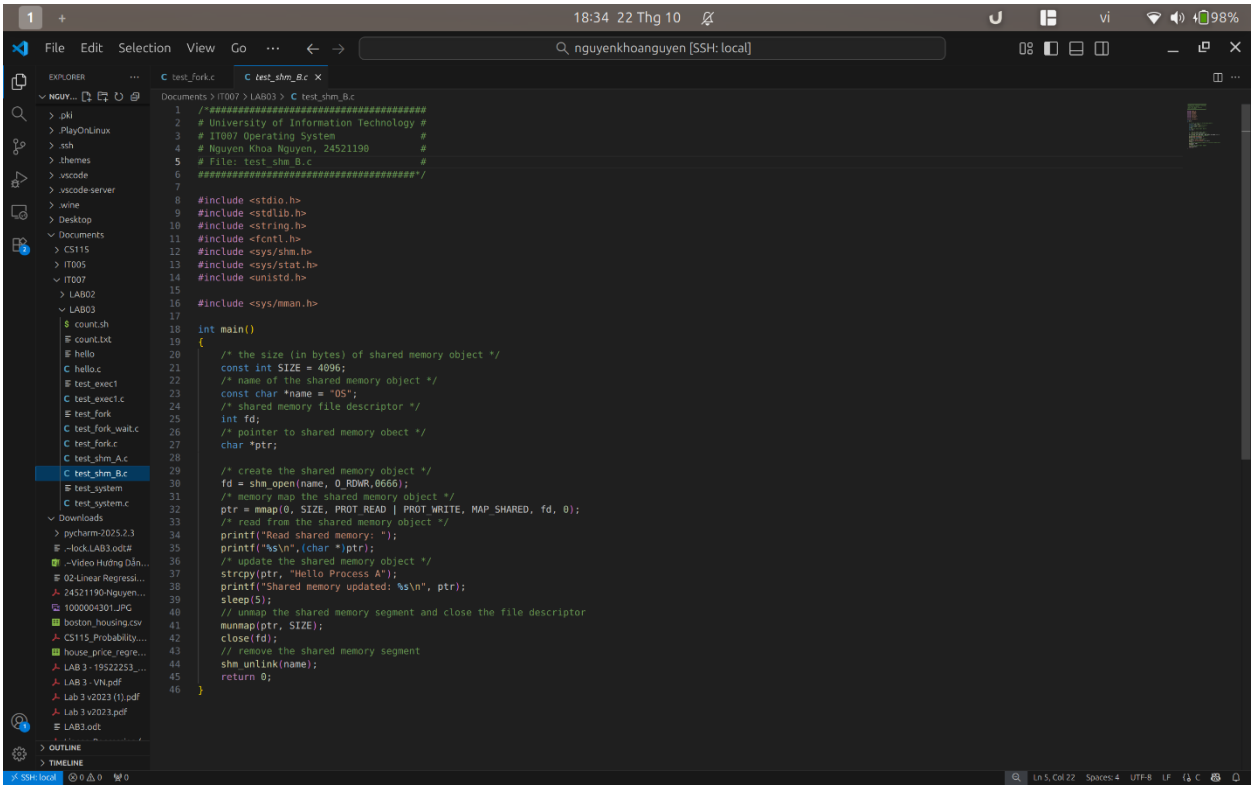
- Ví dụ 3-4:



Hình 8 Nội dung file test_shm_A.c

- Giải thích code:

- fd = shm_open(name, O_CREAT | O_RDWR, 0666): Tạo một đối tượng bộ nhớ chia sẻ mới với tên là "OS". Nếu đối tượng bộ nhớ chia sẻ đã tồn tại nó sẽ được mở
- ftruncate(fd, SIZE): Đặt kích thước của đối tượng bộ nhớ chia sẻ thành SIZE (4096 byte)
- ptr = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0): Ánh xạ đối tượng bộ nhớ chia sẻ vào không gian địa chỉ của tiến trình. Kết quả của hàm mmap() là một con trỏ trỏ đến đầu của đối tượng bộ nhớ chia sẻ
- strcpy(ptr, "Hello Process B"): Ghi chuỗi "Hello Process B" vào đối tượng bộ nhớ chia sẻ
- while (strncmp(ptr, "Hello Process B", 15) == 0): Chờ đến khi chuỗi trong đối tượng bộ nhớ chia sẻ thay đổi
- printf("Memory updated: %s\n", (char*)ptr): in ra chuỗi mới từ đối tượng bộ nhớ chia sẻ
- munmap(prt,SIZE) và close(fd): Hủy ánh xạ bộ nhớ và đóng file descriptor của đối tượng bộ nhớ chia sẻ



Hình 9 Nội dung file test_shm_B.c

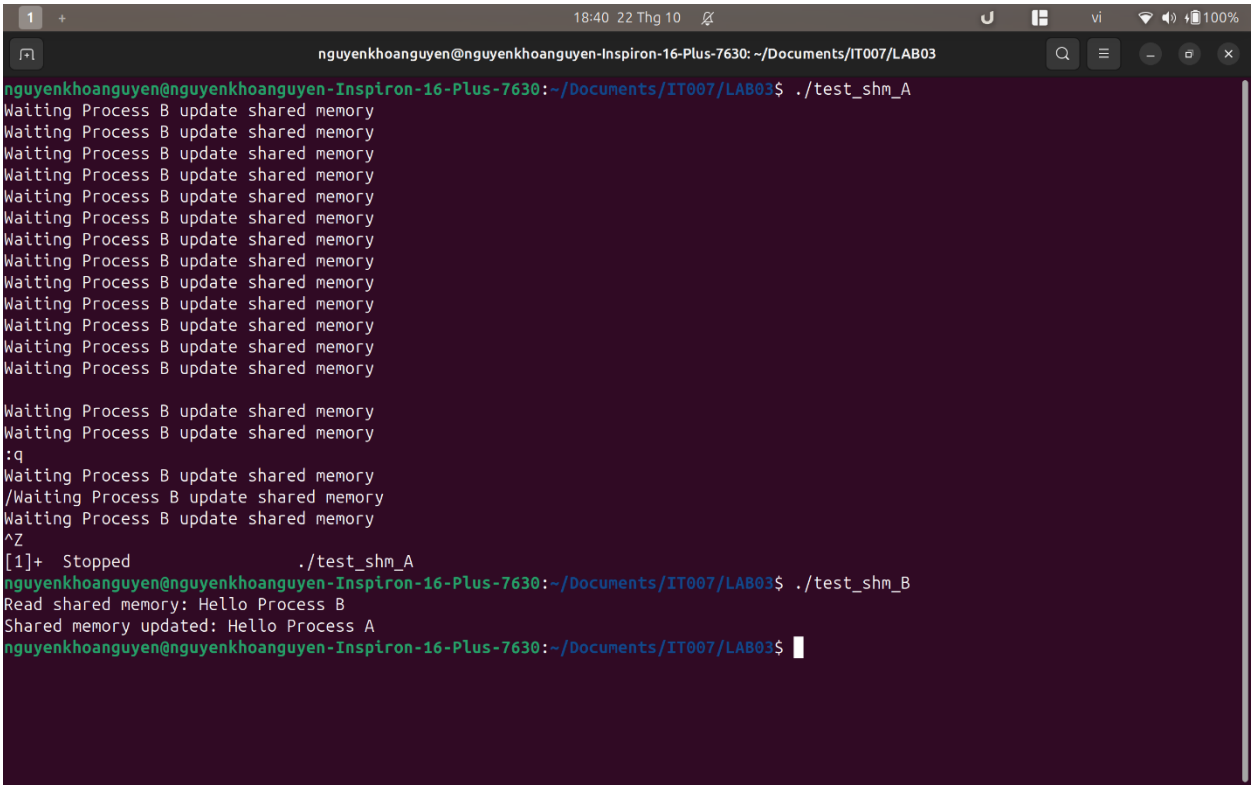
- Giải thích code:

- fd = shm_open(name, O_RDWR, 0666): mở một đối tượng bộ nhớ chia sẻ hiện có với tên là "OS" đã được tạo trước đó

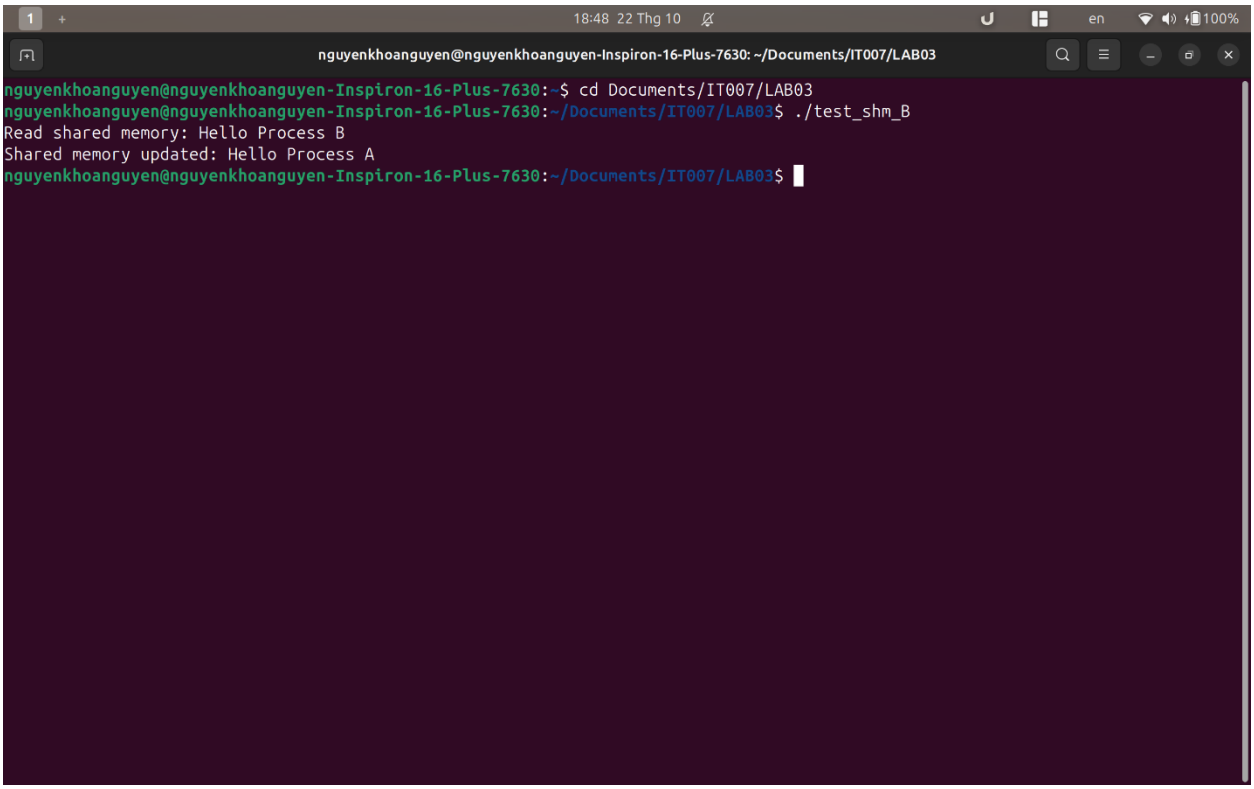
- ptr = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0): Ánh xạ đối tượng bộ nhớ chia sẻ vào không gian địa chỉ của tiến trình.. Kết quả của hàm mmap() là một con trỏ trỏ đến đầu của đối tượng bộ nhớ chia sẻ
- printf("%s\n", (char *)ptr): Đọc và in ra chuỗi từ bộ nhớ chia sẻ
- strcpy(ptr, "Hello Process A"): cập nhật chuỗi trong đối tượng bộ nhớ chia sẻ thành "Hello Process A"
- munmap(ptr, SIZE) và close (fd): hủy ánh xạ bộ nhớ và đóng file descriptor của đối tượng bộ nhớ chia sẻ
- shm_unlink(name): Xóa đối tượng bộ nhớ chia sẻ

- Kết quả:

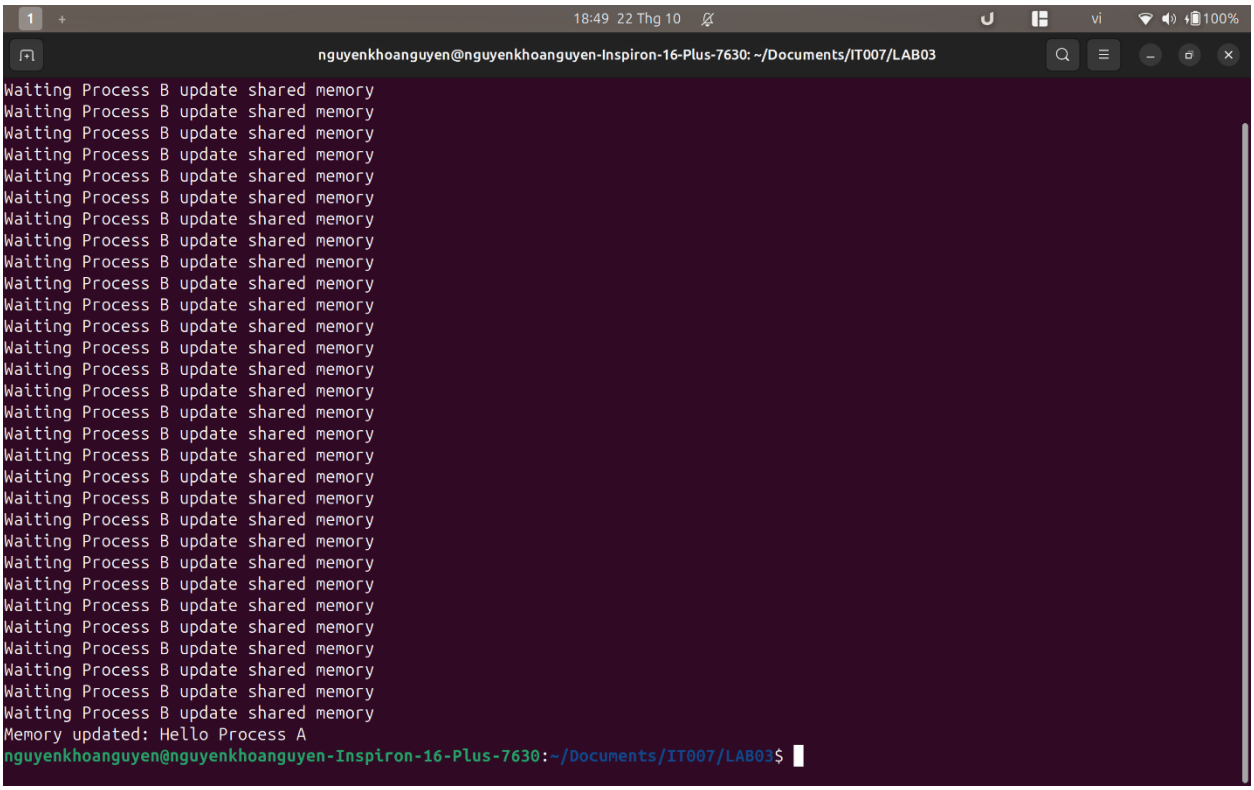
- Thực thi ./test_shm_A



Hình 10 Kết quả thực thi file test_shm_A.c

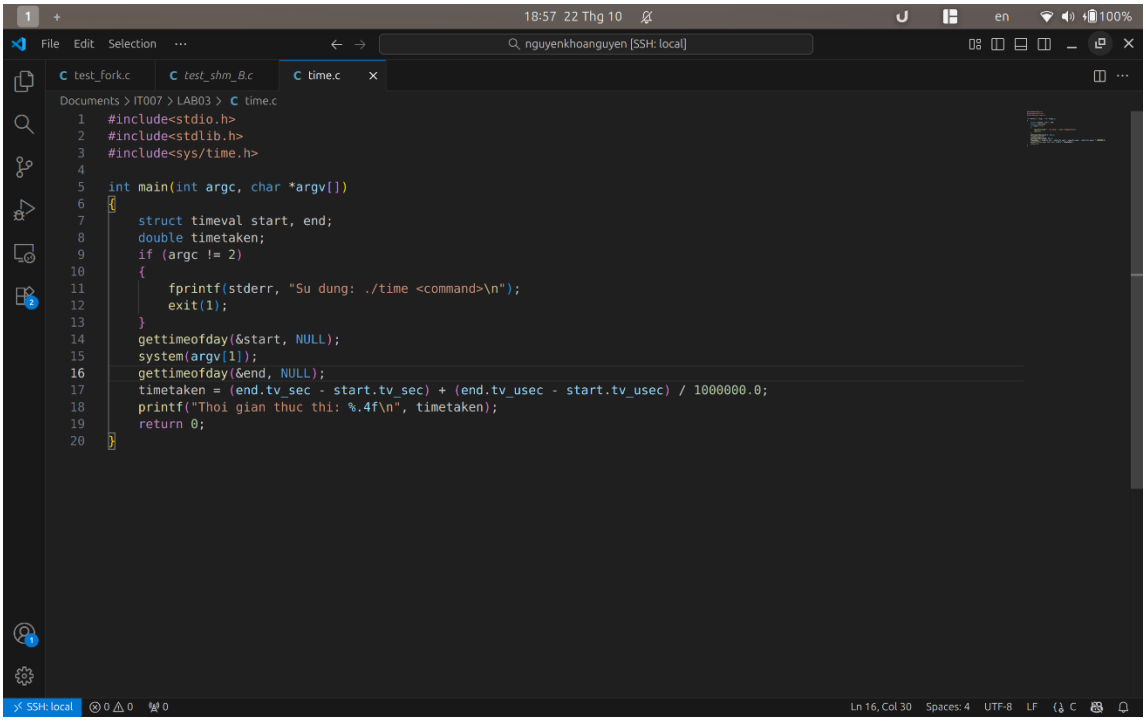


Hình 11 Kết quả thực thi file test_shm_B.c



Hình 12 Kết quả thực thi test_shm_A sau khi thực thi xong test_shm_B

2. Viết chương trình time.c thực hiện đo thời gian thực thi của một lệnh shell. Chương trình sẽ được chạy với cú pháp ".\time <command>" với <command> là lệnh shell muốn đo thời gian thực thi



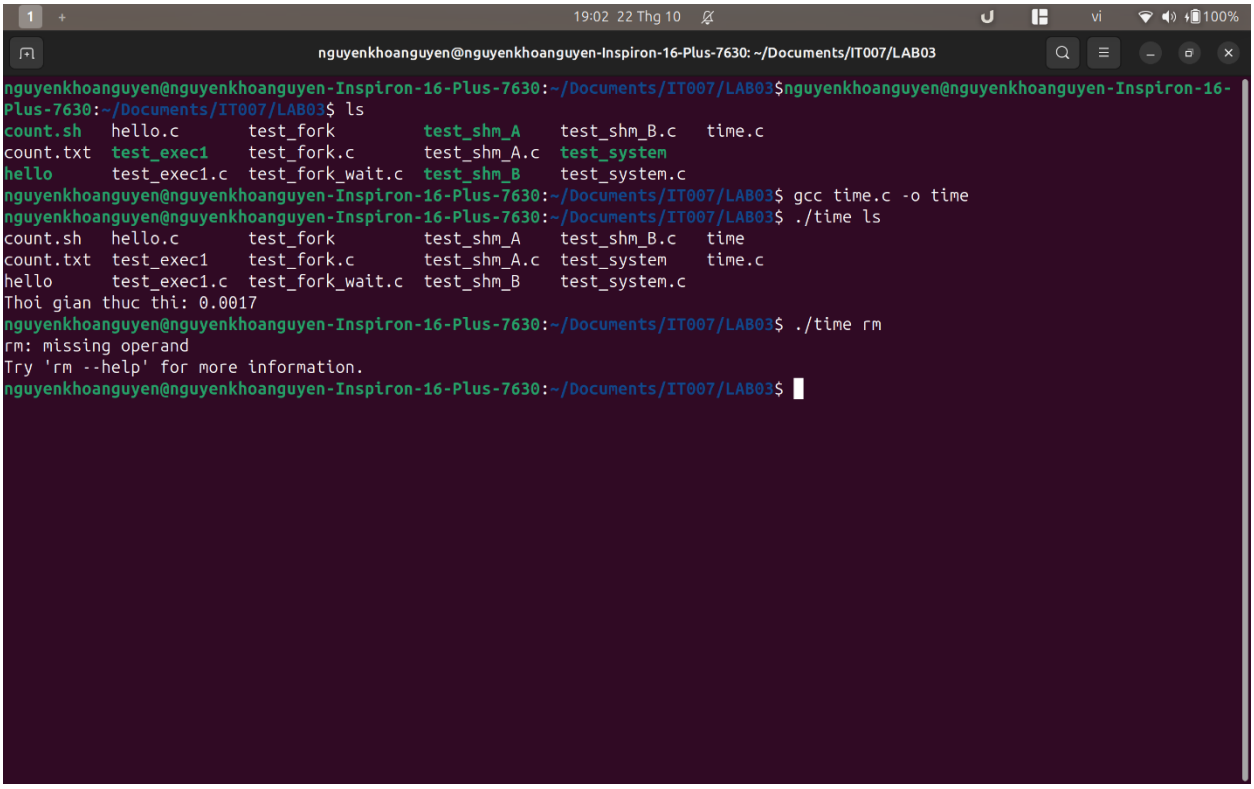
Hình 13 Nội dung code file time.c

- Giải thích code:

- sys/time.h: cung cấp cấu trúc timeval và hàm gettimeofday() để lấy thời gian thực
- struct timeval là một cấu trúc chứa: time_t tv_sec (số giây) và suseconds_t tv_usec (số microsecond)
- struct timeval start, end: lưu thời điểm bắt đầu và kết thúc của lệnh
- timetaken: biến double để tính thời gian thực thi
- if (argc != 2): kiểm tra người dùng có nhập đúng 1 lệnh hay không, nếu không thì in ra thông báo lỗi và kết thúc chương trình
- gettimeofday(&start, NULL): Lấy thời gian hiện tại và lưu vào start, NULL là không dùng timezone
- system(argv[1]): tạo một process con để thực thi lệnh truyền vào, hàm này chạy xong lệnh rồi mới trả về nên sau khi chạy xong mới tiếp tục sang bước sau
- gettimeofday(&end, NULL): lấy thời gian hiện tại sau khi system() chạy xong và lưu vào end

- timetaken = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1000000.0: Tính thời gian chênh lệch và chuyển sang giây

- Kết quả:



Hình 14 Kết quả thực thi file time.c

- Kết quả trả về lệnh ls có thời gian thực thi là 0.0017s

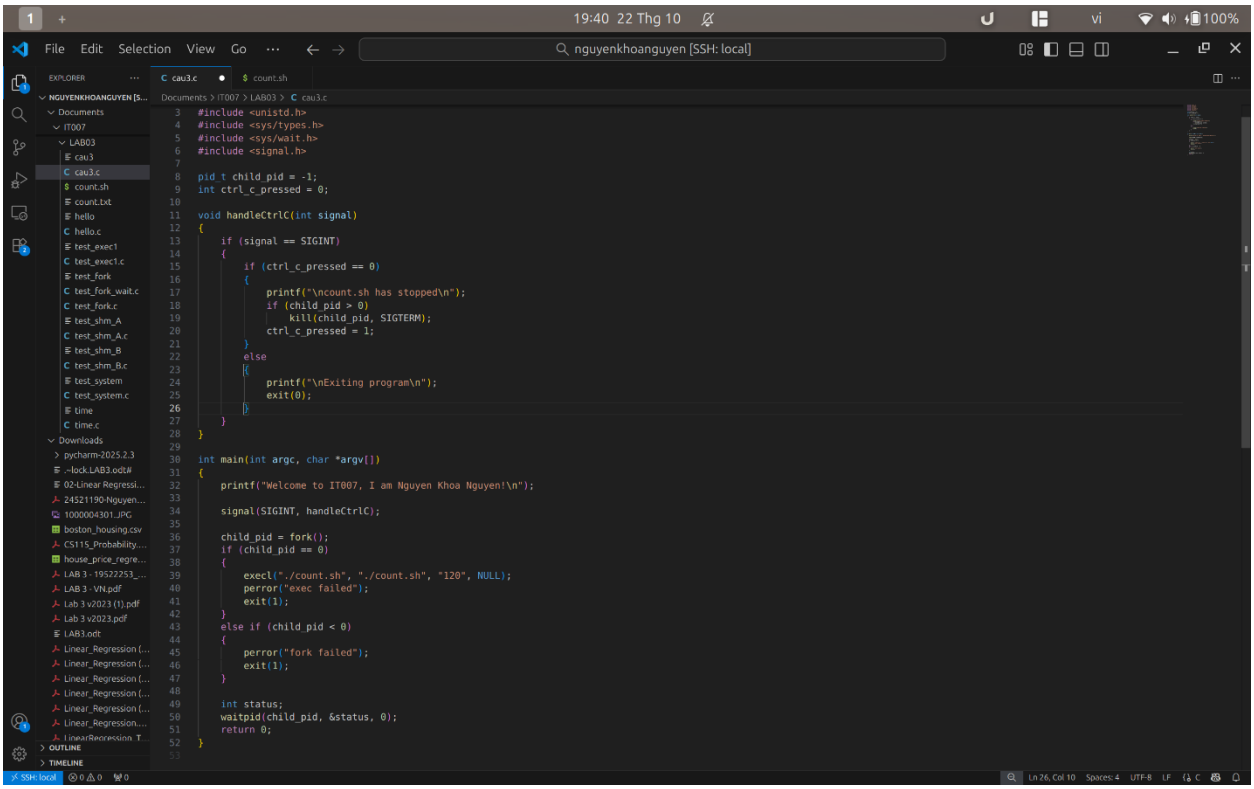
3. Viết một chương trình làm bốn công việc sau theo thứ tự:

- In ra dòng chữ: “Welcome to IT007, I am !”

- Thực thi file script count.sh với số lần đếm là 120

- Trước khi count.sh đếm đến 120, bấm CTRL+C để dừng tiến trình này

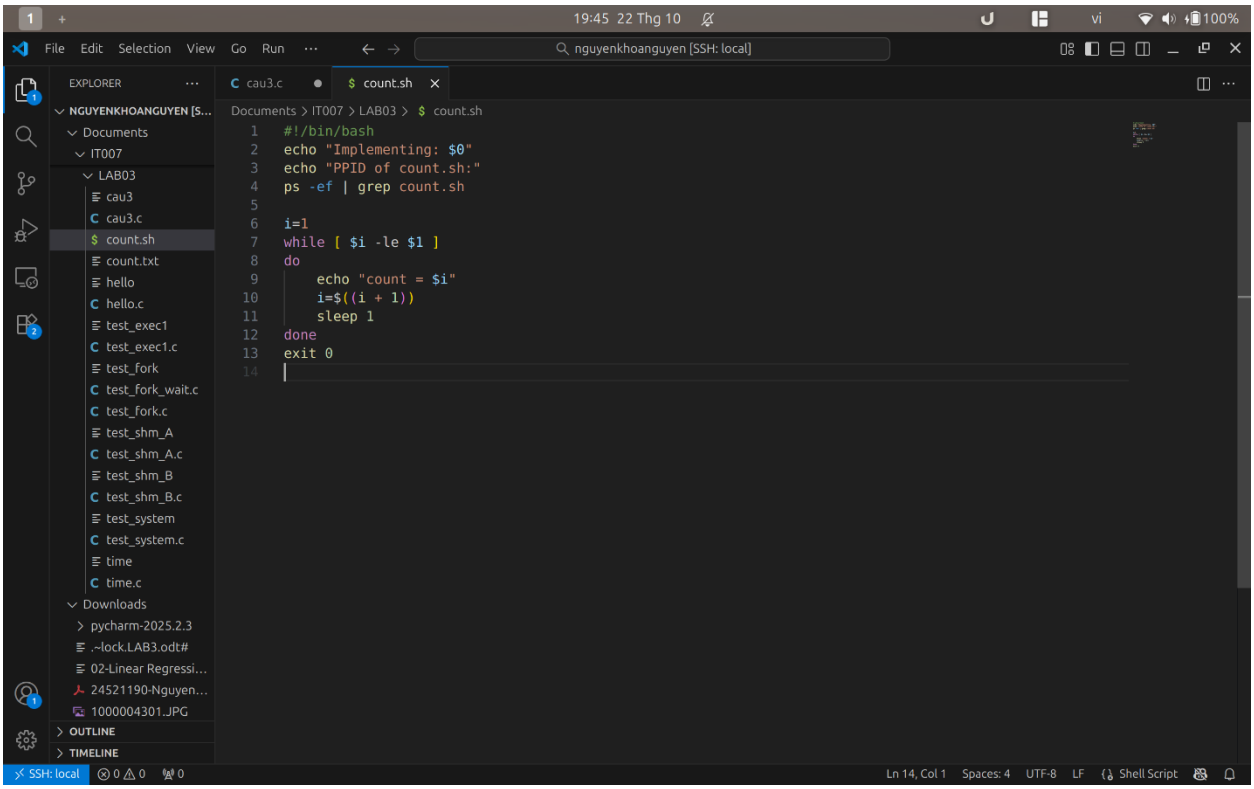
- Khi người dùng nhấn CTRL+C thì in ra dòng chữ: “count.sh has stopped”



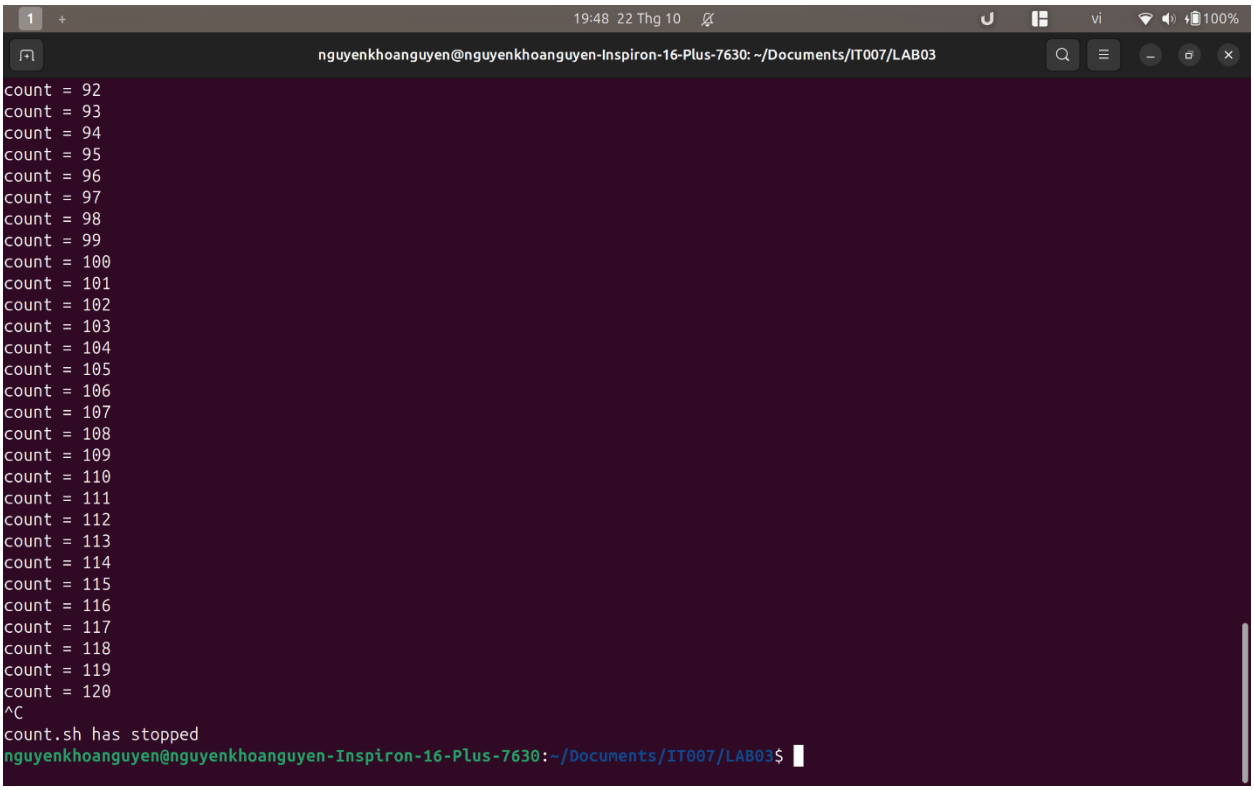
Hình 15 Nội dung code file cau3.c

- Nội dung code:

- Hàm handleCtrlC: Khi người dùng nhấn Ctrl + C, Linux sẽ gửi tín hiệu SIGINT cho chương trình. Nếu là lần đầu nhấn Ctrl + C thì in ra "count.sh has stopped". Nếu bấm 2 lần sẽ thoát hoàn toàn chương trình
- signal(SIGINT, handleCtrlC): đăng ký hàm handleCtrlC để xử lý tín hiệu SIGINT
- child_pid = fork(): Tạo ra một tiến trình mới
- if (child_pid == 0): hàm execl() thay tiến trình con bằng tiến trình mới (count.sh) với tham số là đường dẫn đến script và đối số truyền vào script
- else if (child_pid < 0): nếu tạo tiến trình thất bại thì in lỗi và thoát
- waitpid(): giúp tiến trình cha chờ tiến trình con kết thúc, nếu người dùng nhấn Ctrl + C và hàm handleCtrlC gửi SIGTERM cho con thì waitpid() sẽ thấy con đã chết và kết thúc



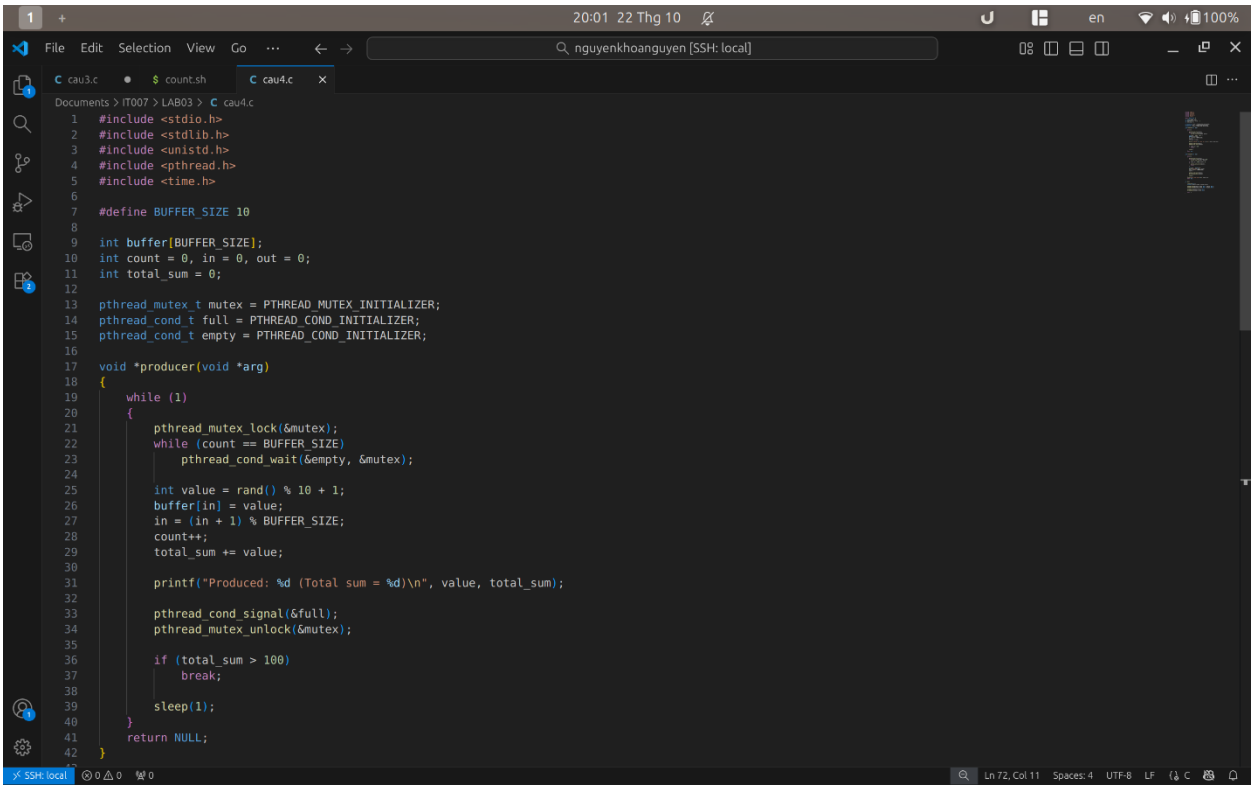
Hình 16 Nội dung file count.sh



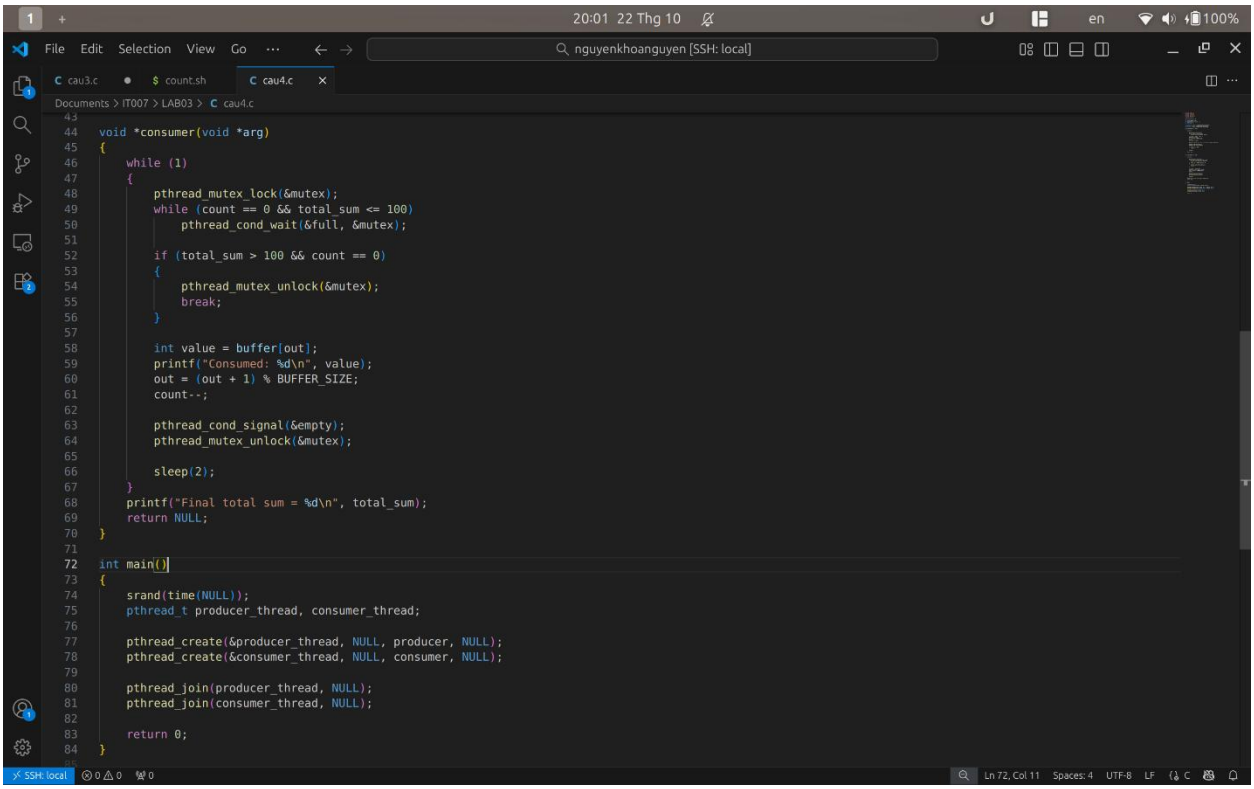
Hình 17 Kết quả thực thi file bai3.c

4. Viết chương trình mô phỏng bài toán Producer - Consumer như sau:

- Sử dụng kỹ thuật shared-memory để tạo một bounded-buffer có độ lớn là 10 bytes.
- Tiến trình cha đóng vai trò là Producer, tạo một số ngẫu nhiên trong khoảng [10, 20] và ghi dữ liệu vào buffer
- Tiến trình con đóng vai trò là Consumer đọc dữ liệu từ buffer, in ra màn hình và tính tổng
- Khi tổng lớn hơn 100 thì cả 2 dừng lại



Hình 18 Nội dung code file cau4.c (1)



Hình 19 Nội dung code file file cau4.c (2)

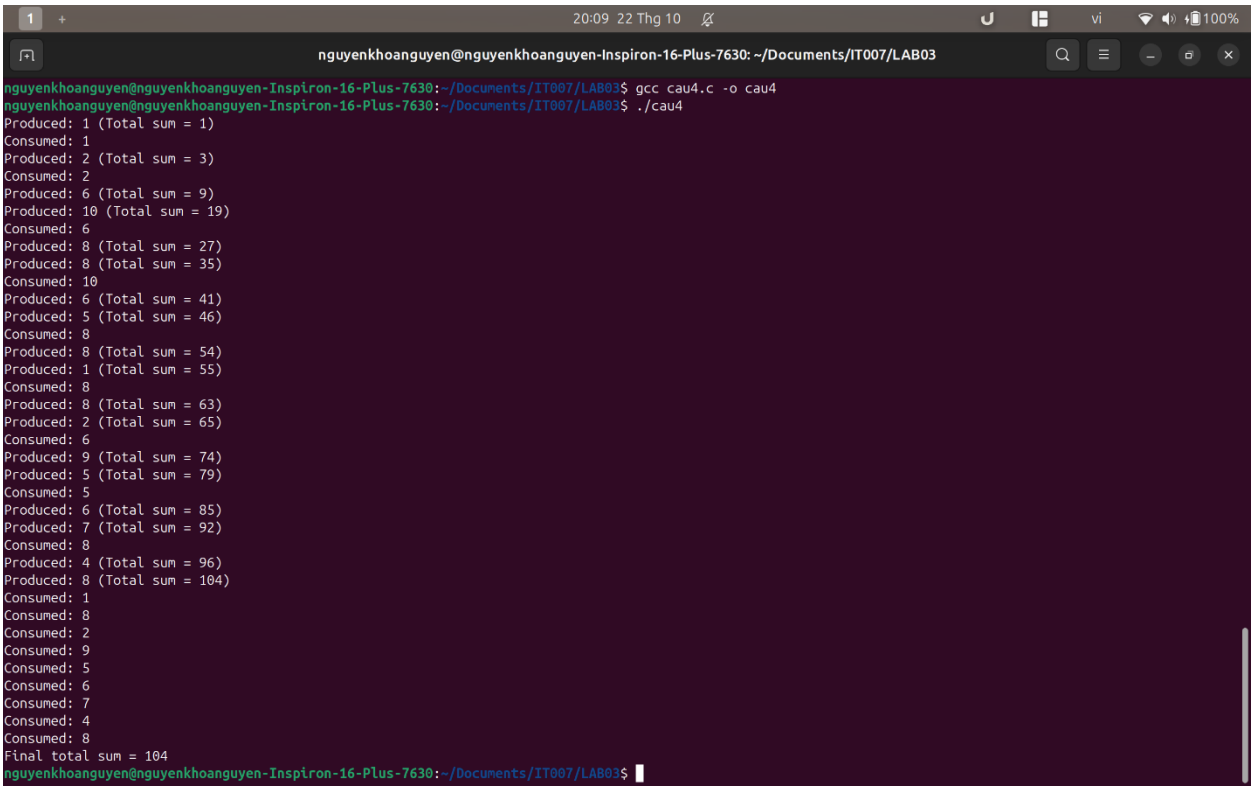
- Giải thích code:

- Ta có hai tiến trình: producer tạo ra các số ngẫu nhiên bỏ vào buffer và consumer lấy các số trong buffer ra tiêu thụ. Hai tiến trình chạy song song và chia sẻ chung vùng nhớ buffer[] nên phải đồng bộ để tránh producer ghi vào khi buffer đầy và consumer đọc khi buffer trống
- buffer: mảng dùng làm vùng bounded buffer
- in: chỉ vị trí producer sẽ ghi tiếp theo
- out: chỉ vị trí consumer sẽ đọc tiếp theo
- count: số phần tử hiện có trong buffer
- total_sum: tổng các giá trị mà producer đã sinh ra, và dừng khi vượt quá 100
- mutex: khóa và chỉ 1 tiến trình được vào vùng dùng chung tại 1 thời điểm
- full: condition variable dùng để báo buffer đã có dữ liệu
- empty: báo buffer còn trống
- Hàm producer: Khóa mutex sẽ đảm bảo không có consumer nào truy cập buffer cùng lúc. Nếu buffer đầy thì producer đợi condition empty và khi có chỗ trống producer sẽ sinh số ngẫu nhiên thêm vào buffer. Sau đó in ra thông tin, cộng tổng và báo cho consumer biết rằng buffer đã có dữ

liệu (pthread_cond_signal(&full)). Cuối cùng mở khóa cho tiến trình khác vào và tổng > 100 thì dừng.

- Hàm consumer: Consumer khóa vùng dùng chung. Nếu buffer trống thì chờ producer báo đã thêm dữ liệu. Nếu total_sum > 100 và buffer đã trống thì thoát vòng lặp. Sau đó lấy phần tử ra khỏi buffer và giảm count, báo cho producer đã có chỗ trống và mở khóa để producer có thể ghi tiếp. Cuối cùng in ra tổng giá trị đã tạo
- Hàm main: pthread_create() để tạo ra hai luồng (hoặc cũng có thể là tiến trình) và pthread_join để đợi cho cả hai luồng kết thúc trước khi thoát chương trình

- Kết quả



Hình 20 Kết quả thực thi file cau4.c