



ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Khoa Công nghệ Thông tin



**BÀI BÁO CÁO NHÓM
Xv6 và Unix Utilities**

MÔN: HỆ ĐIỀU HÀNH

GIẢNG VIÊN: LÊ VIẾT LONG

LỚP: 23CLC02

SINH VIÊN THỰC HIỆN :

1. Trương Hoàng Lâm – MSSV: 23127402
2. Lương Thành Lộc – MSSV: 23127405
3. Bùi Nam Việt – MSSV: 23127516

TP.HCM, ngày 06 tháng 03 năm 2025.

MỤC LỤC

I.	GIỚI THIỆU	3
II.	CÁC THÀNH PHẦN CHÍNH CỦA XV6	3
1.	Tổng quan kiến trúc Xv6.....	3
a.	Chế độ người dùng (User Mode)	3
b.	Hạt nhân (Kernel Mode)	3
c.	Phần cứng (Hardware)	3
2.	Bootloader.....	4
3.	Quản lý tiến trình.....	4
4.	Quản lý bộ nhớ	5
5.	Hệ thống tệp (File System)	5
III.	GIAO TIẾP GIỮA KERNEL VÀ CHƯƠNG TRÌNH TRONG XV6	6
1.	Kiến trúc giao tiếp giữa user-space và kernel-space	6
2.	Thực thi System Call trong XV6.....	6
a.	Ứng dụng gọi system call	6
b.	Chuyển đổi sang kernel mode	6
c.	Xử lý trong kernel	7
d.	Xử lý system call trong kernel	8
e.	Trả kết quả về user-space.....	9
3.	Tổng kết quy trình	9
IV.	CÀI ĐẶT XV6	9
V.	Ý TƯỞNG MÃ NGUỒN CÁC CHƯƠNG TRÌNH	11
1.	Chương trình Sleep	11
2.	Chương trình pingpong	11
3.	Chương trình Primes	12
4.	Chương trình Find	13
5.	Chương trình xargs	16
5.1	Ý tưởng chương trình	16
5.2	Phân tích mã nguồn	16
VI.	TÀI LIỆU THAM KHẢO	18
VII.	PHÂN CÔNG CÔNG VIỆC	18

I. GIỚI THIỆU

Hệ điều hành đóng vai trò quản lý và trừu tượng hóa phần cứng, giúp nhiều chương trình có thể chạy đồng thời và cung cấp các dịch vụ hữu ích hơn so với phần cứng đơn thuần. Một trong những hệ điều hành đơn giản nhưng hiệu quả để nghiên cứu là **Xv6**, phiên bản mô phỏng Unix Sixth Edition, được thiết kế phục vụ giảng dạy về hệ điều hành.

Xv6 có cấu trúc truyền thống với **kernel** làm trung tâm, cung cấp dịch vụ cho các chương trình thông qua **hệ thống gọi hệ thống (system calls)**. Mỗi tiến trình chạy trong **user space**, khi cần truy cập tài nguyên hệ thống, sẽ gọi system call để chuyển sang **kernel space**, nơi kernel xử lý yêu cầu và trả về kết quả. Hệ điều hành đảm bảo bảo vệ bộ nhớ giữa các tiến trình bằng cơ chế phân quyền của CPU.

Hệ thống gọi hệ thống trong Xv6 được thiết kế đơn giản nhưng linh hoạt, tương tự Unix, giúp lập trình viên có thể tận dụng để phát triển nhiều ứng dụng khác nhau. Các thành phần quan trọng của Xv6 bao gồm **quản lý tiến trình, bộ nhớ, hệ thống tập tin, file descriptor và pipe**. Đặc biệt, shell của Xv6 minh họa cách các system call này được sử dụng để xây dựng môi trường dòng lệnh, từ đó cho thấy sự linh hoạt của giao diện hệ thống.

Báo cáo này sẽ trình bày tổng quan về **Xv6**, từ cấu trúc, nguyên lý hoạt động đến phân tích mã nguồn. Qua đó, người đọc sẽ có cái nhìn rõ hơn về cách một hệ điều hành hoạt động ở cấp thấp và làm nền tảng để nghiên cứu các hệ điều hành hiện đại như Linux, macOS hay Windows.

II. CÁC THÀNH PHẦN CHÍNH CỦA XV6

1. Tổng quan kiến trúc Xv6

Hệ điều hành Xv6 có thể chia thành ba phần chính:

a. Chế độ người dùng (User Mode)

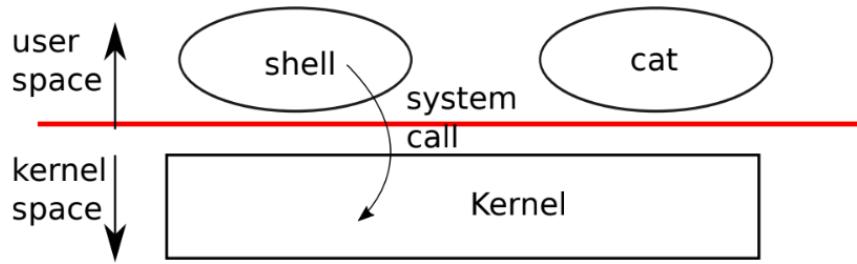
- Chạy chương trình người dùng (vd: Shell, ls, cat,...)
- Gọi hệ thống (System Calls) để tương tác với kernel

b. Hạt nhân (Kernel Mode)

- Quản lý tiến trình (**Process Management**)
- Quản lý bộ nhớ (**Memory Management**)
- Hệ thống tập tin (**File System**)
- Điều khiển thiết bị (**Device Drivers**)

c. Phần cứng (Hardware)

- CPU, bộ nhớ, thiết bị nhập/xuất



2. Bootloader

Hệ điều hành Xv6 sử dụng **bootloader** để khởi động CPU, tải kernel vào bộ nhớ và chuyển quyền điều khiển cho kernel.

Quá trình khởi động của Xv6:

- **Bootloader khởi động trong chế độ Real Mode (16-bit)**
 - Chỉ có thể truy cập **1MB đầu tiên** của RAM
 - Chạy mã khởi động từ sector đầu tiên của ổ đĩa
- **Chuyển CPU sang chế độ Protected Mode (32-bit)**
 - Cho phép truy cập toàn bộ bộ nhớ RAM
 - Sử dụng **bảng phân trang (paging)**
- **Tải kernel vào bộ nhớ**
 - Bootloader đọc file `kernel` từ ổ đĩa vào RAM
- **Chuyển quyền điều khiển cho kernel**

3. Quản lý tiến trình

Tiến trình (**Process**) là chương trình đang thực thi. Xv6 hỗ trợ **đa nhiệm (multitasking)** bằng cách sử dụng **Round Robin Scheduler**. Mỗi tiến trình có không gian địa chỉ riêng bao gồm 3 phần chính:

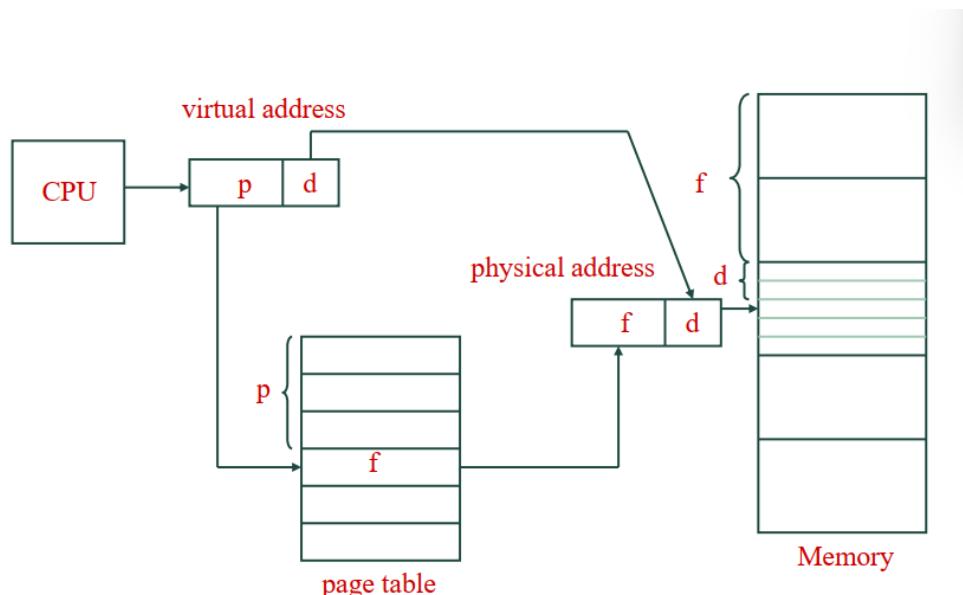
- Instruction(Mã lệnh) thực hiện quá trình tính toán của chương trình
- Data(Dữ liệu) là các biến mà quá trình tính toán tác động lên, stack sắp xếp các lệnh gọi thủ tục của chương trình. Khi một tiến trình gọi system call, bộ xử lý sẽ chuyển từ user mode sang kernel mode. Kernel thực hiện dịch vụ, sau đó trả kết quả về user mode.
- Xv6 time-shares processes: khi 1 tiến trình không thực thi xv6 sẽ lưu thanh ghi CPU của tiến trình và sẽ khôi phục lại khi chung chạy ở tiến trình tiếp theo.

❖ Cách tạo tiến trình trong Xv6

- **fork()**: Tạo tiến trình mới
- **exec()**: Nạp chương trình mới vào tiến trình
- **exit()**: Kết thúc tiến trình
- **wait()**: Đợi tiến trình con kết thúc

4. Quản lý bộ nhớ

- Xv6 sử dụng **phân trang (paging)** để quản lý bộ nhớ.
- Xv6 sử dụng **địa chỉ ảo (Virtual Addressing)** với **Page Table** để ánh xạ địa chỉ logic sang địa chỉ vật lý.



5. Hệ thống tệp (File System)

Xv6 sử dụng **hệ thống tệp dạng inode** để lưu trữ dữ liệu trên ổ đĩa. Mỗi inode giữ metadata của 1 file bao gồm kiểu (file, thư mục, thiết bị), chiều dài của file, vị trí chứa nội dung của tập tin trên đĩa, và số lượng liên kết đến 1 file. Các thư mục được tổ chức thành 1 cây.

Các system call liên quan

System Call	Chức năng
open()	Mở file
read()	Đọc file
write()	Ghi file
close()	Đóng file
unlink()	Xóa file

III. GIAO TIẾP GIỮA KERNEL VÀ CHƯƠNG TRÌNH TRONG XV6

Trong XV6, user-space không thể trực tiếp truy cập vào kernel-space do cơ chế bảo vệ của hệ thống. Vì vậy, **cách duy nhất để chương trình giao tiếp với kernel là thông qua system calls.**

1. Kiến trúc giao tiếp giữa user-space và kernel-space

Cơ chế này hoạt động dựa trên:

- **Trap (interrupt 64 - int \$64)** để chuyển quyền điều khiển từ user-space vào kernel-space.
- **Thanh ghi** để truyền tham số từ chương trình đến kernel.
- **Bảng ánh xạ system call** trong kernel để gọi hàm xử lý phù hợp.
- Quá trình giao tiếp giữa user-space và kernel-space diễn ra qua các bước chính sau:
 - **Chương trình user-space gọi system call** (ví dụ: write, fork, exit, v.v.).
 - **Chuyển vào kernel mode**: Bộ xử lý chuyển sang chế độ kernel và gọi hàm xử lý system call tương ứng.
 - **Kernel xử lý system call**, đọc tham số từ thanh ghi và gọi hàm xử lý tương ứng.
 - **Chuyển lại user mode**: Kernel trả kết quả về chương trình người dùng thông qua thanh ghi và tiếp tục thực thi.

2. Thực thi System Call trong XV6

Mỗi system call trong XV6 được xử lý thông qua nhiều lớp abstraction:

a. Ứng dụng gọi system call

Trong user-space, các system calls được gọi thông qua các wrapper functions trong thư viện user.h.
Ví dụ:

```
#include "user.h"

int main() {
    printf(1, "Hello, XV6!\n"); // Gọi write() để in ra màn hình
    exit();                  // Gọi exit() để kết thúc chương trình
}
```

Hàm printf() thực tế gọi write(), là một system call của XV6.

b. Chuyển đổi sang kernel mode

Các system call được định nghĩa trong user.h như:

```
int fork(void); int exit(void) __attribute__((noreturn));
int write(int, const void*, int);
```

Các lời gọi này thực tế là các wrapper gọi vào interrupt để chuyển điều khiển vào kernel.

Ví dụ, *syscall.h* ánh xạ các system call thành các số nguyên:

```
#define SYS_fork 1
#define SYS_exit 2
#define SYS_write 16
```

Trong usys.S, các system call được thực hiện bằng **instruction trap**:

```
.globl sys_exit
sys_exit:
    movl $SYS_exit, %eax // Đặt số ID của system call vào thanh ghi EAX
    int $64                // Gọi interrupt 64 để vào kernel
    ret                   // Trả về kết quả từ kernel
```

Lệnh int \$64 gây gián đoạn, đẩy chương trình vào kernel.

Khi int \$64 xảy ra, CPU chuyển vào kernel mode, thực hiện trap().

Hàm trap() được định nghĩa trong trap.c:

```
void trap(struct trapframe *tf) {
    if (tf->trapno == T_SYSCALL) {
        if (proc->killed) exit();
        proc->tf = tf;
        syscall(); // Xử lý system call
        if (proc->killed) exit();
        return;
    }
    // Xử lý các trường hợp khác như page fault, timer interrupt,
    v.v.
}
```

tf->trapno == T_SYSCALL kiểm tra xem có phải system call không.

Nếu đúng, hàm *syscall()* sẽ được gọi để xử lý system call.

c. Xử lý trong kernel

Trong file trap.c, kernel nhận system call và gọi *syscall()*. Hàm này kiểm tra số ID của system call, sau đó gọi hàm tương ứng:

```
void syscall(void) {
    int num = proc->tf->eax; // Lấy số ID của
    system call từ thanh ghi eax
```

```

    if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        proc->tf->eax = syscalls[num](); // Gọi hàm xử lý
tương ứng
    } else {
        cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
        proc->tf->eax = -1;
    }
}

```

Danh sách các hàm xử lý system call được định nghĩa trong *sysproc.c* hoặc *sysfile.c*.
num = *proc->tf->eax* lấy số ID của system call (do user-space đã đặt trong thanh ghi EAX).
Dùng bảng *syscalls[]* để ánh xạ số ID đến hàm xử lý tương ứng.

Bảng ánh xạ này được định nghĩa trong *syscall.c*:

```

static int (*syscalls[]) (void) = {
    [SYS_fork]  sys_fork,
    [SYS_exit]   sys_exit,
    [SYS_write]  sys_write,
};

```

Ví dụ, với system call *write()*, kernel sẽ gọi *sys_write()*.

d. Xử lý system call trong kernel

Hàm *sys_write()* được định nghĩa trong *sysfile.c*:

```

int sys_write(void) {
    int fd;
    char *buf;
    int n;
    if (argint(0, &fd) < 0 || argptr(1, &buf, n) < 0 ||
argint(2, &n) < 0)
        return -1;
    return filewrite(proc->ofile[fd], buf, n);
}

```

argint(0, &fd), *argptr(1, &buf, n)*, *argint(2, &n)* lấy các tham số từ user-space.

filewrite() thực hiện ghi dữ liệu vào file.

Kết quả sẽ được trả về thanh ghi EAX.

e. Trả kết quả về user-space

Sau khi `sys_write()` hoàn thành:

Giá trị trả về được đặt trong `proc->tf->eax`.

`syscall()` kết thúc, điều khiển quay lại `trap()`.

CPU khôi phục trạng thái user-mode và tiếp tục thực thi chương trình.

3. Tổng kết quy trình

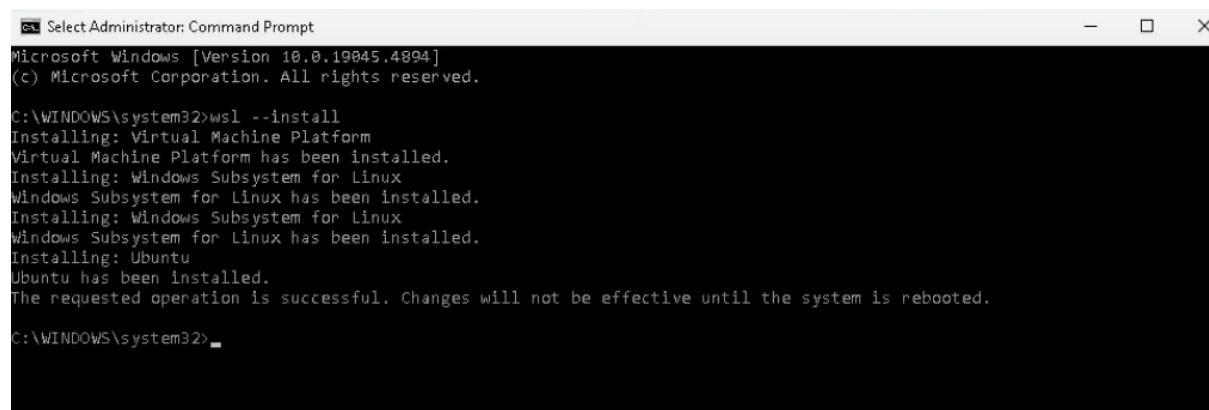
Tóm lại, giao tiếp giữa user-space và kernel trong XV6 thông qua system call diễn ra như sau:

Bước	Hoạt động
1	Chương trình gọi <code>write()</code> .
2	usys.S đặt ID system call vào EAX và gọi <code>int \$64</code> .
3	<code>trap()</code> bắt interrupt và gọi <code>syscall()</code> .
4	<code>syscall()</code> tìm system call tương ứng và gọi <code>sys_write()</code> .
5	<code>sys_write()</code> thực thi ghi file.
6	Giá trị trả về được đặt vào EAX.
7	<code>trap()</code> khôi phục user-space và tiếp tục thực thi.

IV. CÀI ĐẶT XV6

- Bước 1 : Kiểm tra và cập nhật WSL

Mở **PowerShell** với quyền **Administrator** và chạy: `wsl -install`



```
Microsoft Windows [Version 10.0.19045.4894]
(c) Microsoft Corporation. All rights reserved.

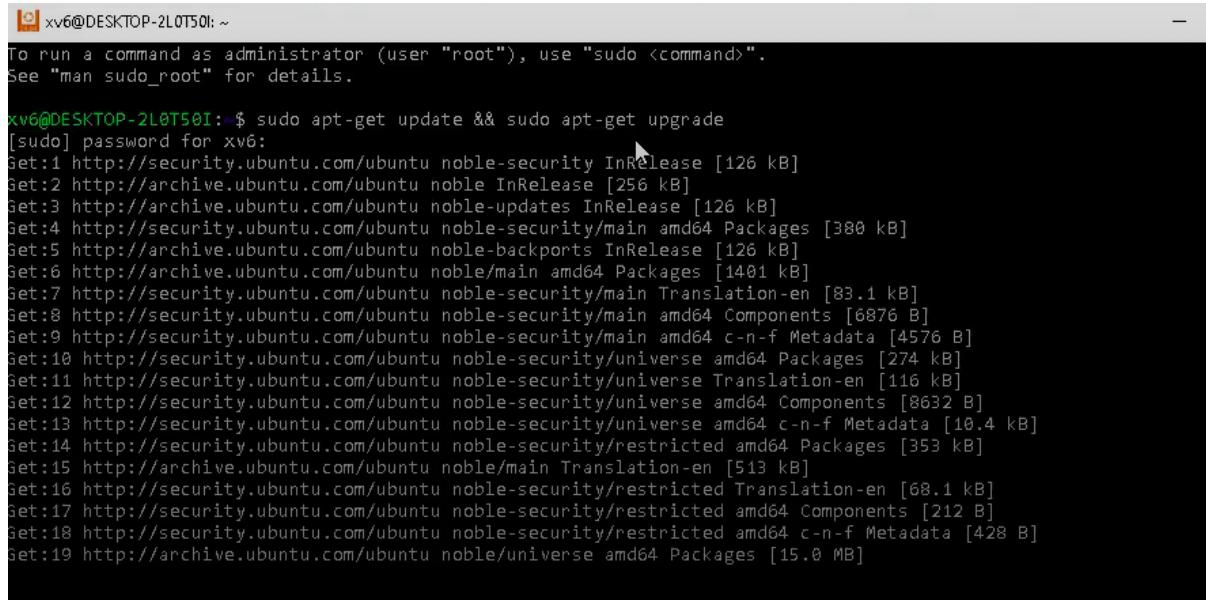
C:\WINDOWS\system32>wsl --install
Installing: Virtual Machine Platform
Virtual Machine Platform has been installed.
Installing: Windows Subsystem for Linux
Windows Subsystem for Linux has been installed.
Installing: Windows Subsystem for Linux
Windows Subsystem for Linux has been installed.
Installing: Ubuntu
Ubuntu has been installed.
The requested operation is successful. Changes will not be effective until the system is rebooted.

C:\WINDOWS\system32>
```

- Bước 2: Cài đặt các gói cần thiết trong WSL

Mở Ubuntu trên WSL và chạy:

```
sudo apt-get update && sudo apt-get upgrade
```



```
xv6@DESKTOP-2L0T50I:~$ sudo apt-get update && sudo apt-get upgrade
[sudo] password for xv6:
Get:1 http://security.ubuntu.com/ubuntu noble-security InRelease [126 kB]
Get:2 http://archive.ubuntu.com/ubuntu noble InRelease [256 kB]
Get:3 http://archive.ubuntu.com/ubuntu noble-updates InRelease [126 kB]
Get:4 http://security.ubuntu.com/ubuntu noble-security/main amd64 Packages [380 kB]
Get:5 http://archive.ubuntu.com/ubuntu noble-backports InRelease [126 kB]
Get:6 http://archive.ubuntu.com/ubuntu noble/main amd64 Packages [1401 kB]
Get:7 http://security.ubuntu.com/ubuntu noble-security/main Translation-en [83.1 kB]
Get:8 http://security.ubuntu.com/ubuntu noble-security/main amd64 Components [6876 B]
Get:9 http://security.ubuntu.com/ubuntu noble-security/main amd64 c-n-f Metadata [4576 B]
Get:10 http://security.ubuntu.com/ubuntu noble-security/universe amd64 Packages [274 kB]
Get:11 http://security.ubuntu.com/ubuntu noble-security/universe Translation-en [116 kB]
Get:12 http://security.ubuntu.com/ubuntu noble-security/universe amd64 Components [8632 B]
Get:13 http://security.ubuntu.com/ubuntu noble-security/universe amd64 c-n-f Metadata [18.4 kB]
Get:14 http://security.ubuntu.com/ubuntu noble-security/restricted amd64 Packages [353 kB]
Get:15 http://archive.ubuntu.com/ubuntu noble/main Translation-en [513 kB]
Get:16 http://security.ubuntu.com/ubuntu noble-security/restricted Translation-en [68.1 kB]
Get:17 http://security.ubuntu.com/ubuntu noble-security/restricted amd64 Components [212 B]
Get:18 http://security.ubuntu.com/ubuntu noble-security/restricted amd64 c-n-f Metadata [428 B]
Get:19 http://archive.ubuntu.com/ubuntu noble/universe amd64 Packages [15.0 MB]
```

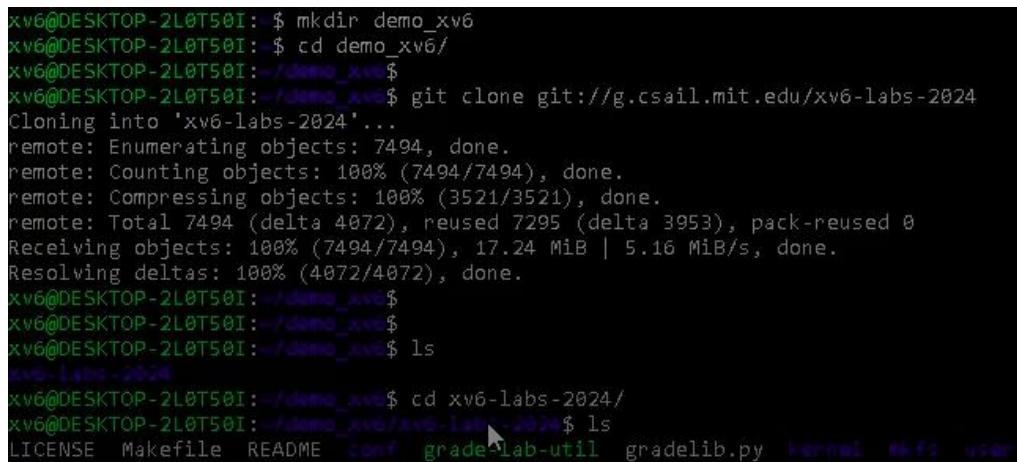
Tương tự với lệnh bên dưới:

```
sudo apt install build-essential qemu-system-i386 gdb-multiarch git -y
```

- Bước 3: Clone Source Code Xv6

```
git clone https://github.com/mit-pdos/xv6-public.git
```

```
cd xv6-public
```



```
xv6@DESKTOP-2L0T50I:~$ mkdir demo_xv6
xv6@DESKTOP-2L0T50I:~$ cd demo_xv6/
xv6@DESKTOP-2L0T50I:/demo_xv6$ git clone git://g.csail.mit.edu/xv6-labs-2024
Cloning into 'xv6-labs-2024'...
remote: Enumerating objects: 7494, done.
remote: Counting objects: 100% (7494/7494), done.
remote: Compressing objects: 100% (3521/3521), done.
remote: Total 7494 (delta 4072), reused 7295 (delta 3953), pack-reused 0
Receiving objects: 100% (7494/7494), 17.24 MiB | 5.16 MiB/s, done.
Resolving deltas: 100% (4072/4072), done.
xv6@DESKTOP-2L0T50I:/demo_xv6$ 
xv6@DESKTOP-2L0T50I:/demo_xv6$ 
xv6@DESKTOP-2L0T50I:/demo_xv6$ ls
xv6-labs-2024
xv6@DESKTOP-2L0T50I:/demo_xv6$ cd xv6-labs-2024/
xv6@DESKTOP-2L0T50I:/demo_xv6/xv6-labs-2024$ ls
LICENSE  Makefile  README  config  grade-lab-util  gradelib.py  kernel  mfs  user
```

- Bước 4: Build Xv6 trên Visual Studio Code

- o Cài đặt Entension WSL
- o Nhấn vào nút >< bên dưới góc trái màn hình VSC để **Connect to WSL**
- o Nhấn File -> Open Folder để chọn thư mục xv6-labs-2024 vừa cài đặt

- Chọn View -> Terminal gõ lệnh `make clean` hoặc `make` để build Xv6

```
riscv64-linux-gnu-ld: warning: kernel/kernel has a LOAD segment with RWX permissions
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
xv6@DESKTOP-2L0T50I:~/demo_xv6/xv6-labs-2024$ make qemu
gcc -DSOL_UTIL -DLAB_UTIL -Werror -Wall -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_UTIL -DLAB_UTIL -MD -mcmov
-fno-builtin-memset -fno-builtin-memmove -fno-builtin-memcmp -fno-builtin-log -fno-builtin-bzero -fno-b
-e -fno-builtin-memcpy -fno-main -fno-builtin-printf -fno-builtin-fprintf -fno-builtin-vprintf -I. -fno-stack-p
perl user/usys.pl > user/usys.S
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_UTIL -DLAB_UTIL -MD -mcmov
-fno-builtin-memset -fno-builtin-memmove -fno-builtin-memcmp -fno-builtin-log -fno-builtin-bzero -fno-b
-e -fno-builtin-memcpy -fno-main -fno-builtin-printf -fno-builtin-fprintf -fno-builtin-vprintf -I. -fno-stack-p
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_UTIL -DLAB_UTIL -MD -mcmov
-fno-builtin-memset -fno-builtin-memmove -fno-builtin-memcmp -fno-builtin-log -fno-builtin-bzero -fno-b
-e -fno-builtin-memcpy -fno-main -fno-builtin-printf -fno-builtin-fprintf -fno-builtin-vprintf -I. -fno-stack-p

```

Nếu build thành công, bạn sẽ thấy các file như `kernel` và `fs.img` được tạo ra.

V. Ý TƯỞNG MÃ NGUỒN CÁC CHƯƠNG TRÌNH

1. Chương trình Sleep

- ***Yêu cầu bài tập:*** cho phép người dùng dừng hệ thống trong một khoảng thời gian
- ***Thuật toán:*** kiểm tra các đối số nhận được từ command line nếu argc (số đối số) khác 2 thì gọi system call exit(1) để thông báo chương trình bị lỗi, nếu bằng hai chuyển đổi số sang số nguyên nếu bé hơn hoặc bằng 0 gọi system call exit(1) để báo lỗi, ngược lại gọi sleep(time) để và gọi exit(0)

2. Chương trình pingpong

- ***Chức năng của chương trình:***

Chương trình này minh họa cơ chế **giao tiếp liên tiến trình (IPC)** trong Xv6 bằng **pipe**. Cụ thể:

- + **Tạo một đường ống (pipe)** để truyền dữ liệu giữa tiến trình cha và tiến trình con.
- + **Tiến trình cha gửi dữ liệu (ping)** đến tiến trình con.
- + **Tiến trình con nhận dữ liệu và phản hồi (pong)** về cho tiến trình cha.
- + **Cả hai tiến trình in ra thông báo tương ứng.**

- ***Phân tích:***

- + **Bước 1: Tạo đường ống pipe**
 - ❖ **Cấu trúc pipe**
 - `fd[0] (read end)`: Đọc dữ liệu từ pipe.
 - `fd[1] (write end)`: Ghi dữ liệu vào pipe.
- + **Bước 2: Tiến trình con thực hiện "ping-pong"**
 - Tiến trình con chờ **đọc dữ liệu** từ pipe (do cha gửi).
 - In ra màn hình: "received ping".
 - Gửi phản hồi "pong" về cho tiến trình cha qua `pipe(fd[1])`.
 - Kết thúc bằng `exit(0)`.
- + **Bước 3: Tiến trình cha gửi và nhận dữ liệu**
 - Tiến trình cha gửi "ping" cho con qua `fd[1]`.
 - Chờ tiến trình con hoàn thành (`wait(0)`).

- Nhận phản hồi "pong" từ con qua `fd[0]`.
- In ra màn hình "received pong".
- Thoát bằng `exit(0)`.

3. Chương trình Primes

– *Chức năng của chương trình:*

Chương trình này cài đặt thuật toán **sàng số nguyên tố (Eratosthenes)** trong Xv6 bằng cách sử dụng **pipeline (pipe)** và **tiến trình con (fork)**.

- **Dòng số tự nhiên** được tạo từ $2 \rightarrow 280$ thông qua `generate_natural()`.
- **Mỗi tiến trình cha nhận một số nguyên tố** và tạo một tiến trình con để lọc các bội số của số đó.
- **Kết quả cuối cùng là dãy số nguyên tố** được in ra.

– *Phân tích:*

+ *Bước 1: Hàm main()*

FUNCTION main () :

*Gọi generate_natural() để nhận danh sách số tự nhiên từ pipe
WHILE TRUE:*

Đọc số từ pipe vào prime

IF đọc hết dữ liệu THEN break

In ra "prime"

Gọi prime_filter() để tạo một tiến trình mới lọc số

Thoát chương trình

- **Nhận luồng số tự nhiên** từ `generate_natural()`.

- **Đọc số đầu tiên**, in ra màn hình (đây là số nguyên tố đầu tiên).

- **Gọi prime_filter()** để tạo một tiến trình con **lọc bội số** của số nguyên tố vừa tìm được.

- **Lặp lại quá trình** trên với phần dữ liệu còn lại.

+ *Bước 2: Hàm generate_natural() - Tạo số tự nhiên từ 2 → 280*

FUNCTION generate_natural () :

Tạo một pipe fd

fork một tiến trình con

IF là tiến trình con:

FOR i từ 2 đến 280:

Ghi i vào fd[1] (pipe)

Đóng fd[1]

Thoát tiến trình con

Đóng fd[1] của tiến trình cha

RETURN fd[0] (đầu đọc của pipe)

- **Tạo pipe** để truyền dữ liệu giữa các tiến trình.

- **Tiến trình con (`fork() == 0`) ghi** các số từ $2 \rightarrow 280$ vào pipe.

- **Tiến trình cha chỉ đọc** từ pipe, không ghi thêm dữ liệu.

+ *Bước 3: Hàm prime_filter() - Lọc bội số của số nguyên tố*

FUNCTION prime_filter(in_fd, prime) :

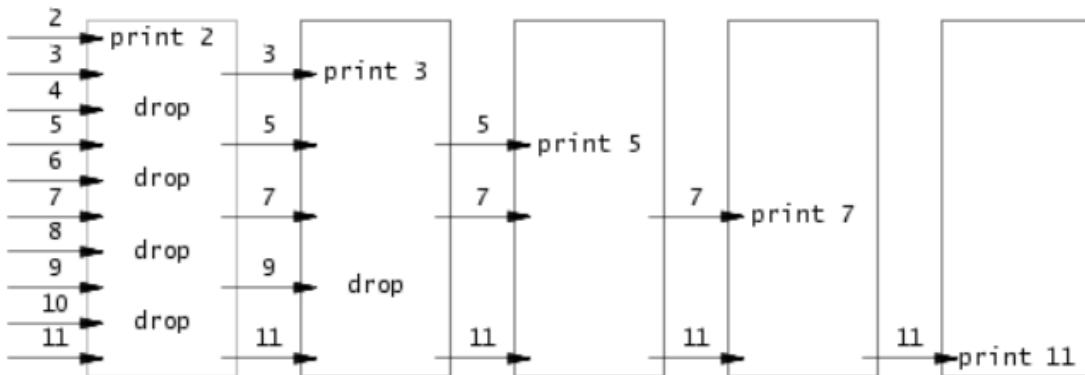
Tạo một pipe fd

fork một tiến trình con

```

IF là tiến trình con:
    WHILE đọc số từ in_fd vào num:
        IF num không phải bội số của prime:
            Ghi num vào fd[1]
        Đóng in_fd và fd[1]
        Thoát tiến trình con
    Đóng in_fd và fd[1] của tiến trình cha
    RETURN fd[0] (đầu đọc của pipe mới)
    o Nhận vào một số nguyên tố prime và đầu đọc in_fd của pipe.
    o Tạo một tiến trình con để lọc các số không chia hết cho prime.
    o Ghi dữ liệu lọc được vào một pipe mới để tiếp tục xử lý.

```



4. Chương trình Find

- Ý tưởng của lệnh find

Lệnh find trong xv6 được thiết kế để tìm kiếm tệp có tên cụ thể trong một thư mục và các thư mục con. Ý tưởng chính của chương trình là:

- + **Duyệt qua hệ thống tệp đệ quy**, kiểm tra từng thư mục con.
- + **So sánh tên tệp** trong mỗi thư mục với tên cần tìm.
- + **In ra đường dẫn của tệp** nếu tìm thấy.

- Cấu trúc chính của code

Lệnh find được triển khai trong file find.c, có hai phần chính:

- + **Hàm main()**: Xử lý đầu vào, gọi hàm find().
- + **Hàm find()**: Duyệt cây thư mục và tìm kiếm tệp.

4.1 Hàm main()

Hàm **main()** kiểm tra số lượng tham số đầu vào và gọi find().

```

int main(int argc, char* argv[]) {
    if(argc != 3) {
        printf("Usage: find [directory] [filename]\n");
        exit(1);
    }
}

```

```

        }

    find(argv[1], argv[2]);
    exit(0);
}

```

- + Chương trình yêu cầu **2 tham số**: thư mục bắt đầu tìm kiếm và tên tệp cần tìm.
- + Nếu số lượng tham số không đúng, in hướng dẫn và thoát (exit(1)).
- + Gọi hàm find() với thư mục cần tìm và tên tệp.

4.2 Hàm find()

Đây là phần quan trọng nhất của chương trình, thực hiện việc duyệt thư mục.

```

void find(char* path, char* file) {
    char buf[512], *p;
    int fd;
    struct dirent de;
    struct stat st;
void find(char* path, char* file) {
    char buf[512], *p;
    int fd;
    struct dirent de;
    struct stat st;

```

- + **buf[512]** : Bộ đệm để lưu đường dẫn tệp hiện tại.
- + **fd**: File descriptor để mở thư mục.
- + **de**: Biến lưu thông tin từng tệp trong thư mục.
- + **st**: Cấu trúc chứa thông tin về tệp (loại tệp, kích thước, v.v.).

- Mở thư mục

```

if((fd = open(path, 0)) < 0) {
    printf("find: cannot open %s\\n", path);
    return;
}

```

- Mở thư mục cần tìm (**open(path, 0)**).
- Nếu không mở được, in lỗi và thoát khỏi hàm.

- Lấy thông tin về thư mục

```

if(fstat(fd, &st) < 0) {
    printf("find: cannot stat %s\\n", path);
    close(fd);
    return;
}

```

- + **fstat(fd, &st)** lấy thông tin về tệp/thư mục.
- + Nếu thất bại, in lỗi và đóng file descriptor.
 - Kiểm tra nếu không phải thư mục

```

if(st.type != T_DIR) {
    printf("find: %s is not a directory\\n", path);
    close(fd);
    return;
}

```

Nếu không phải thư mục (T_DIR), thoát vì find chỉ duyệt thư mục.

- Đọc từng tệp trong thư mục

```

strcpy(buf, path);
p = buf + strlen(buf);
*p++ = '/';
while(read(fd, &de, sizeof(de)) == sizeof(de)) {
    if(de.inum == 0)
        continue;

```

- + **strcpy(buf, path)**: Sao chép đường dẫn thư mục vào buf.
- + **p = buf + strlen(buf)** : Trỏ đến cuối chuỗi buf.
- + ***p++ = '/'**: Thêm / vào cuối để chuẩn bị nối với tên tệp.
- + **while(read(fd, &de, sizeof(de)) == sizeof(de))** : Đọc từng tệp trong thư mục.
- + **if(de.inum == 0) continue;**: Bỏ qua mục không hợp lệ.
- Kiểm tra nếu tệp khớp với tên cần tìm

```

if(strcmp(de.name, file) == 0) {
    printf("%s/%s\\n", path, de.name);
}

```

- + So sánh tên tệp với file.
- + Nếu trùng, in ra đường dẫn đầy đủ.

- Đệ quy tìm trong thư mục con

```

if(strcmp(de.name, ".") != 0 && strcmp(de.name, "..") != 0) {
    memmove(p, de.name, strlen(de.name));
    p[strlen(de.name)] = 0;
    find(buf, file);
}
close(fd);
}

```

- + **Bỏ qua . và ..** (tránh lặp vô tận).
- + **Nối de.name vào buf để tạo đường dẫn tệp con.**
- + **Gọi find(buf, file) đệ quy để tìm trong thư mục con.**
- + **Đóng file descriptor sau khi duyệt xong.**

4.3 Tóm tắt hoạt động

- Nhận tham số thư mục & tên tệp.
- Mở thư mục và kiểm tra lỗi.
- Duyệt từng tệp/thư mục con.
- Nếu là tệp và khớp tên, in ra.
- Nếu là thư mục, gọi đệ quy để tìm trong thư mục con.
- Lặp lại cho đến khi duyệt hết tất cả thư mục con.

Ví dụ chạy lệnh

Nếu ta chạy:

sh

find / myfile.txt

find() duyệt từ /, kiểm tra tất cả tệp và thư mục con.

Nếu **myfile.txt** tồn tại trong **/home/user/**, chương trình sẽ in:

5. Chương trình xargs

5.1 Ý tưởng chương trình

Chương trình trên là một **phiên bản của xargs trong Xv6**, có nhiệm vụ:

- Đọc đầu vào từ **stdin** (dữ liệu từ pipe hoặc nhập trực tiếp).
- Ghép từng dòng đầu vào với danh sách đối số từ **argv[]**.
- **Tạo tiến trình con** để thực thi lệnh với đối số đã ghép.
- **Chờ tiến trình con hoàn thành**, sau đó lặp lại cho dòng tiếp theo.

5.2 Phân tích mã nguồn

- Hàm **copy_argv()**

```
void copy_argv(char **ori_argv, int argc, char *new_argv, char **argv)
```

- + **Chức năng:** Sao chép danh sách đối số từ `ori_argv` (danh sách cũ) sang `argv` (danh sách mới), sau đó thêm đối số mới `new_argv` vào cuối.
- + **Cách hoạt động:**
 - Duyệt qua `ori_argv`, sao chép từng phần tử vào `argv[]`.
 - Thêm `new_argv` vào `argv[]`.
 - Đặt `NULL` cuối danh sách để đánh dấu kết thúc.
- Hàm `main()`

```
int main(int argc, char *argv[])
```

 - + **Bước 1: Kiểm tra đầu vào**

```
if (argc <= 1) {  
    fprintf(2, "Usage: xargx command [arg ...]\n");  
    exit(1);  
}
```

Nếu không có **lệnh để thực thi**, in hướng dẫn và thoát.
 - + **Bước 2: Đọc dữ liệu từ `stdin` (đầu vào chuẩn)**

```
char param[MAX_ARG_LEN];  
int i = 0;  
char ch;  
int ignore = 0;  
  
while (read(0, &ch, 1) > 0)
```

 - Đọc từng ký tự từ `stdin`.
 - Ghi vào `param[]` để lưu từng dòng.
 - + **Bước 3: Khi gặp ký tự `\n` (xuống dòng), tạo tiến trình mới**

```
if (ch == '\n') {  
    if (ignore) {  
        i = 0;  
        ignore = 0;  
        continue;  
    }  
    param[i] = 0;  
    i = 0;
```

 - Khi gặp `\n`, đánh dấu kết thúc chuỗi.
 - Bỏ qua dòng nếu nó quá dài (`ignore = 1`).

- + **Bước 4: Fork một tiến trình con để thực thi lệnh**

```
int pid = fork();  
if (pid == 0) {
```

- Fork **tiến trình con** để thực thi lệnh.

- Trong tiến trình con:
 - Tạo **mảng cmd_argv**, sao chép danh sách đối số (`argv`) và thêm `param`.
 - Gọi `exec()` để chạy lệnh với đối số mới.

```
if (exec(cmd_argv[0], cmd_argv) < 0) {
    fprintf(2, "exec %s failed\n", cmd_argv[0]);
    exit(1);
}
```

- Nếu `exec()` thất bại, in thông báo lỗi.

+ **Bước 5: Tiến trình cha đợi tiến trình con hoàn thành**

```
else {
    wait((int *)0);
}
```

- Tránh tạo quá nhiều tiến trình cùng lúc.

VI. TÀI LIỆU THAM KHẢO

- [1] <https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf>
 [2] <https://github.com/FarahAbdelmoneim/Xv6-and-Unix-Utilities.git>

VII. PHÂN CÔNG CÔNG VIỆC

Bài tập	Người thực hiện	Mức độ hoàn thành
Sleep	Trương Hoàng Lâm	100%
Pingpong		100%
Primes		100%
Find	Lương Thành Lộc	100%
Xargs	Bùi Nam Việt	100%
Report	Bùi Nam Việt Trương Hoàng Lâm Lương Thành Lộc	100%

----- HẾT -----