

**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN, ĐHQG-HCM**  
**Khoa Công nghệ Thông Tin**



Môn học: Hệ Điều Hành

---

**BÁO CÁO PROJECT 2 – XV6 LABS: SYSTEM CALLS**

---

GV hướng dẫn: Lê Giang Thanh  
Trần Trung Dũng

SV thực hiện:

MSSV	Họ và Tên
21120056	Nguyễn Đặng Tường Duy
21120462	Đỗ Khải Hưng

# Mục lục

0.1	Thông tin môn học . . . . .	2
0.2	Thông tin lab . . . . .	2
<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	xv6 là gì? . . . . .	3
1.2	Lab: System calls . . . . .	3
1.2.1	Mục tiêu . . . . .	3
1.2.2	Nội dung . . . . .	3
1.2.3	Lợi ích . . . . .	3
1.3	Cài đặt môi trường . . . . .	3
<b>2</b>	<b>Exercises</b>	<b>4</b>
2.0	Prerequisite . . . . .	4
2.1	System call tracing (moderate) . . . . .	4
2.1.1	Mô tả đề bài và gợi ý . . . . .	4
2.1.2	Cách làm và Giải thích . . . . .	4
2.1.3	Lưu ý: . . . . .	4
2.2	Sysinfo (moderate) . . . . .	5
2.2.1	Mô tả đề bài và gợi ý . . . . .	5
2.2.2	Cách làm và Giải thích . . . . .	5
2.2.3	Lưu ý: . . . . .	5
<b>3</b>	<b>Kết luận</b>	<b>5</b>
3.1	Kết quả đạt được . . . . .	5
3.2	Lưu ý khi tự chấm điểm . . . . .	6
3.3	Lời kết . . . . .	6

## Thông tin lab và thông tin môn học

### 0.1 Thông tin môn học

- Tên môn học: Hệ điều hành.
- Khoa: Khoa Công nghệ Thông tin, Trường Đại học Khoa học Tự nhiên, ĐHQG-HCM.
- Giảng viên: Trần Trung Dũng, Lê Giang Thanh.
- Lớp: 21 Cử nhân Tài năng.

### 0.2 Thông tin lab

- Lab: System calls
- Người hướng dẫn: Trần Trung Dũng, Lê Giang Thanh.
- Hạn nộp: 14/11/2023.
- Nhóm thực hiện: Nhóm gồm các thành viên:
  1. Nguyễn Đặng Tường Duy – 21120056
  2. Đỗ Khải Hưng – 21120462

# 1 Introduction

## 1.1 xv6 là gì?

Xv6 là một hệ điều hành thời gian thực đơn giản được phát triển dựa trên Unix V6. Xv6 được sử dụng trong nhiều khóa học về hệ điều hành và hệ thống tại các trường đại học, bao gồm cả khóa học "6.828: Operating System Engineering" tại MIT.

Xv6 được viết bằng ngôn ngữ lập trình C và được thiết kế để hoạt động trên kiến trúc máy tính x86. Xv6 có cấu trúc đơn giản nhưng cung cấp một số tính năng quan trọng của một hệ điều hành như quản lý tiến trình, quản lý bộ nhớ ảo, quản lý tệp, và hỗ trợ mạng cơ bản.

Mục tiêu của xv6 là giúp sinh viên hiểu rõ cách hoạt động của một hệ điều hành, từ việc lên lịch thực thi tiến trình đến giao tiếp với các thiết bị và tệp tin. Xv6 cung cấp một môi trường thí nghiệm lý tưởng để nghiên cứu và thực hành các khái niệm hệ điều hành cơ bản.

Ngoài ra, xv6 cũng đi kèm với các bài thực hành, như lab "Lab: system calls" để giúp sinh viên làm quen với hệ điều hành và phát triển kỹ năng lập trình hệ thống.

## 1.2 Lab: System calls

### 1.2.1 Mục tiêu

Lab: System calls là một bài thực hành của khóa học 6.828: Operating System Engineering tại MIT. Mục tiêu của bài thực hành này là giúp sinh viên hiểu rõ hơn về lời gọi hệ thống, một trong những khái niệm quan trọng nhất trong hệ điều hành.

### 1.2.2 Nội dung

Trong bài thực hành này, sinh viên sẽ được yêu cầu:

- Tìm hiểu về cấu trúc và hoạt động của lời gọi hệ thống trong xv6.
- Tìm hiểu về các loại lời gọi hệ thống khác nhau, chẳng hạn như lời gọi hệ thống nhập/xuất, lời gọi hệ thống quản lý bộ nhớ và lời gọi hệ thống xử lý tín hiệu.
- Thực hiện các bài tập thực hành để viết chương trình sử dụng lời gọi hệ thống.

### 1.2.3 Lợi ích

Bài thực hành này sẽ giúp sinh viên:

- Hiểu rõ hơn về cách thức hoạt động của hệ điều hành, hiểu được cách thức hoạt động của các lệnh hệ thống như `read`, `write`, `open`, và `fork`.
- Phát triển kỹ năng lập trình hệ thống, biết được cách viết ra một `system call` trong hệ điều hành là như thế nào.
- Tạo nền tảng vững chắc cho việc học các bài thực hành tiếp theo trong khóa học.

## 1.3 Cài đặt môi trường

Nhóm chúng em sử dụng máy ảo Linux với bản phân phối là Ubuntu, do đó, để cài đặt được hệ điều hành xv6, ta cần làm các bước như sau:

- Trong giao diện của Ubuntu, ta mở Terminal.
- Chạy lệnh: `sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu` để cài đặt các công cụ giúp hệ điều hành xv6 có thể chạy được.
- Ta cũng cần cài đặt Git, bằng cách chạy lệnh: `sudo apt-get install git`.

Chi tiết về cách thiết lập môi trường cho các hệ điều hành khác có thể tham khảo ở: <https://pdos.csail.mit.edu/6.828/2023/tools.html>.

## 2 Exercises

### 2.0 Prerequisite

Trước khi bắt đầu, ta bật Terminal trong folder `xv6-labs-2023` và thực hiện các bước sau:

```
$ git fetch
$ git checkout syscall
$ make clean
```

### 2.1 System call tracing (moderate)

#### 2.1.1 Mô tả đề bài và gợi ý

- **Mô tả đề bài:** Bài tập **tracing** yêu cầu sinh viên viết chương trình để truy vết (**trace**) các **system calls** được chạy trong **process** hiện tại, với tham số đầu vào là **mask** của **system call**.
- **Các gợi ý cho bài toán**(cũng là cách để thêm một **system call** vào hệ thống:
  - Thêm `$U/_trace` vào `UPROGS` trong `Makefile`
  - Hàm `read` sẽ dừng khi tiến trình cha đóng file
  - Thêm prototype cho **system call** `trace` vào `user/user.h` và `user/usys.pl`, và define hằng số cho lời gọi.
  - Viết hàm `sys_trace` ở trong `kernel/syscall.c`.
  - Thay đổi **system call** `fork` để có thể truy vết các tiến trình con.
  - Cuối cùng, sửa hàm gọi các **system call** để có thể in ra kết quả truy vết.

#### 2.1.2 Cách làm và Giải thích

- Ta cần thực hiện các bước đầu để cài đặt một **system call** trong hệ thống, để làm được điều đó ta cần hiểu được luồng hoạt động của các **system call** từ **user mode** đến **system mode**.
  - **Ở phía user:**
    1. Định nghĩa **system call** trong file `user.h`
    2. Thêm một stub vào file `usys.pl`
    3. Thêm một flag trong file `Makefile`
  - **Ở phía kernel:**
    1. Định nghĩa hằng số của **system call** trong file `syscall.h`, sửa biến để mapping các **system calls** trong file `syscall.c`
    2. Trong file `sysproc.c` cài đặt hàm **system call** với tên gọi theo chuẩn `sys_"system call name"`, để phân tách các tác vụ, sau khi đọc dữ liệu từ thanh ghi `a7` cần định nghĩa thêm hàm có tên **system call name** trong `defs.h` và cài đặt trong `proc.c`
- Với **system call** `trace` để có thể biết được **system call** nào cần được truy vết trong **process**, ta cần thay đổi cấu trúc **process** trong file `proc.h`, ta thêm một trường **integer** `trace_mask` là `1<SYS_"system call name"` để khi mỗi lần gọi **system call**, ta có thể check để in ra kết quả `trace` ra màn hình.
- Do vậy, sau khi đọc tham số `trace_mask` từ thanh ghi, ta gán giá trị vào tiến trình hiện tại, đồng thời để có thể in ra kết quả ra màn hình, thì trong hàm `syscall()` ở trong file `syscall.c` ta cần kiểm tra `mask` của **system call** hiện tại có giống với `trace_mask` của tiến trình để ghi ra màn hình.

#### 2.1.3 Lưu ý:

- Với trường hợp `mask` là `INT_MAX` ta mặc định in ra tất cả **systemcall** hiện tại.
- **Syscall** không nhận giá trị tham số truyền vào mà đọc từ thanh ghi `a7` qua hàm `argaddr`.
- Để việc `trace` các **system call** trở nên đầy đủ thì ta cần chỉnh sửa **system call** `fork` để gán các giá trị `trace_mask` đến cả các tiến trình con.

## 2.2 Sysinfo (moderate)

### 2.2.1 Mô tả đề bài và gợi ý

- **Mô tả đề bài:** Viết một chương trình in ra các thông tin của hệ thống gồm các tiến trình không đang ở trạng thái USED và số bytes free memory.
- **Các gợi ý cho bài toán:**
  - Định nghĩa system call ở phía user trong file `user.h`
  - Dùng hàm `copyout` để copy giá trị từ `kernel space` sang `user space`.
  - Viết hàm thu thập các memory chưa được cấp phát trong file `kernel/kalloc.c`.
  - Viết hàm đếm số process trong file `kernel/proc.c`.

### 2.2.2 Cách làm và Giải thích

- Các bước để tạo một system call tương tự như với exercise trước.
- Người dùng truyền tham số là con trỏ trỏ đến địa chỉ của proc được khai báo từ phía người dùng. System call đọc địa chỉ ở dạng `uint64` và truyền vào hàm con ở file `proc.c`.
- Để đếm số bytes memory được giải phóng, ta có thể đọc từ biến struct `kmem` trong file `kalloc.c` có thuộc tính `run` là một `linked list` và ta có thể đếm số bytes được giải phóng bằng cách duyệt đến khi `null`, vì một `page` memory được cấp phát có **4096** bytes nên cần nhân vào để nhận được số bytes chính xác.
- Để đếm số process ở trạng thái nào ta có thể kiểm tra từ mảng các process hiện tại struct `proc proc[NPROC]` và dùng vòng lặp để kiểm tra `state` của từng trạng thái.
- Sau cùng tạo một biến struct `proc` để lưu 2 giá trị trên sau đó dùng `copyout()` để trả dữ liệu về `user space`

### 2.2.3 Lưu ý:

- Khi duyệt cần tạo một biến con trỏ mới để mất liên kết trong linked list ban đầu.
- Cần khai báo tất cả các hàm phụ trong file `defs.h`.

## 3 Kết luận

### 3.1 Kết quả đạt được

Bằng các cách làm trên, chúng em đã triển khai các chương trình và dùng lệnh chấm điểm `make grade`. Kết quả đạt được đều pass các test của hệ thống:

```

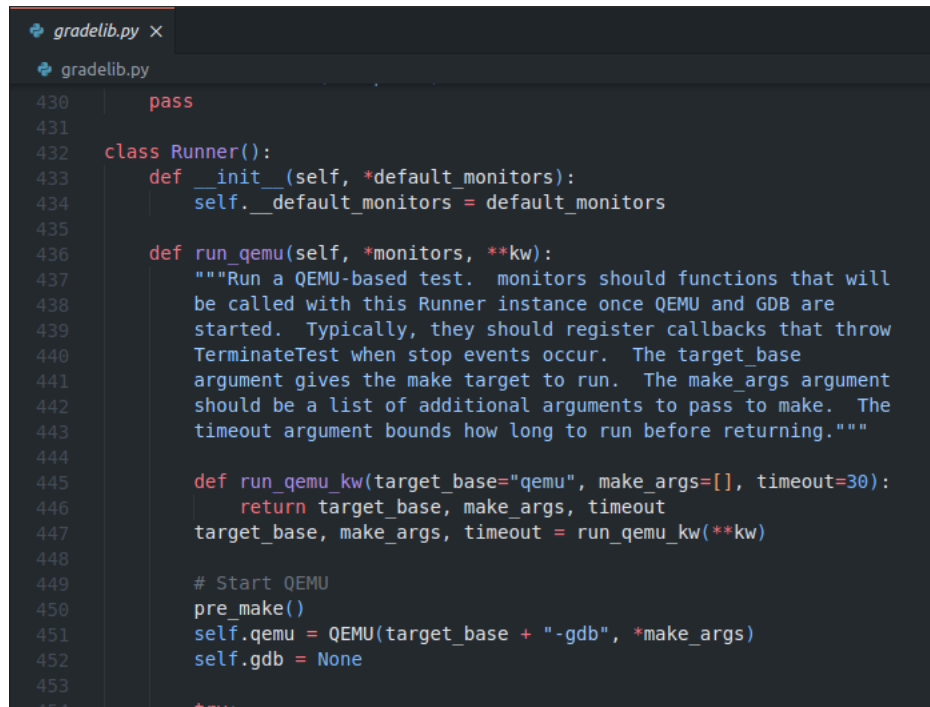
Cannot read answers-syscall.txt
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (4.8s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (1.0s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (0.7s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (25.1s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (3.8s)
== Test time ==
time: OK
Score: 35/40
make: *** [Makefile:342: grade] Error 1
o dokhaihung@dokhaihung-virtual-machine:~/xv6-labs-2023$

```

### 3.2 Lưu ý khi tự chấm điểm

Đối với một số máy Linux có cấu hình không đủ mạnh thì ở trường hợp test `Test trace children` sẽ chạy quá thời gian `timeout` của trình chấm điểm được tích hợp trong xv6.

Giới hạn `timeout` của trình chấm `gradelib.py` có thể xem ở đây:



```
430     pass
431
432     class Runner():
433         def __init__(self, *default_monitors):
434             self.__default_monitors = default_monitors
435
436         def run_qemu(self, *monitors, **kw):
437             """Run a QEMU-based test.  monitors should functions that will
438             be called with this Runner instance once QEMU and GDB are
439             started.  Typically, they should register callbacks that throw
440             TerminateTest when stop events occur.  The target_base
441             argument gives the make target to run.  The make_args argument
442             should be a list of additional arguments to pass to make.  The
443             timeout argument bounds how long to run before returning."""
444
445             def run_qemu_kw(target_base="qemu", make_args=[], timeout=30):
446                 return target_base, make_args, timeout
447             target_base, make_args, timeout = run_qemu_kw(**kw)
448
449             # Start QEMU
450             pre_make()
451             self.qemu = QEMU(target_base + "-gdb", *make_args)
452             self.gdb = None
453
454     try:
```

Lý do cho việc chạy quá thời gian của test `Test trace children` không đến từ việc có sai sót trong lập trình mà đến từ yếu tố khách quan là cấu hình máy. Do đó, nếu muốn qua (pass) được testcase này, ta có thể tăng giới hạn thời gian `timeout` trong file `gradelib.py` lên, chẳng hạn như 50 (s) hoặc 100 (s).

Ở đây, bài làm của chúng em có thể pass được test này với giới hạn `timeout` mặc định là 30 (s).

### 3.3 Lời kết

- Trong lab này, chúng em đã thực hiện được các bài tập về cách thêm và sử dụng các hàm hệ thống mới trong hệ điều hành xv6. Chúng em đã hiểu được cơ chế hoạt động của các `system call` và cách tạo ra các `system call` mới cũng như các `struct` được khai báo.
- Lab rất bổ ích và chúng em rất mong được tiếp tục học tập và thực hành các lab tiếp theo của môn học này để có thể nắm vững kiến thức về hệ điều hành và phát triển kỹ năng lập trình của mình.