

Trường Đại học Khoa học tự nhiên – Khoa Công nghệ thông tin.

Đồ án số 01

HỆ ĐIỀU HÀNH – Operating System

Nhóm 3T
Tháng 10, 2024.

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN



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.

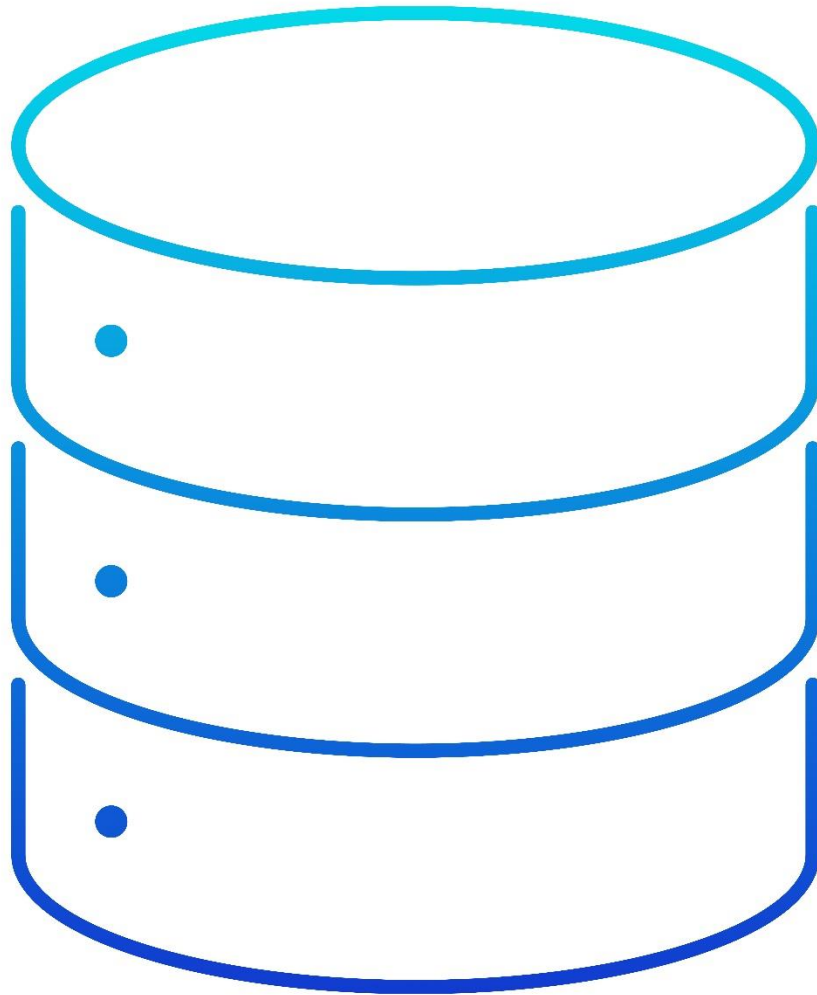
Thông tin chung:

- 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:**
 - a. Nguyễn Đình Trí – 22120384.
 - b. Vũ Hoàng Nhật Trường – 22120398.
 - c. Nguyễn Anh Tường – 22120412.
- 7. Link github repository:** [“Click here to go to our github repository”](#)

MỤC LỤC

ĐỒ ÁN THỰC HÀNH SỐ 01	2
Thông tin chung:	3
Phân giới thiệu:	5
Giới thiệu về các thành phần được sử dụng	6
Yêu cầu 1:	7
I. Thành viên thực hiện:	8
II. Mô tả thực hiện:	8
III. Kết quả chương trình:	10
Yêu cầu 2:	11
I. Thành viên thực hiện:	12
II. Mô tả thực hiện:	12
III. Một số vấn đề đáng chú ý để hoàn thành yêu cầu 2 (đã được giải quyết):	17
IV. Kết quả chương trình:	19
Yêu cầu 3:	20
I. Thành viên thực hiện:	21
II. Mô tả thực hiện:	21
III. Một số lưu ý	25
IV. Kết quả chương trình	26
Yêu cầu 4:	27
I. Thành viên thực hiện:	28
II. Các bước thực hiện và kết quả chương trình:	28

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.

2. Công cụ lập trình:

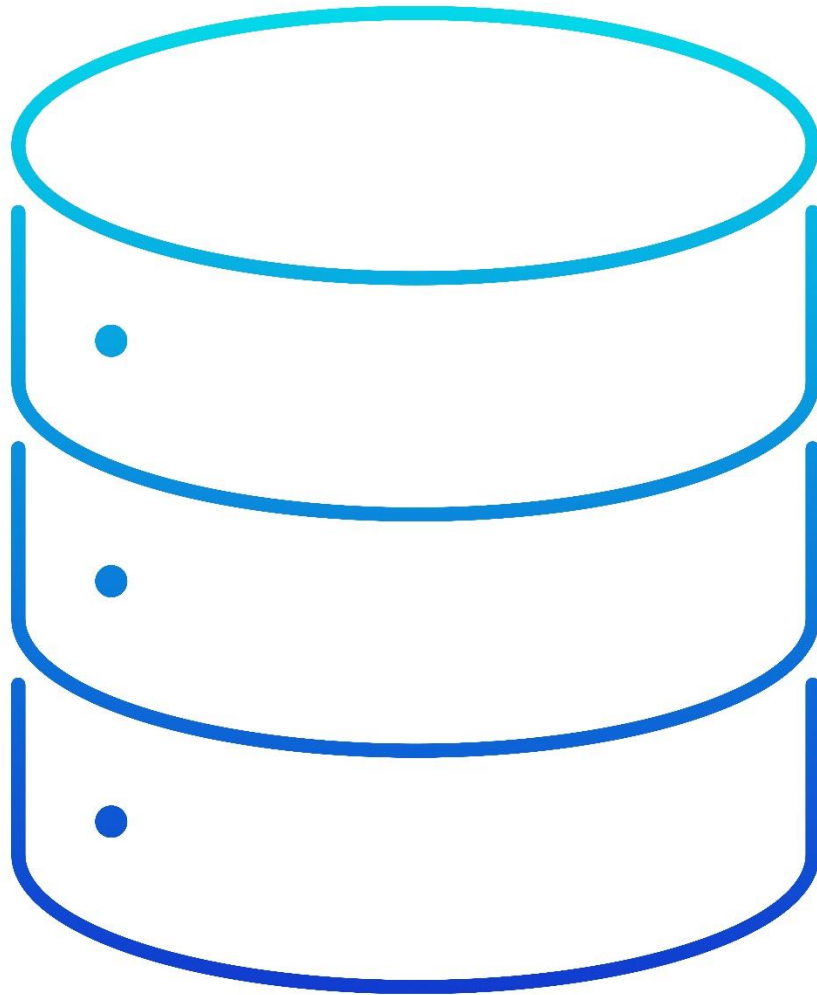
- Visual Studio Code.
- Riscv64-linux-gnu-gcc (Debian 10.3.0-8) 10.3.0.
- Riscv64-unknown-elf-gcc (GCC) 10.1.0.
- Riscv64-unknown-linux-gnu-gcc (GCC) 10.1.0.

3. Phiên bản XV6:

- XV6 - UNIX 6th Edition – 2024.
 - 1st Edition (June 14, 2000).
 - ISBN: 1-57398-013-7
 - **Link xv6 repo:** [Link here.](#)

4. Link github repository: [“Click here to go to our github repository”](#)

Yêu cầu 1:
PINGPONG

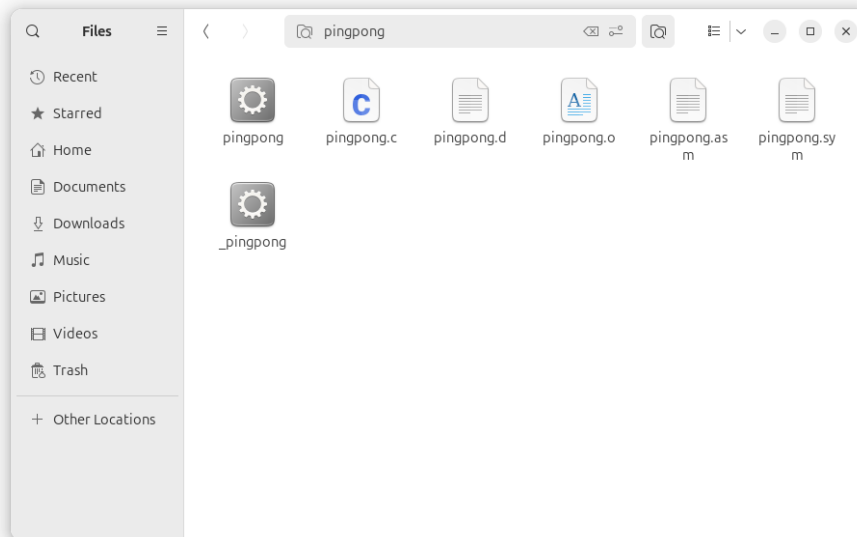


I. Thành viên thực hiện:

- Nguyễn Anh Tường – 22120412.

II. 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 gửi 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 gửi 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 gửi 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.

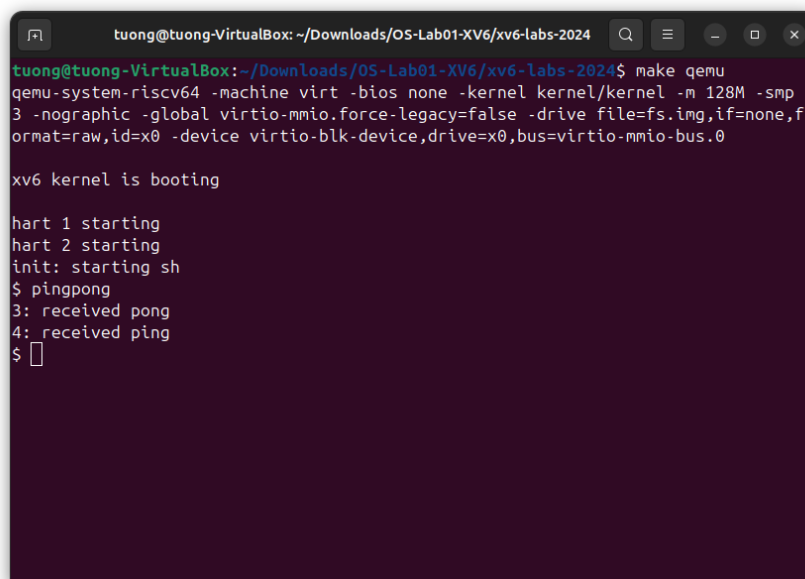
```

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.

III. Kết quả chương trình:



```

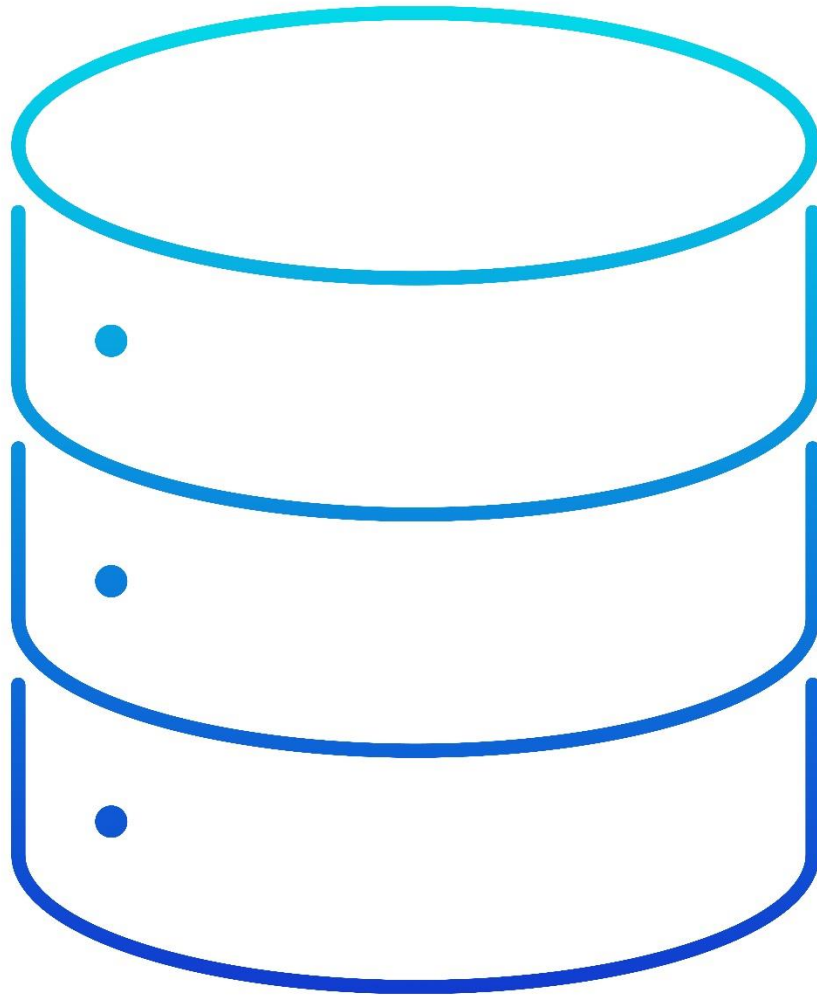
tuong@tuong-VirtualBox: ~/Downloads/OS-Lab01-XV6/xv6-labs-2024
tuong@tuong-VirtualBox:~/Downloads/OS-Lab01-XV6/xv6-labs-2024$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,f
ormat=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ pingpong
3: received pong
4: received ping
$ 

```

Yêu cầu 2:
PRIMES

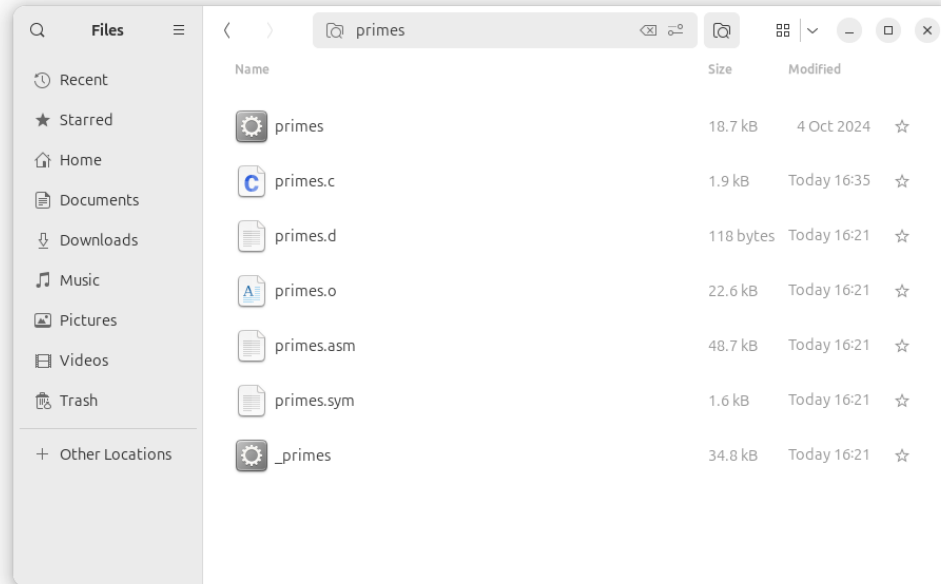


I. Thành viên thực hiện:

Nguyễn Anh Tường – 22120412.

II. 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 gửi 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 kiểm 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: $PID > 0$.

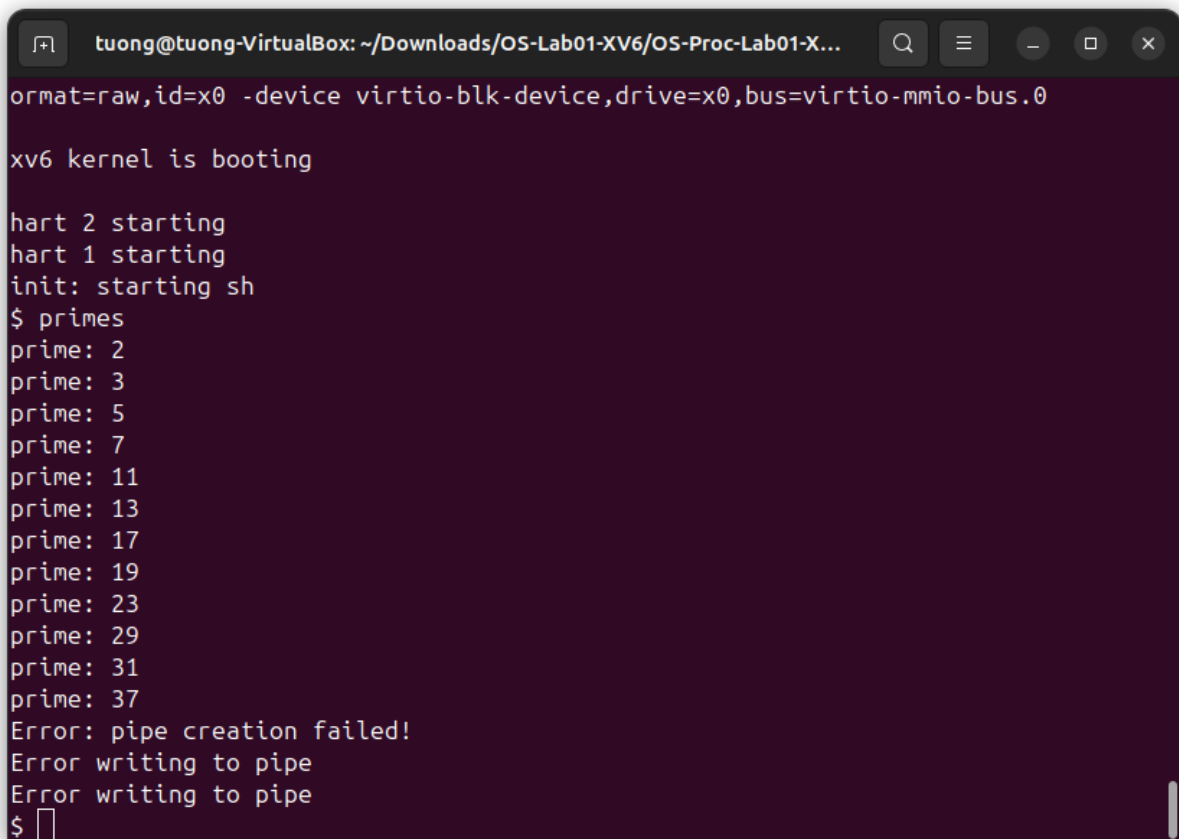
- Tiến hành đọc pipe tham số cho đến khi gặp được một số nguyên tố thì sẽ tiến hành ghi số nguyên tố đó vào pipe tiếp theo và dừng vòng lặp lại.
- Nếu gặp lỗi thì in lỗi ra màn hình, ngược lại chạy tiếp.
- Đóng các pipe hiện tại và đợi cho tiến trình con kết thúc.
- Cuối cùng là thoát hàm.

```
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);
}
```


III. 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 êm xuôi.

a. Vấn đề ban đầu:



```
tuong@tuong-VirtualBox: ~/Downloads/OS-Lab01-XV6/OS-Proc-Lab01-X...  
format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0  
  
xv6 kernel is booting  
  
hart 2 starting  
hart 1 starting  
init: starting sh  
$ primes  
prime: 2  
prime: 3  
prime: 5  
prime: 7  
prime: 11  
prime: 13  
prime: 17  
prime: 19  
prime: 23  
prime: 29  
prime: 31  
prime: 37  
Error: pipe creation failed!  
Error writing to pipe  
Error writing to pipe  
$
```

➔ Chương trình không thể tạo thêm pipe do không đủ tài nguyên.

b. Hướng khắc phục số 1:

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

Thông số ban đầu:

```
param.h
~/Downloads/OS-Lab01-XV6/OS-Proc-Lab01-XV6andUU/xv6-labs-2024/kernel

pingpong.c | proc.h | param.h x

#ifdef LAB_FS
#define NPROC      10 // maximum number of processes
#else
#define NPROC      64 // maximum number of processes (speedsup bigfile)
#endif
#define NCPU       8 // maximum number of CPUs
#define NOFILE     16 // open files per process
#define NFILE      100 // open files per system
#define NINODE     50 // maximum number of active i-nodes
#define NDEV       10 // maximum major device number
#define ROOTDEV    1 // device number of file system root disk
#define MAXARG     32 // max exec arguments
#define MAXOPBLOCKS 10 // max # of blocks any FS op writes
#define LOGSIZE    (MAXOPBLOCKS*3) // max data blocks in on-disk log
#define NBUF       (MAXOPBLOCKS*3) // size of disk block cache
#endif LAB_FS
```

Thông số sau khi tùy chỉnh:

```
param.h
~/Downloads/OS-Lab01-XV6/OS-Proc-Lab01-XV6andUU/xv6-labs-2024/kernel

#ifdef LAB_FS
#define NPROC      10 // maximum number of processes
#else
#define NPROC      64 // maximum number of processes (speedsup bigfile)
#endif
#define NCPU       8 // maximum number of CPUs
#define NOFILE     64 // open files per process
#define NFILE      100 // open files per system
#define NINODE     50 // maximum number of active i-nodes
#define NDEV       10 // maximum major device number
#define ROOTDEV    1 // device number of file system root disk
#define MAXARG     32 // max exec arguments
#define MAXOPBLOCKS 10 // max # of blocks any FS op writes
#define LOGSIZE    (MAXOPBLOCKS*6) // max data blocks in on-disk log
#define NBUF       (MAXOPBLOCKS*6) // size of disk block cache
#define FSSIZE     200000 // size of file system in blocks
#endif LAB_FS
```

c. Hướng khắc phục số 2:

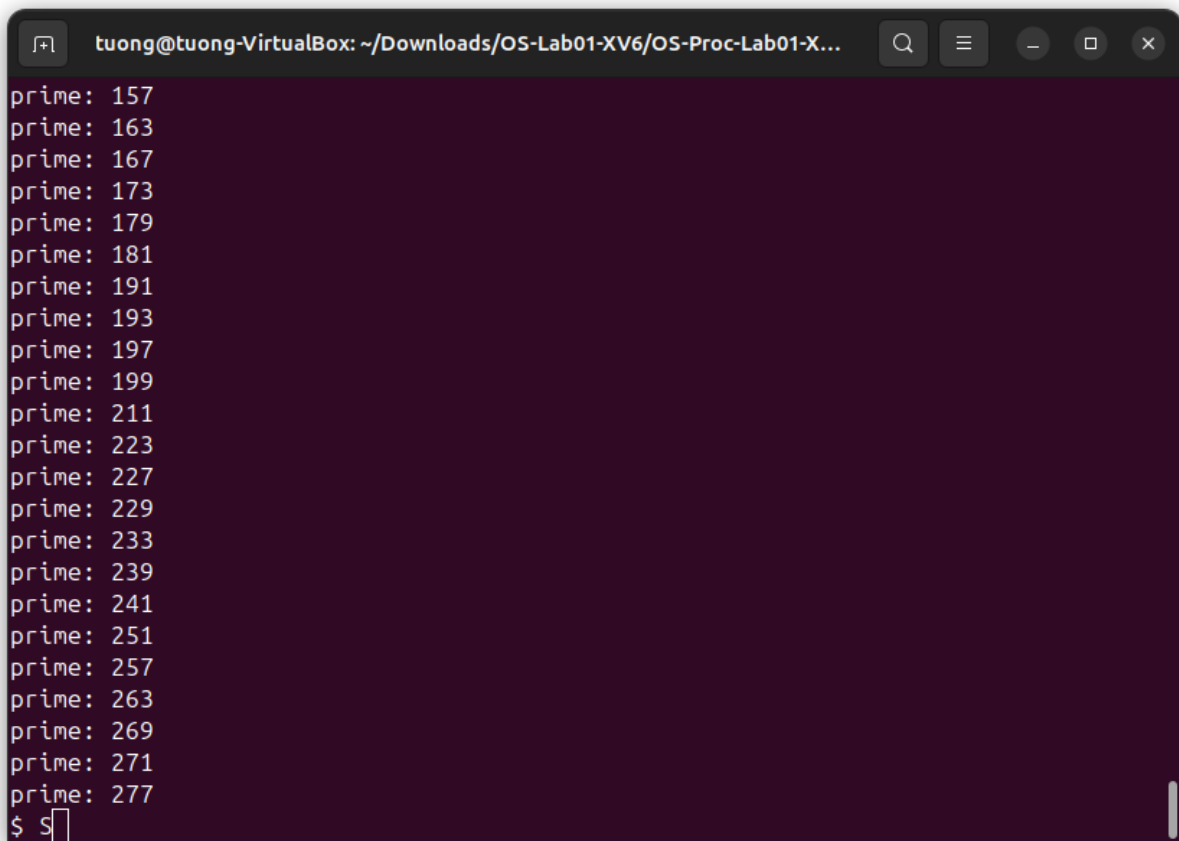
Điều chỉnh lại tính hợp lý của việc đóng, mở của các pipe.

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]);  
}
```

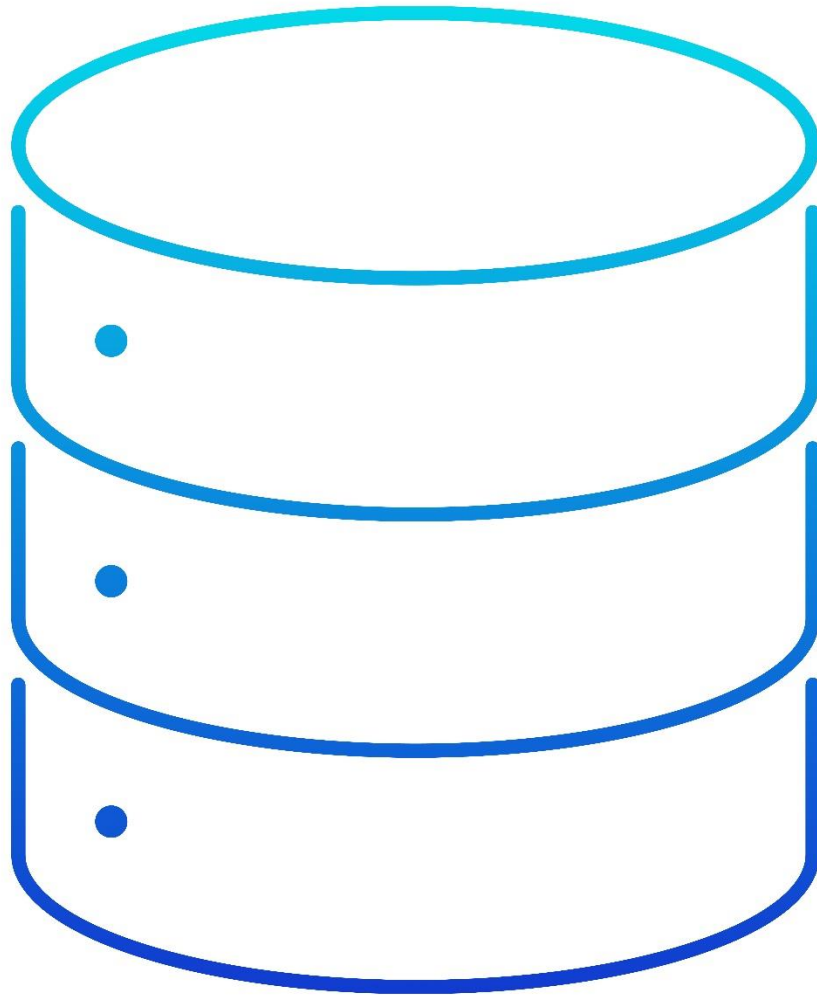
⇒ Ở đây phải đóng curr_pipe, nếu không sẽ bị buffer overflow.

IV. Kết quả chương trình:



```
tuong@tuong-VirtualBox: ~/Downloads/OS-Lab01-XV6/OS-Proc-Lab01-X...  
prime: 157  
prime: 163  
prime: 167  
prime: 173  
prime: 179  
prime: 181  
prime: 191  
prime: 193  
prime: 197  
prime: 199  
prime: 211  
prime: 223  
prime: 227  
prime: 229  
prime: 233  
prime: 239  
prime: 241  
prime: 251  
prime: 257  
prime: 263  
prime: 269  
prime: 271  
prime: 277  
$ S
```

Yêu cầu 3:
FIND



I. Thành viên thực hiện:

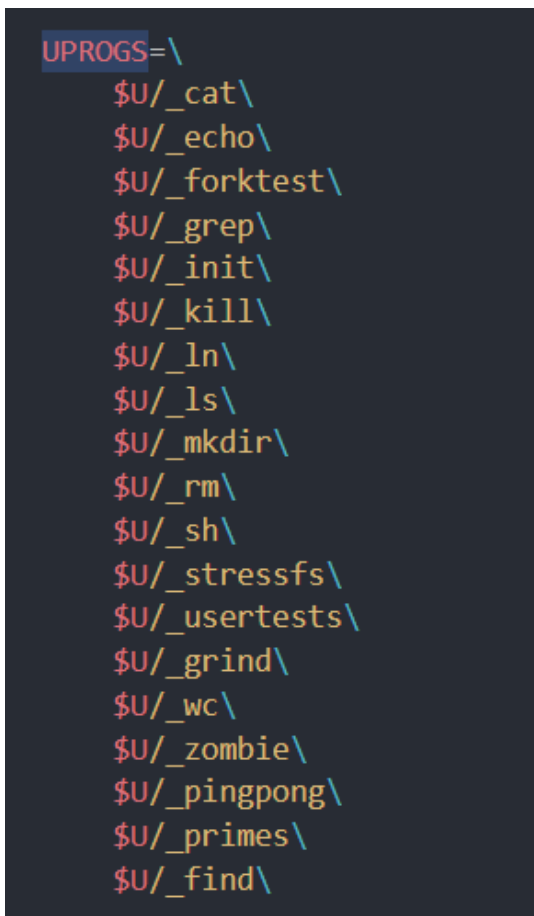
Vũ Hoàng Nhật Trường – 22120398

II. Mô tả thực hiện:

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



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



Bước 3: Lập trình thuật toán:

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

```
$ find . b
```

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+.

```
int
main(int argc, char *argv[])
{
    if (argc < 3)
    {
        printf("Error: Arguments do not fit the command parameter(s).\n");
        exit(0);
    }
}
```

⇒ 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).

```
if((fd = open(currentDir, O_RDONLY)) < 0)
{
    fprintf(2, "ls: cannot open %s\n", fileName);
    return;
}

if(fstat(fd, &st) < 0)
{
    fprintf(2, "ls: cannot stat %s\n", fileName);
    close(fd);
    return;
}
```

```
while(read(fd, &de, sizeof(de)) == sizeof(de))  
{
```

```
$ ls  
.  
..  
README  
xargstest.sh  
cat  
echo  
forktest  
grep  
init  
kill  
ln  
ls  
mkdir  
rm  
sh  
stressfs  
usertests  
grind  
wc  
zombie  
pingpong  
primes  
find  
console
```

- 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:

```
int nameComparison(const char *name1, const char *name2)
{
```

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

```
void
find(char *fileName, char* currentDir) //
```

Hàm dùng để tìm kiếm 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).

```
if(nameComparison(de.name, ".") == 1 || nameComparison(de.name, "..") == 1)
{
    continue;
}
```

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

```
if(nameComparison(fileName, de.name) == 1)
{
    printf("%s\n", buf);
}
```


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

```
if(st.type == T_DIR)
{
    find(fileName, buf);
}
```

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

III. 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.

```
$ mkdir a
$ echo > a/b
$ cd a
$ find b
exec find failed
$
```

- 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.

```
$ find a b
a/b
$
```

IV. Kết quả chương trình

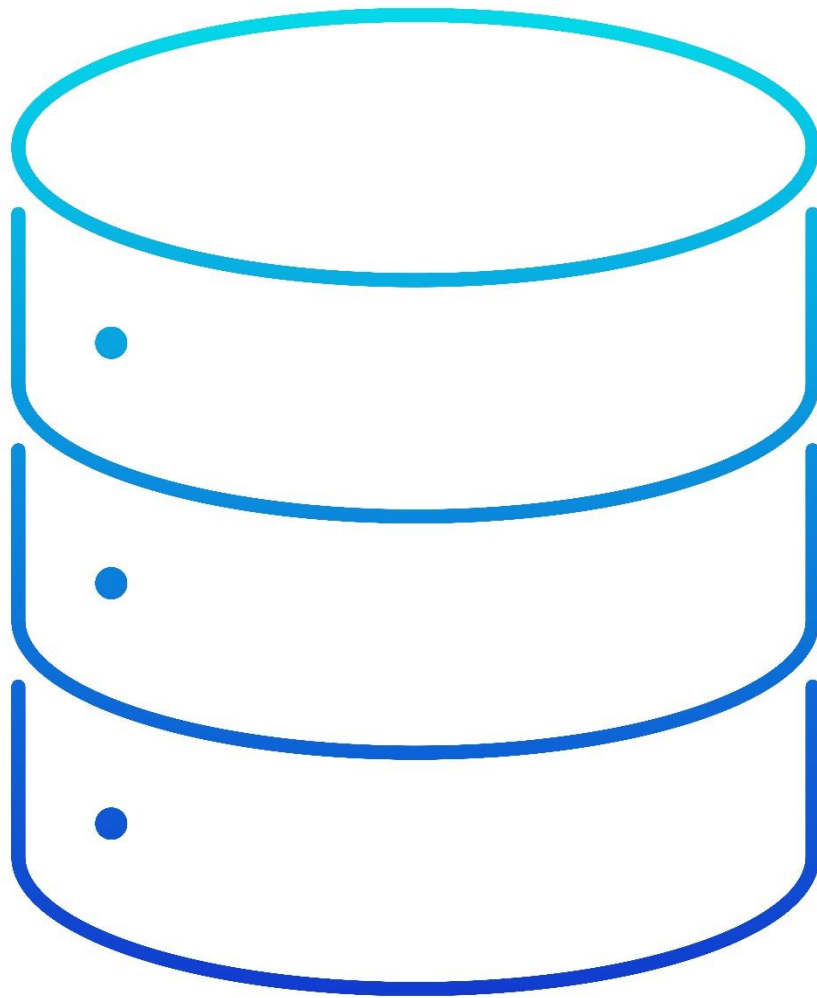
```
$ echo > b
$ mkdir a
$ echo > a/b
$ mkdir a/aa
$ echo > a/aa/b
$ find . b
./b
./a/b
./a/aa/b
```

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

```
$ echo > a/c
$ find . b c
./b
./a/b
./a/aa/b
./a/c
$
```

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

Yêu cầu 4:
XARGS

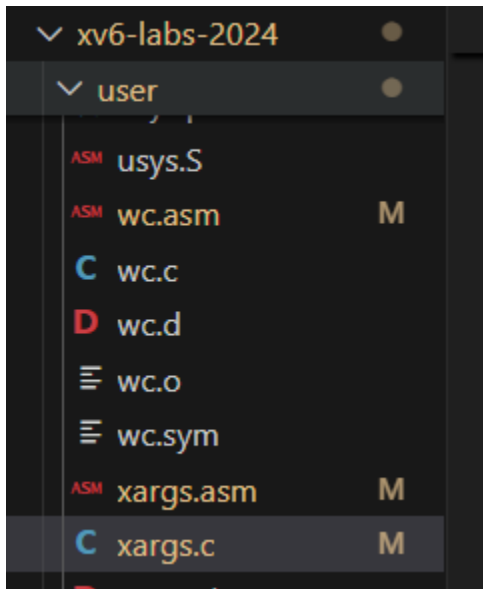


I. Thành viên thực hiện:

Nguyễn Đình Trí, MSSV:22120384

II. 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.

```
UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $U/_grep\
    $U/_init\
    $U/_kill\
    $U/_ln\
    $U/_ls\
    $U/_mkdir\
    $U/_rm\
    $U/_sh\
    $U/_stressfs\
    $U/_usertests\
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_pingpong\
    $U/_primes\
    $U/_find\
    $U/_xargs\
```

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"
#include "kernel/stat.h"
#include "user/user.h"
#include "kernel/param.h"
#include "kernel/fcntl.h"
```

#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

```
// Hàm để đọc lệnh từ stdin
int getcmd(char* buf, int nbuf) {
    memset(buf, 0, nbuf); // Xóa bộ nhớ đệm
    if (read(0, buf, nbuf) <= 0)
    { // Đọc từ stdin
        return -1;
    }
    return 0;
}
```

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.

```
// Mảng chứa các kí tự trắng như dấu cách, tab, xuống dòng
char whitespace[] = " \t\r\n\v";
```

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

```
s = *ps;
```

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

```
if (q)
    *q = s;
```

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

```
while (s < es && !strchr(whitespace, *s))
    s++;
```

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

```
if (eq)
    *eq = s;
```

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

```
*ps = s;
```

```
// Hàm để tách các đối số từ chuỗi
int gettoken(char** ps, char* es, char** q, char** eq) {
    char* s;
    int ret;
    s = *ps;

    while (s < es && strchr(whitespace, *s))
        s++;

    if (q)
        *q = s;

    ret = *s;
    switch (*s) {
        case 0:
            break;

        default:
            ret = 'a';

            while (s < es && !strchr(whitespace, *s))
                s++;

            break;
    }
    if (eq)
        *eq = s;

    while (s < es && strchr(whitespace, *s))
        s++;

    *ps = s;
    return ret;
}
```


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 hơn 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.

```
//  
int main(int argc, char* argv[]) {  
    char* xargs[MAXARG]; // Mảng chứa các đối số  
    int n=0;  
    int flag = 0;  
    int argstart = 0;  
  
    // Xử lý tùy chọn `-n`  
    for (int i = 1; i < argc; i++) {  
        if (strcmp(argv[i], "-n") == 0 && i + 1 < argc)  
        {  
            n = atoi(argv[i + 1]); // Lấy giá trị sau `-n`  
            flag = 1;  
            i++; // Bỏ qua giá trị sau `-n`  
        }  
        else  
        {  
            xargs[argstart++] = argv[i]; // Thêm đối số khác vào xargs  
        }  
    }  
  
    static char buf[MAXARG][512];  
    char* q, *eq;  
    int j = argstart; // Chỉ số bắt đầu để lưu đối số từ input  
    int count = 0; // Đếm số đối số hiện tại  
    int i = 0;
```

```

// Đọc input từ stdin
while (getcmd(buf[i], sizeof(buf[i])) >= 0) {
    char* s = buf[i];
    //print buf[i]

    char* es = s + strlen(s);
    while (gettoken(&s, es, &q, &eq) != 0) {
        *eq = 0; // Kết thúc chuỗi
        xargs[j++] = q; // Lưu đối số vào xargs
        count++;
        i++;

        // Khi đủ số đối số bằng với giá trị `-n`
        if (flag==1 && count == n) {
            xargs[j] = 0; // Thiết lập kết thúc mảng đối số

            // Tạo tiến trình con để thực thi lệnh
            int pid = fork();
            if (pid == 0)
            {
                exec(xargs[0], xargs); // Thực thi lệnh với các đối số
                exit(0); // Thoát khi kết thúc
            }
            wait(0); // Chờ tiến trình con
            // Đặt lại chỉ số `j` và đếm lại số đối số
            j = argstart;
            count = 0;
        }
    }
}

```

```

// Xử lý các đối số còn lại chưa đủ `-n` để thực thi
if (count > 0)
{
    xargs[j] = 0; // Thiết lập kết thúc mảng đối số
    int pid = fork();
    if (pid == 0)
    {
        exec(xargs[0], xargs); // Thực thi lệnh với các đối số còn lại
        exit(0); // Thoát khi kết thúc
    }
    wait(0); // Chờ tiến trình con
}
exit(0);
}

```

Bước 4: Test kết quả

- Yêu cầu 1:

```

$ echo hello too | xargs echo bye
bye hello too
$

```

➔ Kết quả:

```

$ echo hello too | xargs echo bye
bye hello too
$

```

- Yêu cầu 2:

```

$ (echo 1 ; echo 2) | xargs -n 1 echo
1
2
$

```

➔ Kết quả:

```

$ (echo 1 ; echo 2) | xargs -n 1 echo
1
2
$

```

- Yêu cầu 3:

```
$ make qemu
...
init: starting sh
$ sh < xargstest.sh
$ $ $ $ $ $ hello
hello
hello
$ $
```

→ Kết quả:

```
$ sh < xargstest.sh
$ $ $ $ $ $ hello
hello
hello
$ $ █
```