****

**BÁO CÁO ĐỒ ÁN 1:**

***Linux Kernel Module***

*Sinh viên thực hiện:*

*Lê Hồng Quang-18127190*

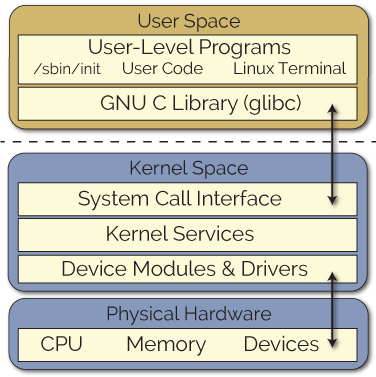
*Nguyễn Đình Thảo-18127219*

* Linux kernel Module

Linux được thiết kế để làm việc với hàng tỉ thiết bị. Nhưng ta không thể đưa tất cả các driver vào trong kernel được, vì sẽ làm cho kích thước kernel rất lớn. Giải pháp cho vấn đề này đó là: thiết kế các driver dưới dạng module tách rời với kernel. Trong quá trình hoạt động, driver nào cần thiết sẽ được lắp vào kernel, còn driver nào không cần thiết sẽ bị tháo ra khỏi kernel (dynamic loading).

Kernel module là một cơ chế giúp thêm hoặc loại bỏ code ra khỏi Linux kernel ở thời gian thực. Cơ chế này thích hợp cho những driver các thiết bị, kích hoạt kernel giao tiếp với phần cứng mà không cần phải biết rõ nguyên lý hoạt động của phần cứng.

Kernel module chạy ở Kernel Space và các ứng dụng chạy ở User Space (hình 1). Cả kernel space và user space đều có những vùng địa chỉ nhớ riêng biệt mà không bị trùng lắp nhau. Phương pháp này đám bảo các ứng dụng chạy ở user space có một cái nhìn nhất quán đối với phần cứng, bất kể nền tảng nào. Các dịch vụ ở kernel được cung cấp cho user space thông qua system calls. Kernel ngăn các ứng dụng ở user space có xung đột với nhau hoặc truy cập vào tài nguyên không cho phép thông qua các mức bảo vệ.



Hình 1 Giao tiếp giữa các thành phần máy tính

Linux kernel module là một file với tên mở rộng là (.ko). Nó sẽ được lắp vào hoặc tháo ra khỏi kernel khi cần thiết. Chính vì vậy, nó còn có một tên gọi khác là loadable kernel module. Một trong những kiểu loadable kernel module phổ biến đó là driver.

Việc thiết kế driver theo kiểu loadable module mang lại 3 lợi ích: Giúp giảm kích thước kernel. Do đó, giảm sự lãng phí bộ nhớ và giảm thời gian khởi động hệ thống. Không phải biên dịch lại kernel khi thêm mới driver hoặc khi thay đổi driver. Không cần phải khởi động lại hệ thống khi thêm mới driver. Trong khi đối với Windows, mỗi khi cài thêm driver, ta phải khởi động lại hệ thống, điều này không thích hợp với các máy server.

Phần lớn các driver đều là các loadable kernel module, nhưng không phải là tất cả. Vẫn có một số driver được tích hợp luôn vào trong kernel, đặc biệt là các bus driver. Chúng được gọi là built-in driver. Các device driver thường sẽ là các loadable kernel module. Ngược lại, không phải loadable kernel module nào cũng là driver, ví dụ kvm.ko là loadable kernel module nhưng không phải là driver. Trên thực tế, loadable kernel module được chia làm 3 loại chính: device driver, system call và file system.

Khi cần một module nhưng nó lại chưa có trong kernel space, kernel sẽ đưa module ấy vào. Quá trình này có thể diễn ra một cách tự động, với trình tự sau: Bước 1: Kernel kích hoạt tiến trìnhmodprobe cùng với tham số truyền vào là tên của module (ví dụ xxx.ko). Bước 2: Tiến trình modprobe kiểm tra file/lib/modules//modules.dep xem xxx.ko có phụ thuộc vào module nào khác không. Giả sử xxx.ko phụ thuộc vào module yyy.ko. Bước 3: Tiến trình modprobe sẽ kích hoạt tiến trìnhinsmod để đưa các module phụ thuộc vào trước (yyy.ko), rồi mới tới module cần thiết (xxx.ko). Như vậy, các module được đưa vào kernel space dưới sự giúp đỡ của tiến trình modprobe.

kernel kích hoạt tiến trình modprobe bằng 2 cách:

Cách 1 là sử dụng kmod. Đây là một thành phần của Linux kernel, hoạt động trong kernel space. Khi một thành phần nào đó của kernel cần đưa một module vào trong kernel space, nó sẽ truyền tên module cho hàm request\_module của kmod. Hàm request\_module sẽ gọi hàmcall\_usermodehelper\_setup để sinh ra tiến trình modprobe. Các bạn có thể tham khảo mã nguồn của kmod tại /kernel/kmod.c. Cách 2 là sử dụngudevd (hình 1). Đây là một tiến trình hoạt động trong user space. Nếu một thiết bị cắm vào hệ thống máy tính, thì điện trở trên bus ngoại vi (ví dụ PCI bus hoặc USB bus) sẽ thay đổi và bộ điều khiển (controller) sẽ biết điều này. Khi đó, bus driver sẽ gửi một bản tin lên cho tiến trình udevd. Bản tin này chứa thông tin về thiết bị. Tiến trình udevd sẽ tra cứu file /lib/modules//modules.alias để tìm ra driver nào tương thích với thiết bị. Sau đó, udevd sinh ra tiến trình modprobe.

A screenshot of a video game

Description automatically generated

Hình minh họa quá trình kích hoạt modprobe bằng udevd

\*Cách viết một linux kernel module

#include <linux/module.h> /\* thu vien nay dinh nghia cac macro nhu module\_init va module\_exit \*/

#define DRIVER\_AUTHOR “…."

#define DRIVER\_DESC "……."

static int \_\_init init\_hello(void)

{ printk("Hello Vietnam\n");

return 0;}

static void \_\_exit exit\_hello(void)

{printk("Goodbye Vietnam\n");}

module\_init(init\_hello);

module\_exit(exit\_hello);

MODULE\_LICENSE("GPL"); /\* giay phep su dung cua module \*/

MODULE\_AUTHOR(DRIVER\_AUTHOR); /\* tac gia cua module \*/

MODULE\_DESCRIPTION(DRIVER\_DESC); /\* mo ta chuc nang cua module \*/

MODULE\_SUPPORTED\_DEVICE("testdevice"); /\* kieu device ma module ho tro \*/

-Tham chiếu tới file của Linux kernel là <linux/module.h>. File này chứa 2 macro quan trọng, là: **module\_init()** và **module\_exit()**. Do đó, dù viết bất cứ kernel module nào, cũng cần tham chiếu tới <linux/module.h>.

* module\_init giúp xác định hàm nào sẽ được thực thi ngay sau khi lắp module vào kernel.
* module\_exit giúp xác định hàm nào được thực thi ngay trước khi tháo module ra khỏi kernel.

-init\_hello() là hàm được gọi ngay sau khi module hello được lắp vào, và exit\_hello() là hàm được gọi ngay trước khi module hello bị tháo ra khỏi kernel.

Macro **\_\_init** thường đi kèm với hàm khởi tạo. macro \_\_init xuất hiện trước tên hàm init\_hello. Macro này giúp kernel biết rằng, hàm init\_hello() chỉ phải thực thi lúc khởi tạo, nên vùng nhớ chứa hàm này có thể được giải phóng sau khi nó thực thi xong mà không ảnh hưởng gì.

Tương tự, macro **\_\_exit** thường đi kèm với hàm kết thúc. \_\_exit xuất hiện trước tên hàm exit\_hello. Macro này cho kernel biết, khi lắp module vào kernel thì chưa cần đưa hàm exit\_hello vào trong bộ nhớ RAM. Chỉ khi chuẩn bị tháo module ra khỏi kernel, hàm exit\_hello này mới cần được đưa vào RAM và thực thi.

Trong quá trình viết kernel module, các lập trình viên thường sử dụng hàm **printk** để ghi lai quá trình hoạt động của module. Việc này được gọi là logging. Mục đích của việc logging là để phục vụ quá trình gỡ lỗi sau này (debug). Ta có thể sử dụng lệnh **dmesg** để xem quá trình hoạt động của kernel kể từ lúc nó khởi động.

Các macro nằm ở cuối ví dụ trên cung cấp các thông tin về module. Ta có thể sử dụng lệnh **modinfo**để xem các thông tin của một module:

* Macro MODULE\_AUTHOR cho biết ai là người tạo ra module.
* Macro MODULE\_DESCRIPTION() cho biết module làm được những gì.
* Macro MODULE\_SUPPORTED\_DEVICE() cho biết module này hỗ trợ làm việc với những thiết bị nào.
* MODULE\_LICENSE cho biết người dùng có cần phải trả phí nếu sử dụng module hay không. Trong ví dụ trên, giấy phép sử dụng module thuộc loại GPL. Với giấy phép sử dụng GPL, người dùng có thể sử dụng module miễn phí. Ngoài GPL, còn có các loại license như GPL v2, BSD/GPL, MIT/GPL, MPL/GPL.

\*Biên dịch Linux Kernel Module

Để biên dịch kernel module, ta sử dụng phương pháp Kbuild. Theo phương pháp này, chúng ta cần tạo ra 2 file: một file có tên là **Makefile**, file còn lại có tên là **Kbuild**. Đầu tiên, ta sẽ tạo ra Makefile.

KDIR = /lib/modules/`uname -r`/build

all:

make -C $(KDIR) M=`pwd`

clean:

make -C $(KDIR) M=`pwd` clean

Trong Makefile trên:

* Thẻ **all** chứa câu lệnh để biên dịch các module trong thư mục hiện tại.
* Thẻ **clean** chứa lệnh xóa tất cả các object file có trong thư mục hiện tại.

Tiếp theo, ta tạo ra file Kbuild nằm trong cùng thư mục với Makefile:

EXTRA\_CFLAGS = -Wall

obj-m = hello.o// ví dụ tên 1 file

Trong file Kbuild trên:

* Biến **obj-m** chỉ ra rằng: object file sẽ được biên dịch theo kiểu kernel module.
* Cờ **-Wall** cho phép trình biên dịch hiển thị tất cả các bản tin cảnh báo trong quá trình biên dịch.

Để tạo ra kernel module, ta gõ lệnh **make** hoặc **make all**. Khi ta gõ lệnh "make", tiến trình **make** sẽ dựa vào Makefile và Kbuild để biên dịch mã nguồn, tạo ra kernel module.

Sau khi biên dịch xong, ta sẽ thấy xuất hiện môt file có tên mở rộng là **.ko** (ko là viết tắt của kernel object). Đây chính là kernel module. Để biết được các thông tin về module, ta sử dụng lệnh **modinfo.**

\*Lắp tháo Linux Kernel Module

Để lắp module vào trong kernel, ta có thể thực hiện thủ công bằng cách gõ lệnh **insmod**. Sau khi lắp xong, ta sẽ dùng lệnh **lsmod** để kiểm tra xem module đã được load thành công chưa. Tiếp theo, ta sẽ dùng lệnh **dmesg** để theo dõi quá trình hoạt động của module . Cuối cùng, chúng ta sẽ dùng lệnh **rmmod** để tháo module ra khỏi kernel.

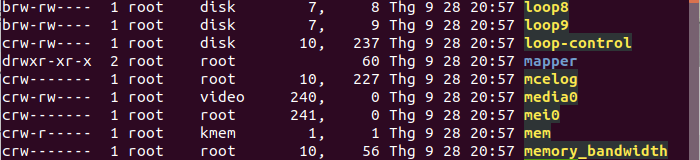
* Hệ thống quản lý file và device trong linux

## Character Device

Một character device sẽ giúp chuyển dữ liệu qua lại giữa kernel và ứng dụng ở user space. Chúng giống như một cổng để kết nối để đọc hoặc ghi các byte dữ liệu theo một luồng từng kí tự. Chúng cung cấp một nền tảng cho nhiều driver, ví dụ như video hay âm thanh. Một sự thay thế cho character device là block device. Block device cho phép một dãy buffer của cache điều khiển những thao tác như đọc, viết và tìm kiếm. Tất cả các thiết bị đều có thể được truy cập thông qua hệ thống file được đính kèm vào cây file hệ thống. Ví dụ, với đồ án này, toàn bộ code của chương trình khi chạy sẽ trở thành một device /dev/generDev

## Major and minor number

Major number được dùng trong kernel để xác định đúng driver của thiết bị khi thiết bị được truy cập. Mặt khác, minor number phụ thuộc vào thiết bị, và chỉ sử dụng bên trong driver. Ta có thể thấy được các cặp major/minor của mỗi device trong thư mục /dev.

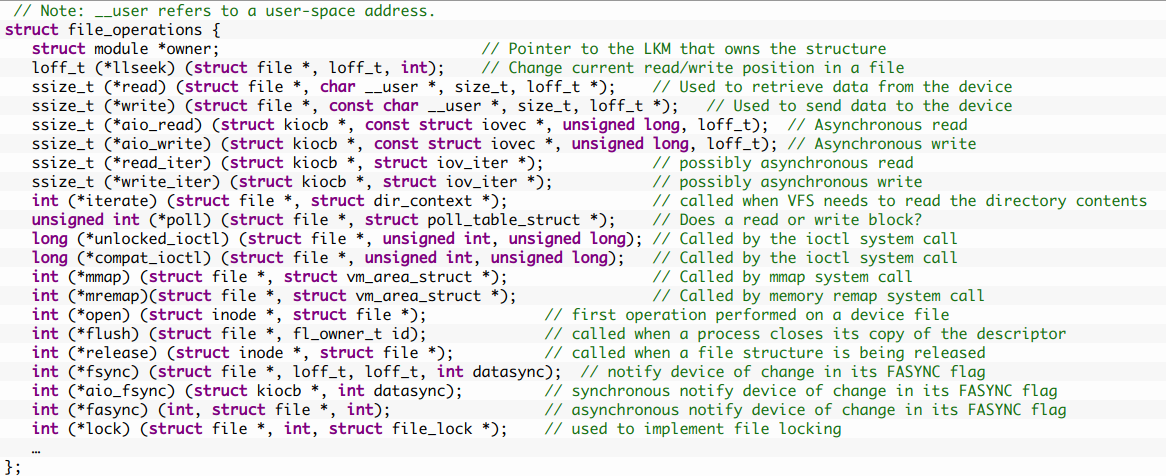


Hình 2 Major và minor number của các thiết bị trong Linux

Character device sẽ được xác định bởi chữ ‘c’ ở cột đầu tiên trong danh sách, và block device là chữ ‘b’. Quyền truy cập, chủ sở hữu và nhóm các thiết bị đều được cung cấp với mỗi thiết bị.

## Cấu trúc quản lý dữ liệu file

Cấu trúc quản lý dữ liệu file được khai bảo ở /linux/fs.h. Cấu trúc này chứa những con trỏ hàm trong một driver, cho phép định nghĩa các thao tác xử lý. Hình 4 à một số ví dụ thao tác trên cấu trúc này.

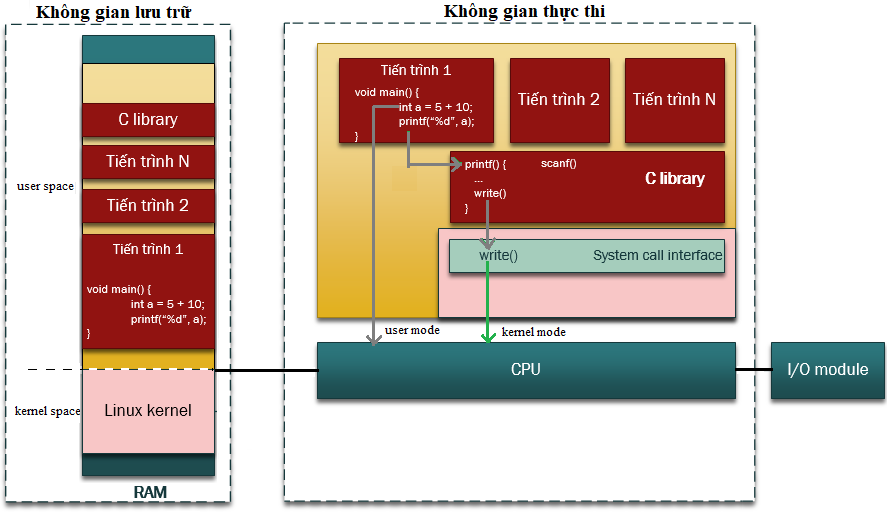


Hình 4 Cấu trúc quản lý dữ liệu file

* Giao tiếp giữa tiến trình ở user space và code kernel space:

Bộ nhớ RAM chứa các lệnh/dữ liệu dạng nhị phân của Linux kernel và các tiến trình. RAM được chia làm 2 miền (hình 4):

* **Kernel space** là vùng không gian chứa các lệnh và dữ liệu của kernel.
* **User space** là vùng không gian chứa các lệnh và dữ liệu của các tiến trình.



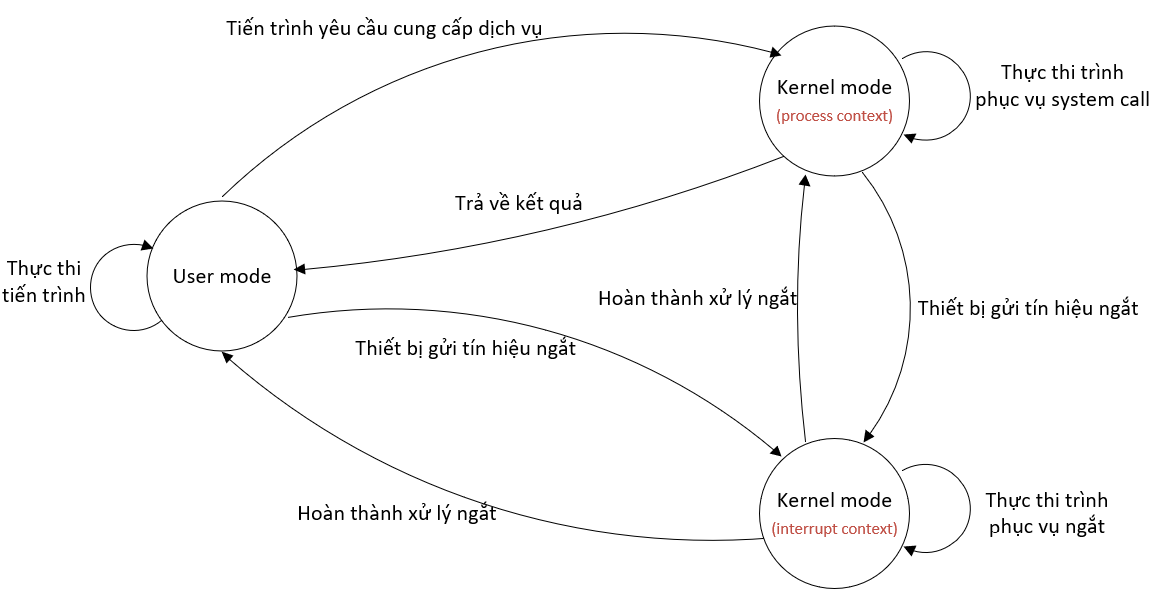
*Hình 4 Kiến trúc Linux kernel đứng ở góc độ thực thi của CPU (execution point of view)*

CPU có 2 chế độ thực thi (hình 4):

* Khi CPU thực thi các lệnh của kernel, thì nó hoạt động ở chế độ **kernel mode**. Khi ở chế độ này, CPU sẽ thực hiện bất cứ lệnh nào trong tập lệnh của nó, và CPU có thể truy cập bất cứ địa chỉ nào trong không gian địa chỉ.
* Khi CPU thực thi các lệnh của tiến trình, thì nó hoạt động ở chế độ **user mode**. Khi ở chế độ này, CPU chỉ thực hiện một phần tập lệnh của nó, và CPU cũng chỉ được phép truy cập một phần không gian địa chỉ.

Để cho dễ hình dung, ta hãy liên tưởng thế này: CPU giống như bạn, tiến trình giống như công việc của sếp giao cho bạn, còn kernel giống như công việc của vợ giao cho bạn. Khi bạn thực thi các việc của sếp, bạn chỉ thực hiện cho xong, không nỗ lực hết mình, những việc nào mà ảnh hưởng đến quyền lợi của bạn thì bạn không làm. Thái độ làm việc này của bạn giống như chế độ user mode của CPU. Khi bạn thực thi các việc của vợ, bạn thực hiện các việc đó toàn tâm toàn ý, nỗ lực hết mình, dù cho đó là việc gì. Thái độ làm việc này của bạn giống như chế độ kernel mode của CPU.

Để hiểu rõ hơn, ta xét tiến trình 1 trong hình 4. Tiến trình này gồm nhiều lệnh nhị phân, tương ứng với 2 lệnh C. CPU sẽ lần lượt lấy các lệnh này ra và thực thi. Lệnh thứ nhất, "a = 5 + 10", là một lệnh tính toán, sẽ được CPU thực thi ở chế độ user mode. Lệnh thứ hai, "printf("%d", a)", là một lệnh vào/ra. Hàm "printf" sẽ gọi system call "write" để yêu cầu Linux kernel in thông tin ra màn hình. Khi đó, CPU sẽ chuyển sang chế độ kernel mode để thực thi các lệnh của Linux kernel.



*Hình 5 Các chế độ hoạt động của CPU*

Khi một tiến trình cần sử dụng một dịch vụ nào đó của kernel, tiến trình sẽ gọi một **system call**. System call cũng tương tự như các hàm bình thường khác (library call). Chỉ có điều, các library call được cung cấp bởi các thư viện trong user space, còn các system call được cung cấp bởi kernel. Do đó, khi tiến trình gọi các library call, CPU vẫn giữ nguyên chế độ thực thi user mode. Còn khi tiến trình gọi các system call, CPU phải chuyển sang chế độ kernel mode để thực thi các lệnh của kernel (hình 5). Lúc này, ta nói rằng, CPU đang thực thi ở chế độ kernel mode, trong ngữ cảnh **process context**. Sau khi kernel thực hiện xong yêu cầu, kernel gửi trả kết quả cho tiến trình. Lúc này, CPU lại chuyển sang chế độ user mode để thực thi tiếp các lệnh của tiến trình.

Ngoài system call, ngắt cũng là một nguyên nhân khiến CPU chuyển chế độ thực thi sang kernel mode (hình 5). Khi có một thiết bị muốn trao đổi dữ liệu với CPU, nó sẽ gửi một tín hiệu ngắt tới CPU bằng cách nâng điện áp trên chân INT của CPU. Khi đó, CPU sẽ ngừng thực thi các lệnh của tiến trình lại, chuyển sang chế độ kernel mode rồi thực thi một chương trình đặc biệt của kernel để xử lý tín hiệu ngắt đó. Lúc này, ta nói CPU đang thực thi ở chế độ kernel mode, trong ngữ cảnh **interrupt context**. Sau khi xử lý xong, CPU trở lại chế độ user mode và tiếp tục thực hiện các lệnh tiếp theo của tiến trình.

Systemcall:

Để đáp ứng yêu cầu: để user gọi xuống kernel hay thao tác điều khiển các device driver, kernel cung cấp cho user space các API (còn gọi là các dịch vụ) là system call.

System call là một cửa ngõ vào kernel, cho phép tiến trình trên tầng user yêu cầu kernel thực thi một vài tác vụ cho mình. Những dịch vụ này có thể là tạo một tiến trình mới (fork), thực thi I/O (read, write), hoặc tạo ra một pipe cho giao tiếp liên tiến trình (IPC).

Có một số điều cần chú ý về system call như sau:

* Khi một tiến trình gọi một system call, CPU sẽ chuyển từ chế độ user mode sang kernel mode, điều này cho phép CPU truy cập các vùng nhớ và thực hiện các chỉ lệnh của kernel.
* Mỗi system call được kernel định danh bằng một số duy nhất. Tiến trình trên tầng user không biết đến các số này, thay vào đó, nó gọi một system call bằng tên hàm (ví dụ như open(), read()...).
* Mỗi system call có thể có một số tham số truyền để cung cấp thông tin từ user truyền xuống kernel và ngược lại.

# **Quá trình thực thi system call**

Đứng trên góc nhìn của lập trình viên, việc gọi một system call trông có vẻ như là chỉ gọi một hàm C bình thường. Tuy nhiên, đằng sau việc đó là rất nhiều bước được thực hiện từ user space xuống kernel space.

Cụ thể, chúng ta thử xét việc gọi một hàm thư viện được dùng rất thường xuyên sau đây:

Hàm gọi:

#include<stdio.h>

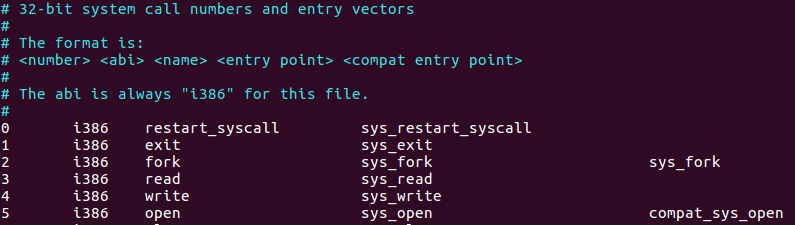
FILE \*fopen(const char \*filename, const char \*mode)

Hàm fopen() là một hàm thư viện (wrapper function) được dùng để thực thi việc chuyển xuống kernel mode và yêu cầu kernel mở một file dưới kernel có đường dẫn là “filename” với chế độ "mode", chi tiết về hàm này các bạn xem tại [đây](http://man7.org/linux/man-pages/man3/fopen.3.html). Hàm fopen() được triển khai bằng cách gọi system call open(), cụ thể các bước như sau:

1. Hàm wrapper copy các đối số ( trong trường hợp này là "filename" và "mode") vào các thanh ghi, nơi mà các lệnh của luồng thực thi system call sẽ đọc và sử dụng được.
2. Hàm wrapper sao chép số system call vào một thanh ghi của CPU (%eax). Ví dụ system call number của open() là 5, hàm này sẽ sao chép giá trị 5 vào thanh ghi %eax.
3. Hàm wrapper thực hiện một chỉ lệnh máy gọi là trap machine instruction để chuyển chế độ CPU từ user mode sang kernel mode. Chỉ lệnh này có thể là một ngắt mềm (software interrupt) với số ngắt (interrupt number) là 0x80 (int 0x80) hoặc chỉ lệnh SYSENTER (trong các kiến trúc Intel gần đây) hoặc chỉ lệnh SYSCALL (trong AMD)
4. Kernel gọi đến luồng system\_call (nằm trong file arch/x86/entry\_32.S), tại đây nó sẽ làm các công việc: copy giá trị các đối số trong các thanh ghi mà đã copy vào trong bước 1 vào kernel stack; kiểm tra tính hợp lệ của các đối số; gọi đến system call service routine thích hợp bằng cách tra cứu số system call được sao chép ở bước 2 trong bảng system call routine (sys\_call\_table); gửi kết quả trả về lên cho hàm wrapper và cuối cùng là chuyển chế độ của CPU từ kernel mode sang user mode.
5. Hàm wrapper trả về giá trị là một số nguyên cho hàm gọi nó để thông báo lời gọi system call có thành công không. Nếu system call trả về giá trị lỗi, hàm wrapper sẽ set giá trị cho một biến toàn cục “errno” từ giá trị lỗi này.

Lưu ý rằng các bước trên chỉ giới thiệu tinh thần chung của việc tiếp nhận xử lý một system call, các kiến trúc khác nhau sẽ có các cách triển khai tinh thần chung đó khác nhau ít nhiều.

Để hiểu rõ hơn về bức tranh tổng thể của system call, chúng ta sẽ đi vào chi tiết cách system call open(). Trong Linux x86\_32, open() có số system call là 5, vì vậy trong system call vector (arch/x86/entry/syscall/syscall\_32.tbl), open system call sẽ là entry thứ 5, tương ứng với system call routine là sys\_open.



Từ đây, kernel sẽ gọi đến hàm sys\_open() trong /fs/open.c để thực thi việc mở một file trong hệ thông file system và trả về một mô tả file fd cho user. Cụ thể, chúng ta xem hình vẽ chi tiết như hình dưới đây:

