|  |
| --- |
| Trường Đại học Khoa học tự nhiên – Khoa Công nghệ thông tin. |
| Đồ án số 01 |
| HỆ ĐIỀU HÀNH – Operating System |

|  |
| --- |
| Richard Nguyen  Tháng 10, 2024. |

**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN**

**KHOA CÔNG NGHỆ THÔNG TIN**

**📖👨‍💻📖**

**A blue and white logo

Description automatically generated**

**ĐỒ ÁN THỰC HÀNH SỐ 01**

**Bộ môn:** Hệ điều hành.

**Tên đề tài:**

*“XV6 and Unix Utilities”.*

**Tên nhóm:** 3T.

**Thành viên:**

1. Trần Đức Trí – 22120387.
2. Vũ Hoàng Nhật Trường – 22120398.
3. Nguyễn Anh Tường – 22120412.

# **Thông tin chung:**

1. **Bộ môn:** Hệ điều hành.
2. **Giảng viên lý thuyết:** Thầy Trần Trung Dũng.
3. **Giảng viên thực hành:** Thầy Nguyễn Thanh Quân.
4. **Mã lớp:** 22\_4.
5. **Tên nhóm:** 3T.
6. **Danh sách thành viên:**
   1. Nguyễn Đình Trí – 22120384.
   2. Vũ Hoàng Nhật Trường – 22120398.
   3. Nguyễn Anh Tường – 22120412.
7. **Link github repository:** [“Click here to go to our github repository”](https://github.com/richardnguyen0715/OS-Proc-Lab01-XV6andUU.git)

MỤC LỤC

[**ĐỒ ÁN THỰC HÀNH SỐ 01** 2](#_Toc179826464)

[**Thông tin chung:** 3](#_Toc179826465)

[**Phần giới thiệu:** 5](#_Toc179826466)

[**Giới thiệu về các thành phần được sử dụng.** 6](#_Toc179826467)

[**Yêu cầu 1:** 7](#_Toc179826468)

[**I.** **Thành viên thực hiện:** 8](#_Toc179826469)

[**II.** **Mô tả thực hiện:** 8](#_Toc179826470)

[**III.** **Kết quả chương trình:** 10](#_Toc179826471)

[**Yêu cầu 2:** 11](#_Toc179826472)

[**I.** **Thành viên thực hiện:** 12](#_Toc179826473)

[**II.** **Mô tả thực hiện:** 12](#_Toc179826474)

[**III.** **Một số vấn đề đáng chú ý để hoàn thành yêu cầu 2 ( đã được giải quyết ):** 17](#_Toc179826475)

[**IV.** **Kết quả chương trình:** 19](#_Toc179826476)

[**Yêu cầu 3:** 20](#_Toc179826477)

[**I.** **Thành viên thực hiện:** 21](#_Toc179826478)

[**II.** **Mô tả thực hiện:** 21](#_Toc179826479)

[**III.** **Một số lưu ý** 25](#_Toc179826480)

[**IV.** **Kết quả chương trình** 26](#_Toc179826481)

[**Yêu cầu 4:** 27](#_Toc179826482)

[**I.** **Thành viên thực hiện:** 28](#_Toc179826483)

[**II.** **Các bước thực hiện và kết quả chương trình:** 28](#_Toc179826484)

# **Phần giới thiệu:**

*GIỚI THIỆU CHUNG*

## **Giới thiệu về các thành phần được sử dụng.**

1. **Môi trường lập trình:**

* Linux: Máy ảo ( Ubuntu 24.04.1 LTS ).
* WSL: Ubuntu - Phiên bản 24.04.5 LTS.
* Qemu: QEMU emulator version 7.2.0.

1. **Công cụ lập trình:**

* Visual Studio Code.
* Riscv64-linux-gnu-gcc ( Debian 10.3.0-8) 10.3.0.
* Riscv64-unknown-elf-gcc ( GCC ) 10.1.0.
* Riscv64-unknown-linux-gnu-gcc ( GCC ) 10.1.0.

1. **Phiên bản XV6:**

* XV6 - UNIX 6th Edition – 2024.
  + 1st Edition ( June 14, 2000 ).
  + ISBN: 1-57398-013-7
  + **Link xv6 repo:** [Link here.](git://g.csail.mit.edu/xv6-labs-2024)

1. **Link github repository:** [“Click here to go to our github repository”](https://github.com/richardnguyen0715/OS-Proc-Lab01-XV6andUU.git)

# **Yêu cầu 1:**

***A blue line drawing of a server

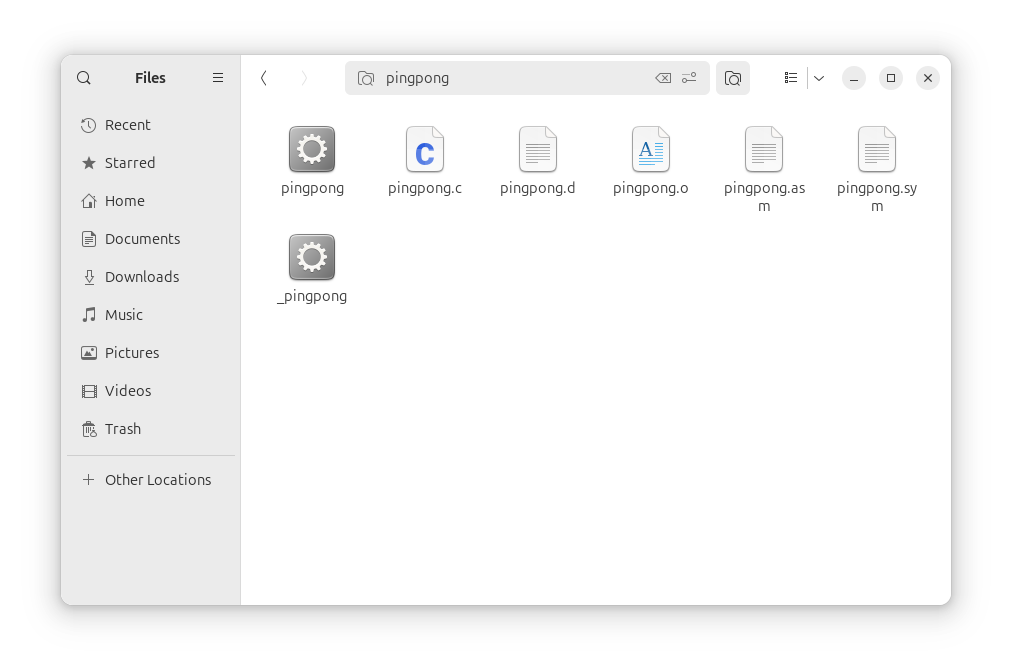
Description automatically generated****PINGPONG*

1. **Thành viên thực hiện:**

* Nguyễn Anh Tường – 22120412.

1. **Mô tả thực hiện:**

**Bước 1:** Tạo file *pingpong.c* trong thư mục **user** của XV6.



**Bước 2:** Vào môi trường VS code để lập trình cho file pingpong.c trên.

**Bước 3:** Sử dụng các thư viện được cung cấp bởi hệ điều hành XV6.

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

**Bước 4:** Tạo tiến trình con bằng hàm fork(). Tạo 2 pipe bằng hàm pipe(<pipe>).

int proc = fork();

...

int check\_pipe1 = pipe(pipe\_one); // Create a parent-to-child pipe.

int check\_pipe2 = pipe(pipe\_two); // Create a child-to-parent pipe.

**Bước 5:** Kiểm tra việc tạo pipe và fork có thành công hay không?

if (check\_pipe1 == -1 || check\_pipe2 == -1) {

    printf("Cannot create pipe! - Error: pipe() function failed!");

    return 0;

}

if (proc < 0) {

    printf("Cannot create a child process! - Error: fork() function failed!");

    return 0;

}

**Bước 6:** Kiểm tra **pid,** nếu bằng 0 tức là ta đang ở tiến trình con.

* Vì tiến trình con đọc trước nên ta đóng đầu ghi của pipe\_one.
* Đọc kí tự được gởi từ tiến trình cha.
* Sau khi nhận được ta in ra màn hình lệnh ping.
* Đóng đầu đọc của pipe\_one ngay sau đó.
* Ta đóng đầu đọc của pipe\_two để tiến hành ghi kí tự gởi đến cha.
* Sau khi ghi xong thì ta đóng đầu ghi của pipe\_two lại.

else if (proc == 0) { // Child process.

        char mess;

        close(pipe\_one[1]); // Close the write end of parent-to-child pipe.

        read(pipe\_one[0], &mess, 1); // Read from parent-to-child pipe.

        printf("%d: received ping\n", getpid());

        close(pipe\_one[0]); // Close the read end of parent-to-child pipe.

        close(pipe\_two[0]); // Close the read end of child-to-parent pipe.

        write(pipe\_two[1], &mess, 1); // Write to child-to-parent pipe.

        close(pipe\_two[1]); // Close the write end of child-to-parent pipe.

}

**Bước 6.2:** Kiểm tra nếu pid là một con số bất kì ( > 0 ) thì ta đang ở tiến trình cha.

* Ta tiến hành đóng đầu đọc của pipe\_one.
* Tiến hành viết kí tự vào đầu ghi của pipe\_one để gởi đến tiến trình con.
* Đóng đầu ghi của pipe\_one sau khi ghi xong.
* Đóng đầu ghi của pipe\_two.
* Mở đầu đọc của pipe\_two để tiến hành nhận thông tin từ tiến trình con.
* Sau khi nhận được thì ta in ra màn hình lệnh “pong”.
* Đóng đầu đọc của pipe\_two.
* Chờ cho tiến trình con kết thúc để kết thúc chương trình.

    else { // Parent process

        char mess = 'T'; // Because all of the member's name starts by 'T' letter

        close(pipe\_one[0]); // Close the read end of parent-to-child pipe.

        write(pipe\_one[1], &mess, 1); // Write to parent-to-child pipe.

        close(pipe\_one[1]); // Close the write end of parent-to-child pipe.

        close(pipe\_two[1]); // Close the write end of child-to-parent pipe.

        read(pipe\_two[0], &mess, 1); // Read from child-to-parent pipe.

        printf("%d: received pong\n", getpid());

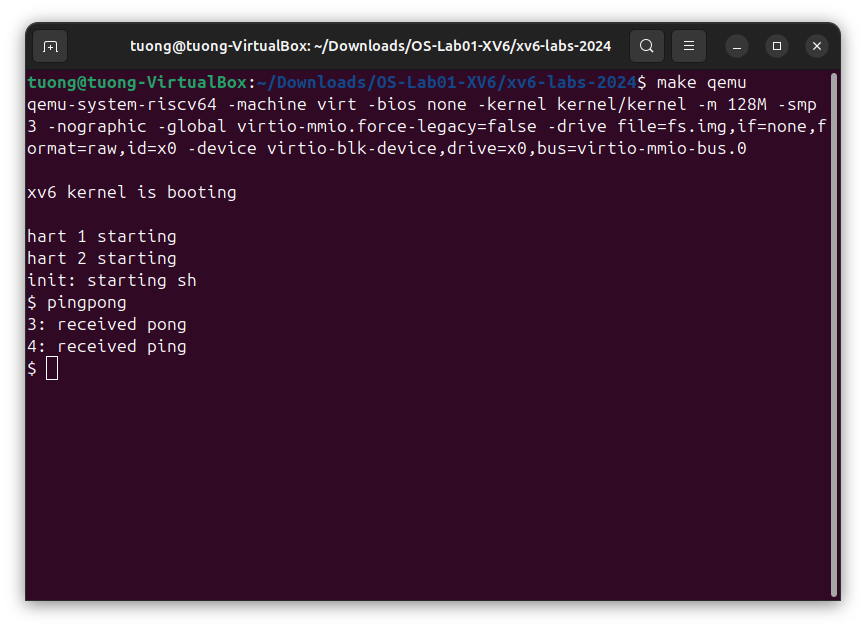
        close(pipe\_two[0]); // Close the read end of child-to-parent pipe.

        wait(0); // Wait for child process to finish.

        }

**Bước 7:** Hoàn thành chương trình.

1. **Kết quả chương trình:**

****

# **Yêu cầu 2:**

***A blue line drawing of a server

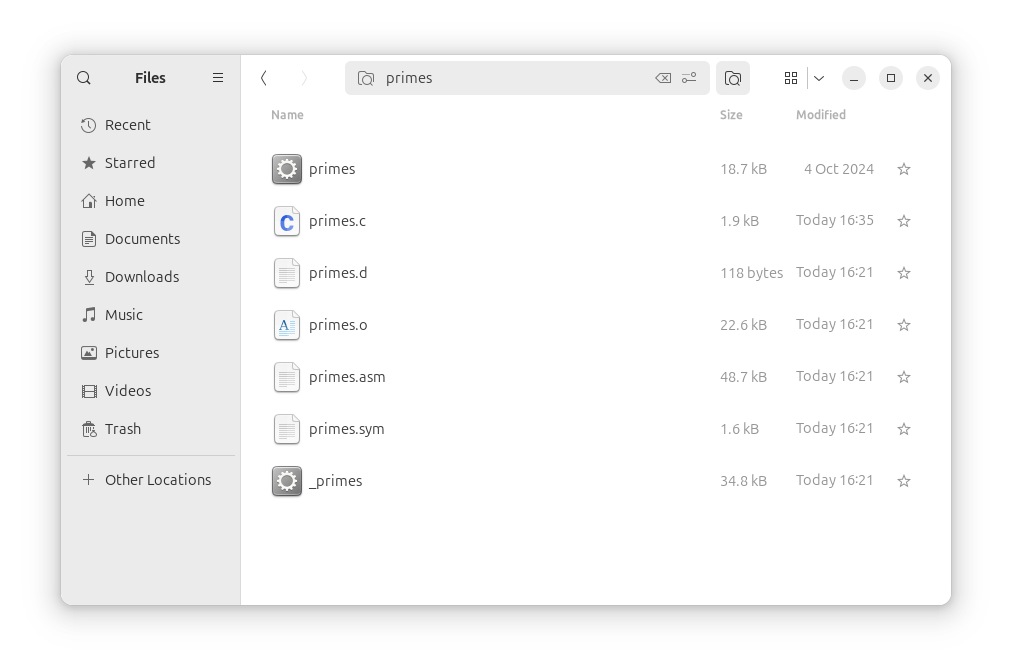
Description automatically generated****PRIMES*

1. **Thành viên thực hiện:**

Nguyễn Anh Tường – 22120412.

1. **Mô tả thực hiện:**

**Bước 1:** Tạo file *primes.c* trong thư mục **user** của XV6.



**Bước 2:** Vào môi trường VS code để lập trình cho file primes.c trên.

**Bước 3:** Sử dụng các thư viện được cung cấp bởi hệ điều hành XV6.

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

**Bước 4:** Tạo tiến trình con và pipe đầu tiên lần lượt bằng hàm fork() và pipe(). Đồng thời kiểm tra xem có tại được không?

int fd[2];

    int c\_pipe = pipe(fd); //fd[0]: read | fd[1]: write

    if(c\_pipe == -1){

        printf("Cannot create pipe! - Error: pipe() function failed!");

        exit(0);

    }

    int pid = fork();

    if(pid < 0){

        printf("RaiseError: Cannot create process!");

        exit(0);

    }

**Bước 5:** Kiểm tra tiến trình bằng PID, nếu là tiến trình con thì bằng 0, lớn hơn không là tiến trình cha.

**Nếu là tiến trình con.**

* Tiến hành đóng đầu ghi của pipe lại.
* Ta chạy hàm primes – là một hàm đệ quy – để thực hiện tìm kiếm các số nguyên tố.

**Nếu là tiến trình cha.**

* Tiến hành đóng đầu đọc của pipe lại.
* Chạy một vòng lặp FOR từ 2 đến 280, bước nhảy là 1.
  + Với mỗi vòng lặp ta tiến hành ghi số nguyên ghi số nguyên hiện tại vào trong pipe.
* Sau khi hết vòng lặp, ta tiến hành đóng pipe lại.
* Đợi tiến trình con kết thúc.

else if (pid == 0){

        //This is the child process

        close(fd[1]);

        primes(fd[0]);

    }

 else{

        //This is the parent process

        close(fd[0]);

        for( int i = 2; i <= 280; i++){

            write(fd[1], &i, sizeof(i));

        }

        close(fd[1]);

        wait(0);

  }

**Bước 6:** Thực thi hàm Primes. Có tham số là một pipe và không có giá trị trả về.

**Lưu ý:** Ta thêm dòng code sau để tránh bị lỗi vòng lặp vô hạn trong Qemu.

void primes(int cur\_pipe)\_\_attribute\_\_((noreturn));

void primes(int cur\_pipe){

...

}

**Bước 7:** Ta nhận dữ liệu được gởi từ cha thông qua pipe được truyền vào. Sau đó ta kiểm tra xem có nhận dữ liệu được không, nếu không thì ta đóng pipe và thoát hàm. Nếu dữ liệu nhận được ok thì ta tiến hành in ra màn hình.

int val;

int c\_read = read(cur\_pipe, &val, sizeof(val)); // c\_read: check\_read

if(c\_read == 0){

  close(cur\_pipe);

  exit(0);

}

printf("prime: %d\n",val);

**Bước 8:** Tiến hành tạo pipe, fork tiếp theo và kiếm tra. Nếu tạo không được thì ta in ra lỗi, đóng các pipe hiện có và thoát chương trình.

int fd[2];

if (pipe(fd) == -1) {

   printf("Error: pipe creation failed!\n");

   close(cur\_pipe);

   exit(1);

}

int pid = fork();

if(pid < 0){

   printf("Error: fork failed!\n");

   close(fd[0]);

   close(fd[1]);

   close(cur\_pipe);

   exit(1);

}

**Bước 9:** Kiểm tra xem ta đang ở tiến trình nào?

**Nếu là tiến trình con:** PID = 0.

* Ta đóng pipe ghi của pipe hiện tại.
* Tiến hành gọi đệ quy hàm primes với tham số là pipe đọc của pipe hiện tại

else if(pid == 0){

    close(fd[1]); // Close write pipe in child process.

close(curr\_pipe); // Close curr\_pipe in child process. Got the Buffer overflows if you don’t close.

    primes(fd[0]);

}

**Nếu là tiến trình cha:** PID > 0.

* Tiến hành đọc pipe tham số cho đến khi gặp được một số nguyên tố thì sẽ tiến hành ghi số nguyên tố đó vào pipe tiếp theo và dừng vòng lặp lại.
* Nếu gặp lỗi thì in lỗi ra màn hình, ngược lại chạy tiếp.
* Đóng các pipe hiện tại và đợi cho tiến trình con kết thúc.
* Cuối cùng là thoát hàm.

    else {

        int check\_prime;

        close(fd[0]); // Close read pipe in parent process.

        while((c\_read = read(cur\_pipe, &check\_prime, sizeof(check\_prime))) > 0){

            if(check\_prime % val != 0){

                if (write(fd[1], &check\_prime, sizeof(check\_prime)) == -1) {

                    printf("Error writing to pipe\n");

                    break;

                }

            }

        }

        if(c\_read == -1) {

            printf("Error reading from pipe\n");

        }

        close(fd[1]);

        close(cur\_pipe);

        wait(0);

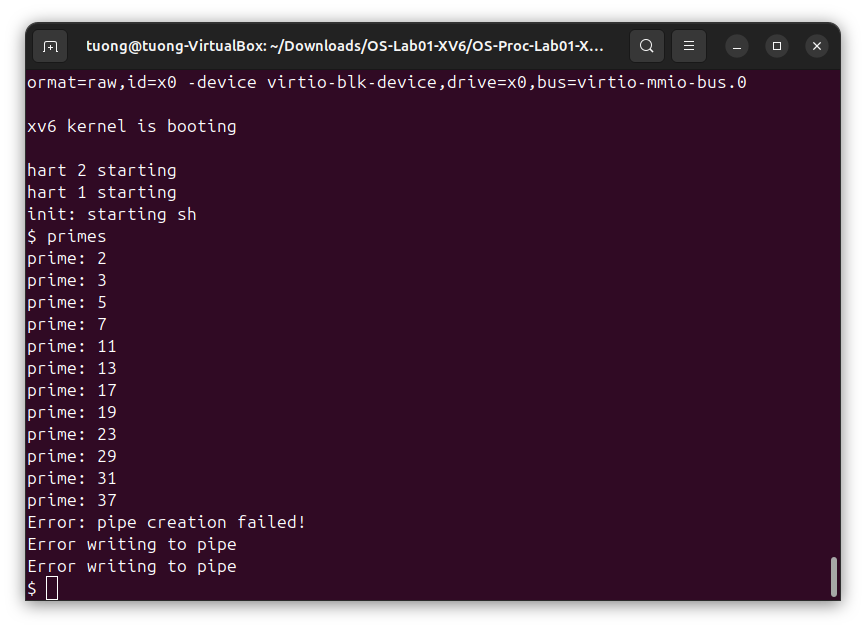
        exit(0);

    }

1. **Một số vấn đề đáng chú ý để hoàn thành yêu cầu 2 ( đã được giải quyết ):**

Vì XV6 là một hệ điều hành khá đơn giản và nhẹ nhàng, nên việc tạo nhiều pipe và nhiều tiến trình, gọi đệ quy,… đã tốn không ít tài nguyên. Do đó ta phải chỉnh sửa lại một số thông số của hệ điều hành để chương trình có thể chạy một cách yêm xuôi.

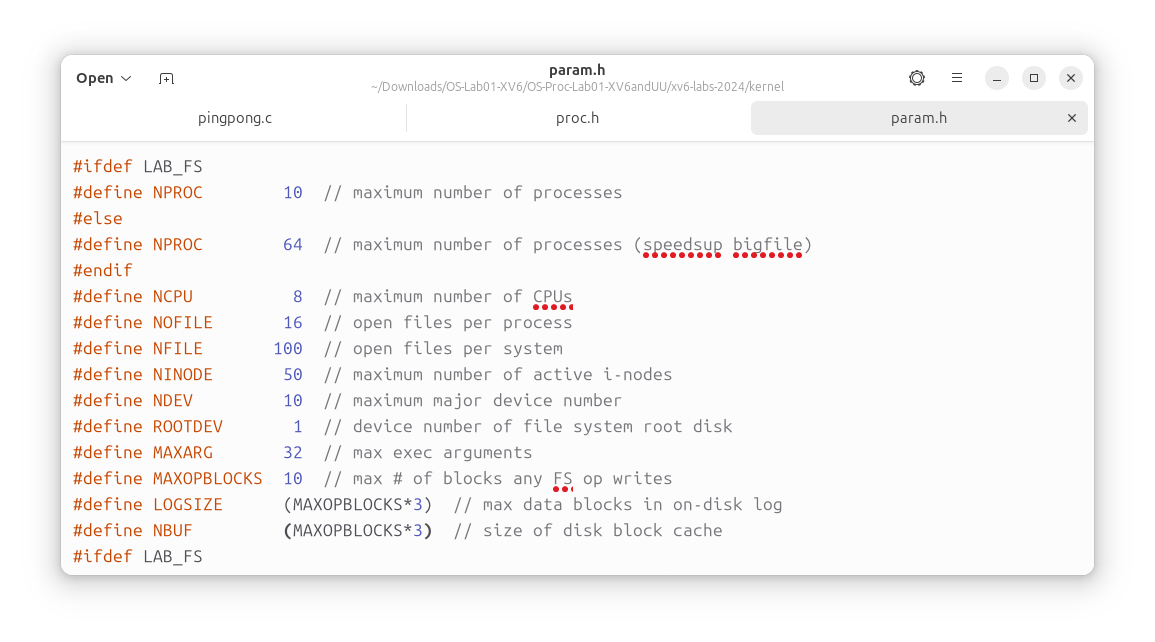
* 1. **Vấn đề ban đầu:**

****

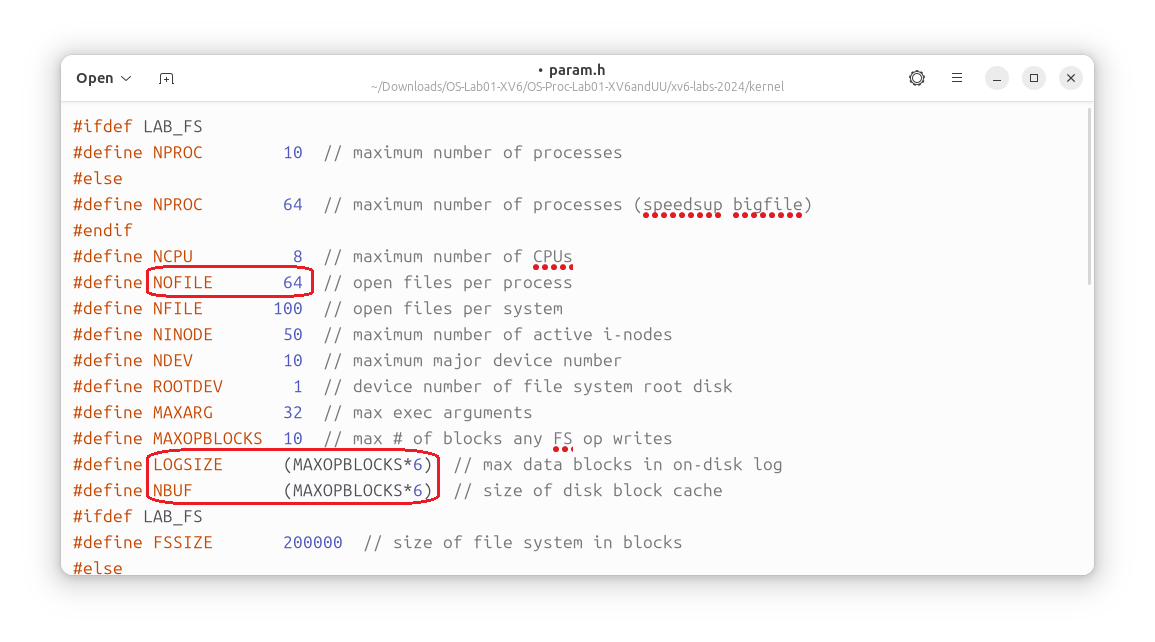
* **Chương trình không thể tạo thêm pipe do không đủ tài nguyên.**
  1. **Hướng khắc phục số 1:**

Điều chỉnh thông số cho hệ điều hành để trành tình trạng **Buffer Overflow** này.

**Thông số ban đầu:**

****

**Thông số sau khi tuỳ chỉnh:**



* 1. **Hướng khắc phục số 2:**

Điều chỉnh lại tính hợp lý của việc đóng, mở của các pipe.

Đặc biệt là trong phần gọi đệ quy ở tiến trình con.

else if(pid == 0){

    close(fd[1]); // Close write pipe in child process.

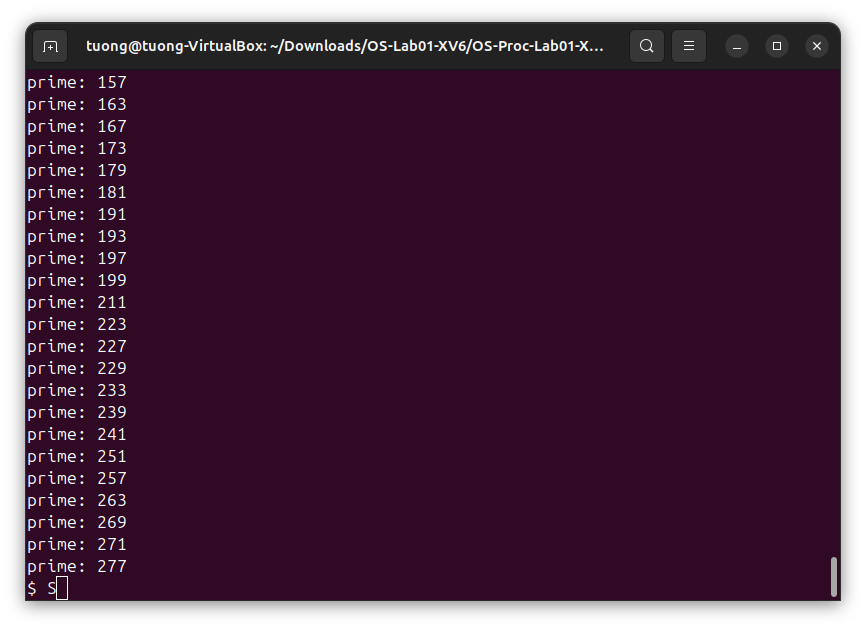
close(curr\_pipe); // Close curr\_pipe in child process. Got the Buffer overflows if you don’t close.

    primes(fd[0]);

}

* Ở đây phải đóng curr\_pipe, nếu không sẽ bị buffer overflow.

1. **Kết quả chương trình:**



# **Yêu cầu 3:**

***A blue line drawing of a server

Description automatically generated****FIND*

1. **Thành viên thực hiện:**

Vũ Hoàng Nhật Trường – 22120398

1. **Mô tả thực hiện:**

**Bước 1:** Thêm file find.c vào thư mục user.

A screenshot of a computer

Description automatically generated

**Bước 2:** Thêm chương trình find.c vào UPROGS (trong Makefile) để QEMU có thể thực hiện.

A computer screen shot of a code

Description automatically generated

**Bước 3:** Lập trình thuật toán:

* + - Ở đây ta xem xét cấu trúc 1 câu lệnh find trong QEMU như sau:

bao gồm *chỉ thị thực hiện* **find,** *thư mục gốc* đại diện cho dấu **.** và *thư mục cần tìm* là **b**. Từ đó ta kết luận rằng số lượng argument cần thiết sẽ là 3+.

A screen shot of a computer

Description automatically generated

* Nếu người dùng cố tình nhập không đủ 3 arguments như định dạng trên thì chương trình sẽ báo lỗi và thoát.
  + - **Thuật toán trong find.c như sau:**

Sau khi ta có đường dẫn của thư mục gốc ban đầu (argv[1]), ta sẽ cho đọc nội dung thư mục đó bằng các lệnh fd(), fstat() và read(), kết quả trả về là 1 list chứa thông tin các thư mục con (tương tự như lệnh ls).

A computer code with colorful text

Description automatically generated

A close up of a logo

Description automatically generated

A screenshot of a computer

Description automatically generated

* + - Sau đó ta sẽ tìm file giống với tên ta cần tìm (argv[2], argv[3],…). Nếu nó là 1 định dạng thư mục (st.type của nó là T\_DIR), sau khi kiểm tra xong thì sẽ đệ quy vào bên trong để tìm kiếm thêm, cuối cùng lặp đến khi nào không có thư mục nào khác nữa. Lưu ý là khi chạy đến 2 thư mục . và .. thì ta sẽ không đệ quy vì đây là thư mục dẫn tới thư mục cha và chính nó, nếu đệ quy sẽ ra vòng lặp vô tận.
    - Các hàm hỗ trợ:

A close up of a text

Description automatically generated

Hàm dùng để kiểm tra 2 chuỗi có trùng tên hay không.

A black background with white text

Description automatically generated

Hàm dùng để tìm kiểm địa chỉ thư mục đích. Hàm này có 2 tham số: chuỗi chứa địa chỉ cần tìm và chuỗi chứa tên thư mục hiện tại (để tiện cho việc đệ quy).

A close up of a computer screen

Description automatically generated

Nếu là . và .. thì không vào vòng lặp đề quy.

A computer code with colorful text

Description automatically generated with medium confidence

Nếu đúng là thư mục cần tìm thì in ra màn hình.

A computer screen shot of text

Description automatically generated

Khi nội tại chính nó là thư mục thì sẽ đệ quy vào bên trong.

1. **Một số lưu ý**
   * + Khi ta truyền 1 địa chỉ ban đầu để tìm kiếm thư mục con, ta phải chắc chắn rằng địa chỉ ấy phải có trong thư mục gốc. Nếu ta lỡ cd vào thư mục con rồi tìm, chương trình sẽ báo lỗi.

A screenshot of a computer program

Description automatically generated

* + - Vì trong thư mục con ta không thể chạy các lệnh cơ bản của chương trình được, do chương trình không biết các lệnh đó lấy ở đâu. Nếu ta muốn bắt đầu địa chỉ là ở thư mục con, ta phải thêm đường dẫn từ gốc vào trong thì chương trình mới chạy được bình thường.

A black background with white text

Description automatically generated

1. **Kết quả chương trình**

A screenshot of a computer code

Description automatically generated

Đối với lệnh find có 3 arguments (tìm file b).

A screenshot of a computer code

Description automatically generated

Đối với lệnh find có 4 arguments (tìm file b và c).

# **Yêu cầu 4:**

***A blue line drawing of a server

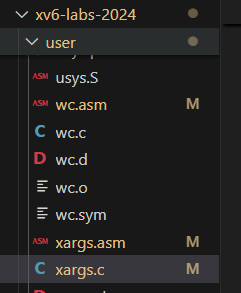
Description automatically generated****XARGS*

1. **Thành viên thực hiện:**

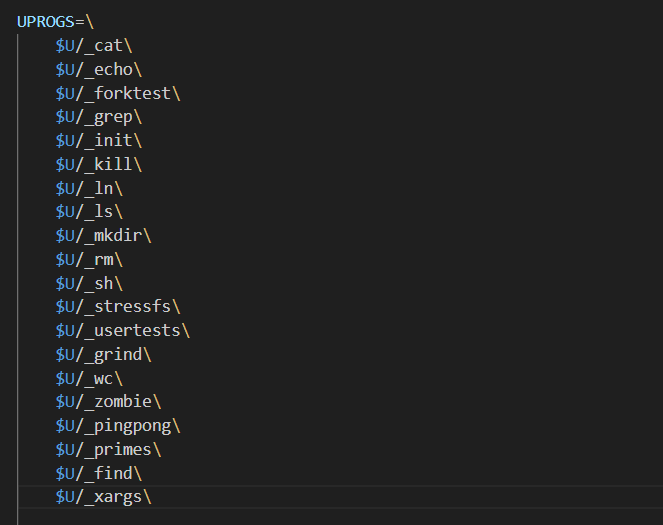
Nguyễn Đình Trí, MSSV:22120384

1. **Các bước thực hiện và kết quả chương trình:**

**Bước 1:** Tạo file xargs.c trong thư mục user.

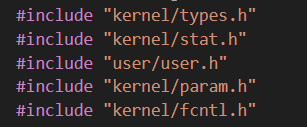


**Bước 2:** Thêm chương trình xargs.c vào UPROGS (trong Makefile) để QEMU có thể thực thi lệnh.



**Bước 3:** Tiến hành lập trình file xargs.c trên VS Code

**Bước 3.1:** Các thư viện được sử dụng



**#include "kernel/types.h"**: Bao gồm các định nghĩa về kiểu dữ liệu cơ bản.

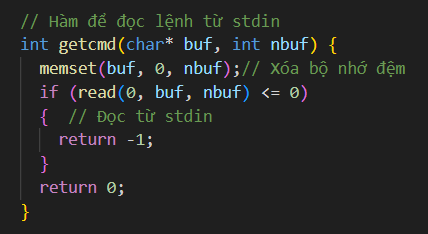
**#include "kernel/stat.h"**: Chứa các thông tin về trạng thái tệp.

**#include "user/user.h"**: Cung cấp các chức năng liên quan đến hệ thống người dùng như fork(), exec(), exit(), v.v.

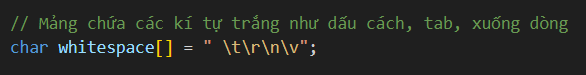
**#include "kernel/param.h"**: Định nghĩa số lượng đối số tối đa (MAXARG).

**#include "kernel/fcntl.h"**: Chứa các định nghĩa về quyền truy cập tệp.

**Bước 3.2:** Xây dựng hàm để đọc lệnh từ cmd

****

**Bước 3.3:** Tạo mảng : char whitespace[] = " \t\r\n\v"; chứa các kí tự khoảng trắng như dấu cách, mũi tên tới, dấu tab, xuống hàng. Mảng này phục vụ cho việc tách các đối số trong input.



**Bước 3.4:** Xây dựng hàm để tách chuỗi các đối số từ input thành các token.

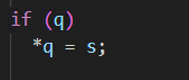
**-** Hàm này sử dụng các con trỏ **ps, es, q, eq.**

- Con trỏ **ps** trỏ tới con trỏ, **\*ps** chính là con trỏ tới vị trí đầu trong chuỗi đầu vào.

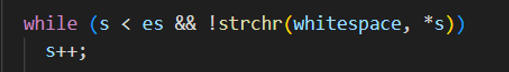
- Biến **s** trỏ đến cùng vị trí trong chuỗi mà **ps** đang trỏ tới. Biến này sẽ được dùng để dịch chuyển qua từng ký tự trong chuỗi, giúp phân tích chuỗi từ vị trí hiện tại



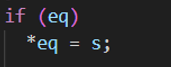
- Sau khi loại bỏ các ký tự trắng, **q** sẽ được gán bằng **s**, tức là **q** sẽ trỏ đến vị trí bắt đầu của từ mới (đối số mới) trong chuỗi



- Vòng lặp này sẽ duyệt qua chuỗi cho đến khi nó gặp kí tự trắng tiếp theo, **s** sẽ trỏ đến cuối của từ/đối số hiện tại

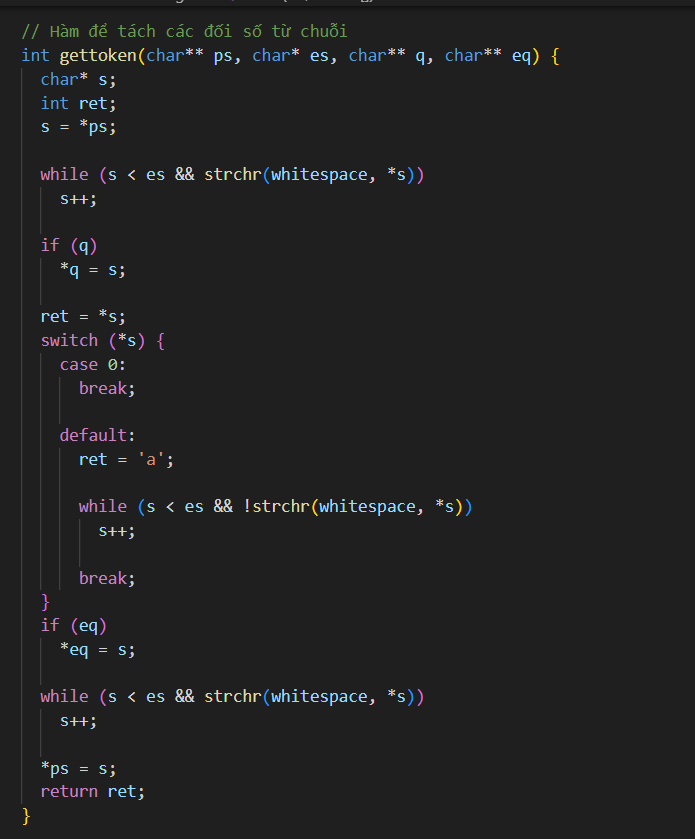


- **eq** sẽ được gán bằng **s**, tức là trỏ đến vị trí ngay sau từ/đối số đó trong chuỗi



- Sau khi xử lý xong một đối số, con trỏ **ps** sẽ được cập nhật để trỏ đến vị trí tiếp theo trong chuỗi (vị trí sau khoảng trắng tiếp theo)



****

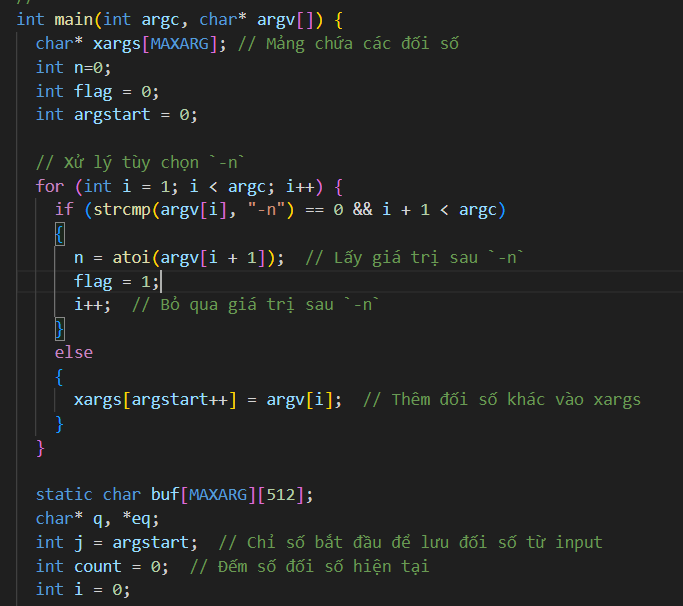
**Bước 3.5:** Xây dựng hàm main

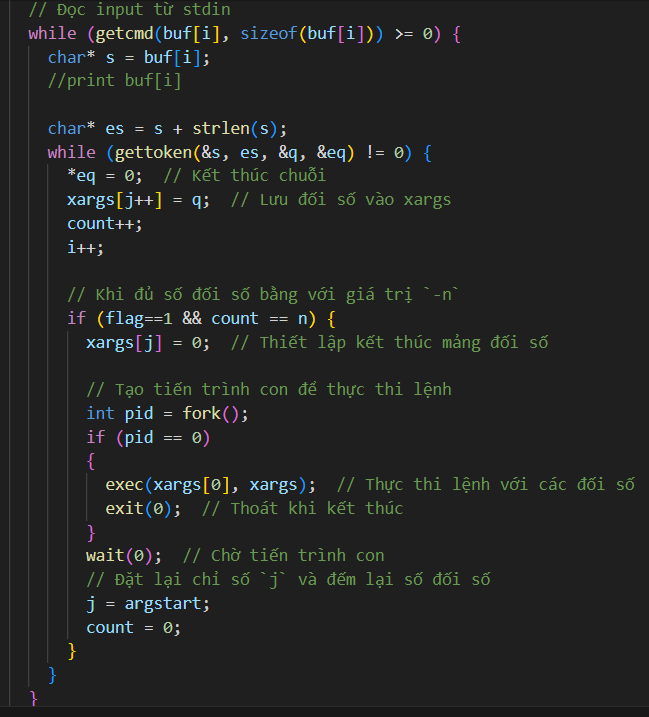
- Ta tạo biến **flag**, biến này sẽ kiểm tra xem có điều kiện về số đối số truyền

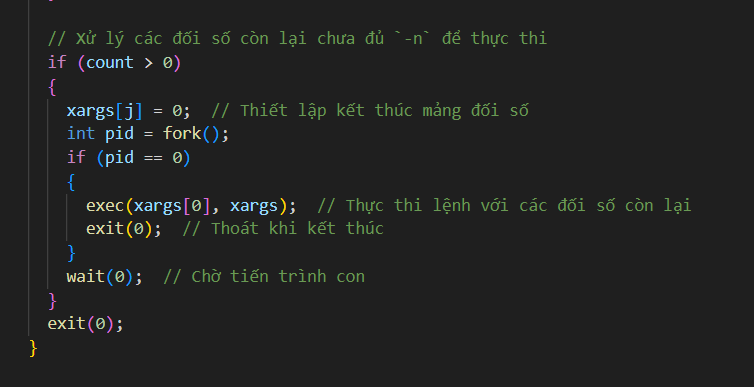
vào hay không. Lúc này ta chia ra trường hợp:

+ Nếu **flag = 1** (tức là có điều kiện): Ta sẽ thêm từng đối số vào mảng xargs, nếu đủ số lượng đối số, thì ta tạo tiến trình con và thực thi, sau đó tiếp tục thực hiện quá trình trên, cho đến khi số đối số còn lại nhỏ hợn n, thì tạo tiến trình con bên ngoài và thực hiện.

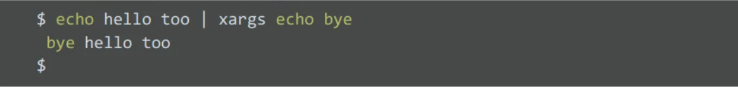
+ Nếu **flag = 0** (tức là không có điều kiện): Ta sẽ thêm tất cả đối số vào mảng xargs rồi sau đó tạo tiến trình con thực hiện 1 lần.



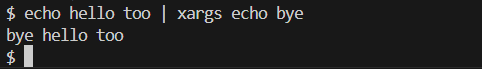




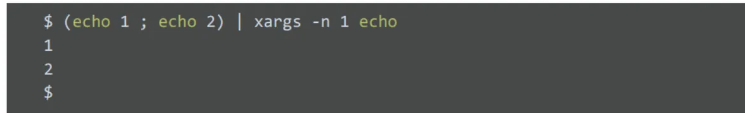
**Bước 4:** Test kết quả

- Yêu cầu 1:

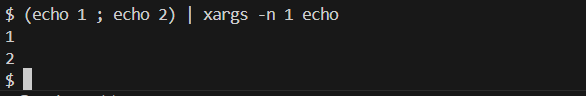
🡺 Kết quả:



- Yêu cầu 2:



🡺 Kết quả:



- Yêu cầu 3:



🡺 Kết quả:

