**TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI**

**-----🙞🙜🕮🙞🙜-----**

****

**BÁO CÁO ĐỒ ÁN MÔN HỌC**

**Đồ án môn học (Thiết kế hệ thống ATTT)**

***Giảng viên hướng dẫn: ThS. Phạm Ngọc Hưng***

Nhóm sinh viên:

|  |  |
| --- | --- |
| Hoàng Viết Huy | 20141932 |

Hà Nội, 20/12/2018

MỤC LỤC

[DANH MỤC CÁC THUẬT NGỮ, TỪ VIẾT TẮT 3](#_Toc534927752)

[CHƯƠNG 1: TÌM HIỂU VỀ LINUX, LẬP TRÌNH KERNEL MODULE TRÊN LINUX 4](#_Toc534927753)

[1.1. Tìm hiểu về Linux. 4](#_Toc534927754)

[1.1.1. Giới thiệu về Linux 4](#_Toc534927755)

[1.1.2. Kiến trúc hệ điều hành Linux 4](#_Toc534927756)

# DANH MỤC CÁC THUẬT NGỮ, TỪ VIẾT TẮT

|  |  |  |
| --- | --- | --- |
| Thuật ngữ / Từ viết tắt | Từ nguyên gốc | Dịch nghĩa |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# CHƯƠNG 1: TÌM HIỂU VỀ LINUX

## Giới thiệu về Linux

Linux là tên gọi của một hệ điều hành máy tính và cũng là tên hạt nhân của hệ điều hành. Nó có lẽ là một ví dụ nổi tiếng nhất của phần mềm tự do và của việc phát triển mã nguồn mở.

Phiên bản Linux đầu tiên do Linus Torvalds viết vào năm 1991, lúc ông còn là một sinh viên của Đại học Helsinki tại Phần Lan. Ông làm việc một cách hăng say trong vòng 3 năm liên tục và cho ra đời phiên bản Linux 1.0 vào năm 1994. Bộ phận chủ yếu này được phát triển và tung ra trên thị trường dưới bản quyền GNU General Public License. Do đó mà bất cứ ai cũng có thể tải và xem mã nguồn của Linux.

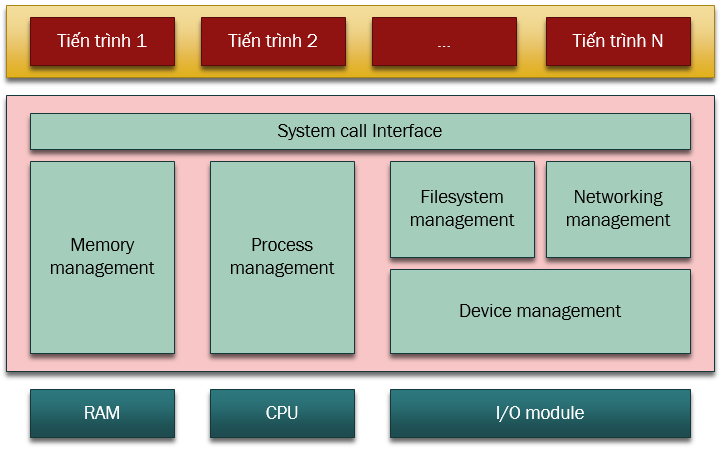
Một cách chính xác, thuật ngữ "Linux" được sử dụng để chỉ Nhân Linux, nhưng tên này được sử dụng một cách rộng rãi để miêu tả tổng thể một hệ điều hành tương tự Unix (còn được biết đến dưới tên GNU/Linux) được tạo ra bởi việc đóng gói nhân Linux cùng với các thư viện và công cụ GNU, cũng như là các bản phân phối Linux. Thực tế thì đó là tập hợp một số lượng lớn các phần mềm như máy chủ web, các ngôn ngữ lập trình, các hệ quản trị cơ sở dữ liệu, các môi trường desktop như GNOME và KDE, và các ứng dụng thích hợp cho công việc văn phòng như OpenOffice, LibreOffice.

## Kiến trúc hệ điều hành Linux

* + 1. **Linux kernel**

Vào năm 1991, dựa trên UNIX kernel, Linus Torvalds đã tạo ra Linux kernel chạy trên máy tính của ông ấy. Dựa vào chức năng của hệ điều hành, Linux kernel được chia làm 6 thành phần:

* **Process management**: có nhiệm vụ quản lý các tiến trình, bao gồm các công việc:
* Tạo/hủy các tiến trình.
* Lập lịch cho các tiến trình. Đây thực chất là lên kế hoạch: CPU sẽ thực thi chương trình khi nào, thực thi trong bao lâu, tiếp theo là chương trình nào.
* Hỗ trợ các tiến trình giao tiếp với nhau.
* Đồng bộ hoạt động của các tiến trình để tránh xảy ra tranh chấp tài nguyên.
* **Memory management:** có nhiệm vụ quản lý bộ nhớ, bao gồm các công việc:
* Cấp phát bộ nhớ trước khi đưa chương trình vào, thu hồi bộ nhớ khi tiến trình kết thúc.
* Đảm bảo chương trình nào cũng có cơ hội được đưa vào bộ nhớ.
* Bảo vệ vùng nhớ của mỗi tiến trình.
* **Device management:** có nhiệm vụ quản lý thiết bị, bao gồm các công việc:
* Điều khiển hoạt động của các thiết bị.
* Giám sát trạng thái của các thiết bị.
* Trao đổi dữ liệu với các thiết bị.
* Lập lịch sử dụng các thiết bị, đặc biệt là thiết bị lưu trữ (ví dụ ổ cứng).
* **File system management:** có nhiệm vụ quản lý dữ liệu trên thiết bị lưu trữ (như ổ cứng, thẻ nhớ). Quản lý dữ liệu gồm các công việc: thêm, tìm kiếm, sửa, xóa dữ liệu.
* **Networking management:** có nhiệm vụ quản lý các gói tin (packet) theo mô hình TCP/IP.
* **System call Interface**: có nhiệm vụ cung cấp các dịch vụ sử dụng phần cứng cho các tiến trình. Mỗi dịch vụ được gọi là một system call.



Hình 1: Kiến trúc của Linux kernel đứng ở góc độ quản lý (management point of view)

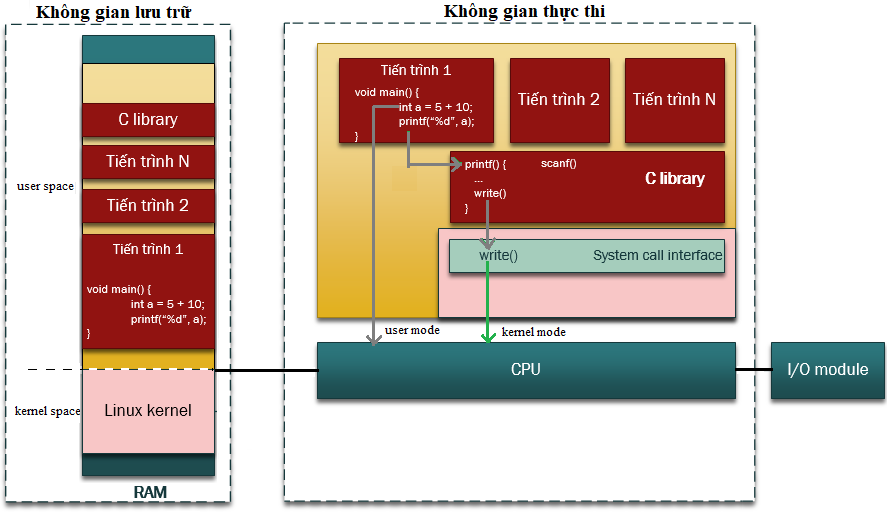
|  |  |
| --- | --- |
| **Thư mục** | **Vai trò** |
| /arch | Chứa mã nguồn giúp Linux kernel có thể thực thi được trên nhiều kiến trúc CPU khác nhau như x86, alpha, arm, mips, mk68, powerpc, sparc,… |
| /block | Chứa mã nguồn triển khai nhiệm vụ lập lịch cho các thiết bị lưu trữ. |
| /drivers | Chứa mã nguồn để triển khai nhiệm vụ điều khiển, giám sát, trao đổi dữ liệu với các thiết bị. |
| /fs | Chứa mã nguồn triển khai nhiệm vụ quản lý dữ liệu trên các thiết bị lưu trữ. |
| /ipc | Chứa mã nguồn triển khai nhiệm vụ giao tiếp giữa các tiến trình |
| /kernel | Chứa mã nguồn triển khai nhiệm vụ lập lịch và đồng bộ hoạt động của các tiến trình. |
| /mm | Chứa mã nguồn triển khai nhiệm vụ quản lý bộ nhớ |
| /net | Chứa mã nguồn triển khai nhiệm vụ xử lý các gói tin theo mô hình TCP/IP. |

Khi triển khai thực tế, mã nguồn của Linux kernel gồm các thư mục sau:

* + 1. **Các khái niệm thường dùng**

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 2):

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

CPU có 2 chế độ thực thi (hình 2):

* 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.

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. 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. 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.

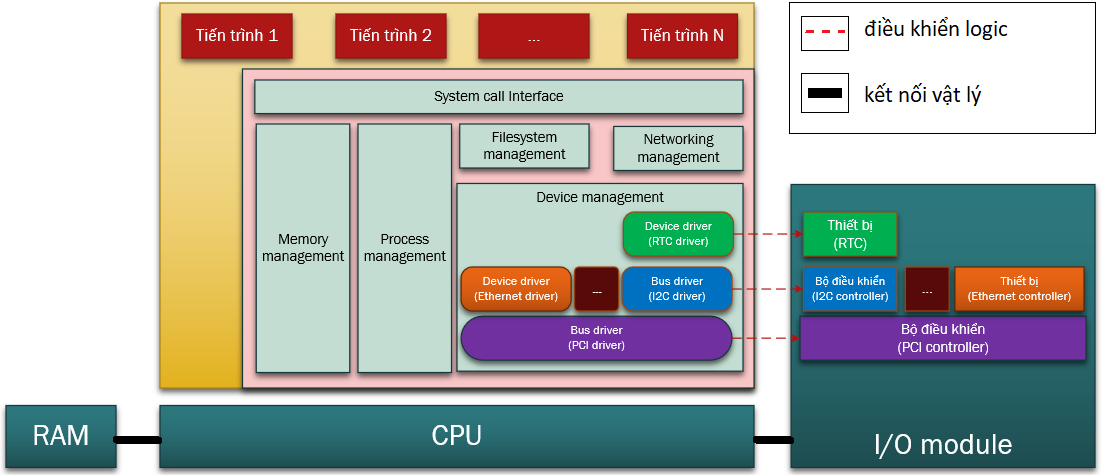
## Giới thiệu về Linux driver

Driver là một phần mềm, gồm các lệnh, hướng dẫn CPU cách tương tác với thiết bị. Các thiết bị có thể là chuột, bàn phím, ổ cứng, card mạng, loa, màn hình,… Tuy nhiên, các thiết bị này không được nối trực tiếp với CPU, bởi vì:

* Hệ thống có nhiều thiết bị, nhưng số lượng chân của CPU hữu hạn.
* Tốc độ làm việc của các thiết bị thấp hơn nhiều so với CPU.

Chính vì vậy, các thiết bị kết nối với CPU thông qua một thiết bị khác, gọi là bộ điều khiển (device controller).

Đứng từ góc độ của CPU, bộ điều khiển cũng chỉ là một thiết bị. Do đó, cần có driver hướng dẫn CPU làm việc với bộ điều khiển. Driver này được gọi là bus driver. Còn driver hướng dẫn CPU làm việc với thiết bị thì được gọi là device driver.



Trong khuôn khổ bản báo cáo này, chúng ta sẽ chỉ nghiên cứu về device driver.

## Giới thiệu về Linux device driver

Device driver gồm 2 phần:

* Thành phần OS specific. Thành phần này cung cấp cho hệ điều hành các dịch vụ đọc/ghi dữ liệu của thiết bị. Điều này cho phép chúng ta xây dựng hệ điều hành không bị phụ thuộc vào cấu trúc của thiết bị.
* Thành phần device specific. Thành phần này chứa các lệnh hướng dẫn CPU điều khiển thiết bị, giám sát thiết bị, trao đổi dữ liệu với thiết bị. Chúng ta sử dụng datasheet của thiết bị để xây dựng thành phần này. Datasheet là một tài liệu được cung cấp bởi nhà sản xuất thiết bị. Nó mô tả sơ đồ khối chức năng, nguyên lý hoạt động, hiệu suất hoạt động, đặc tính điện của thiết bị và đặc biệt là bản đồ thanh ghi (register map).

Trên Linux, các thiết bị được phân thành hai dạng chính:

* Thiết bị kiểu kí tự (Character Device): là các thiết bị phần cứng mà đọc và ghi dữ liệu theo một dòng các byte nối tiếp. Ví dụ: các thiết bị usb, cổng COM…
* Thiết bị kiểu khối (Block Device): là các thiết bị phần cứng mà đọc và ghi dữ liệu theo các khối có kích thước cố định. Khác với các thiết bị kiểu kí tự, thiết bị kiểu khối cung cấp khả năng truy nhập ngẫu nhiên tới dữ liệu được lưu trữ trên thiết bị. Ví dụ: ổ đĩa cứng…

# CHƯƠNG 2: LẬP TRÌNH KERNEL MODULE TRÊN LINUX

## Linux kernel module

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.

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.

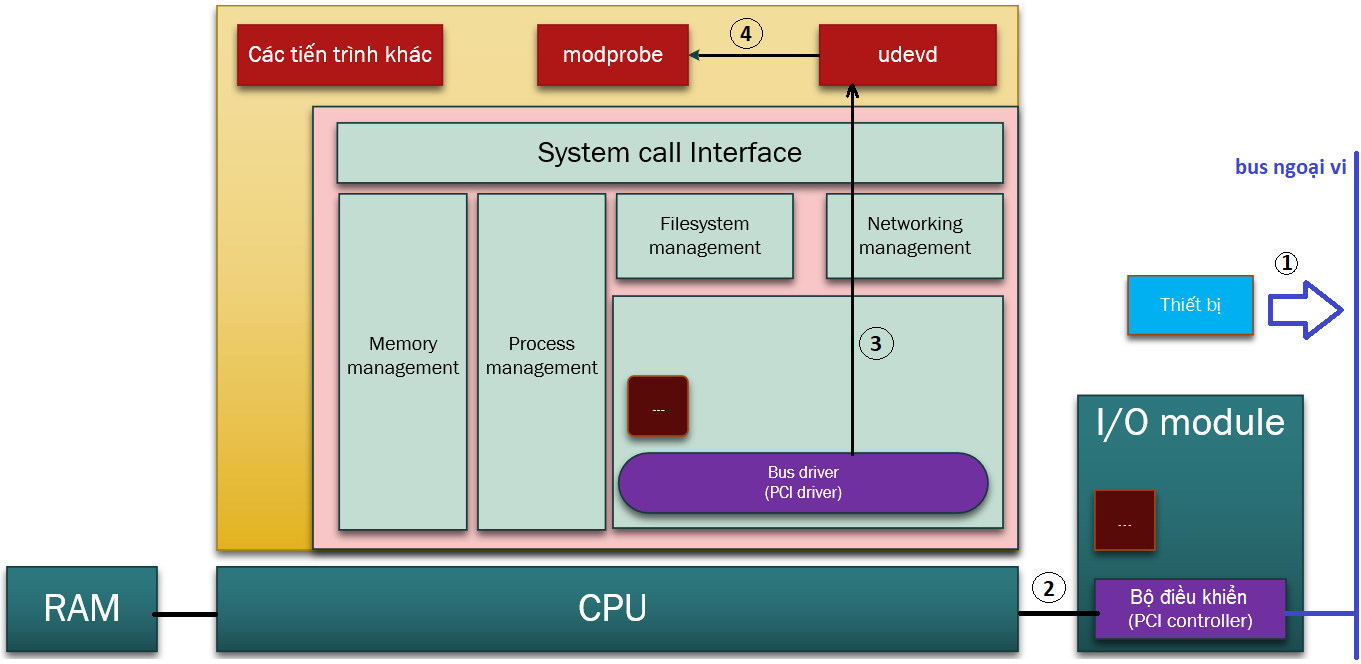
Ngược lại, không phải loadable kernel module nào cũng là driver, ví dụ kvm.ko là loadable kernel module nhưng không phải là driver. Trên thực tế, loadable kernel module được chia làm 3 loại chính: device driver, system call và file system.

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 trình modrobe cùng với tham số truyền vào là tên của module (ví dụ usb\_driver.ko).
* Bước 2: Tiến trình modprobe kiểm tra file /lib/modules/<kernel-version>/modules.dep xem usb\_driver.ko có phụ thuộc vào module nào khác không. Giả sử usb\_driver.ko phụ thuộc vào module management.ko.
* Bước 3: Tiến trình modprobe sẽ kích hoạt tiến trình insmod để đưa các module phụ thuộc vào trước (management.ko), rồi mới tới module cần thiết (usb\_driver.ko).

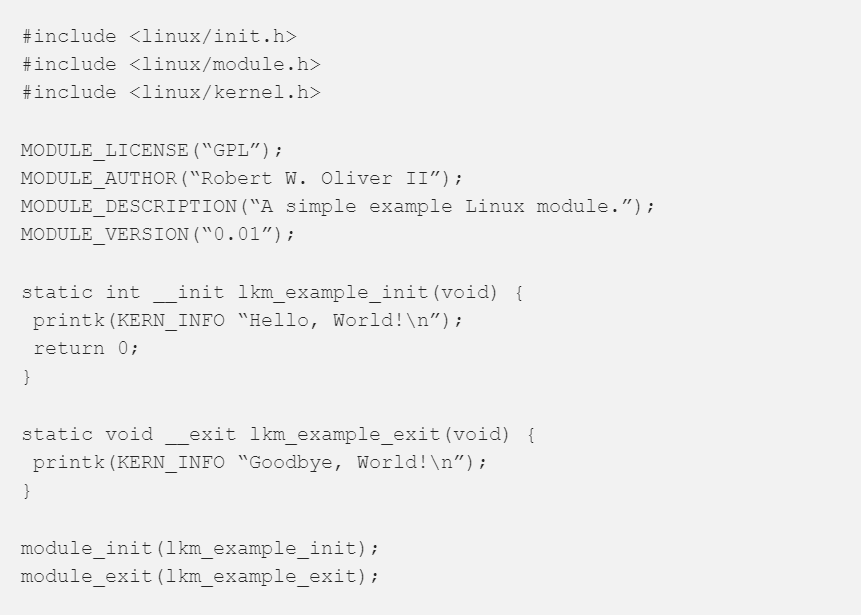
Như vậy, các module được đưa vào kernel space dưới sự giúp đỡ của tiến trình modprobe. Các cách kernel kích hoạt tiến trình modprobe:

* 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 hàm call\_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 dụng udevd (hình 1). Đây là một tiến trình hoạt động trong use 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/<kernel-version>/modules.alias để tìm ra driver nào tương thích với thiết bị. Sau đó, udevd sinh ra tiến trình modprobe.



## Lập Trình Kernel Module

* + 1. Mã nguồn chương trình hello.c



* + 1. Phân tích chương trình

Do module hello cần dùng một số hàm hoặc macro của Linux kernel, nên chúng ta sẽ sử dụng từ khóa #include để chỉ rõ các file cần dùng. Điều này cũng tương tự như khi viết các chương trình ứng dụng trên user space, chúng ta cũng dùng #include cùng với tên của các thư viện. Do đó, có thể nói rằng, Linux kernel chính là một thư viện, cung cấp các hàm, các macro để chúng ta phát triển kernel module.

Trong ví dụ trên, ta chỉ cần 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, ta 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.

Trong ví dụ trên, 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. Trong ví dụ trên, 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.

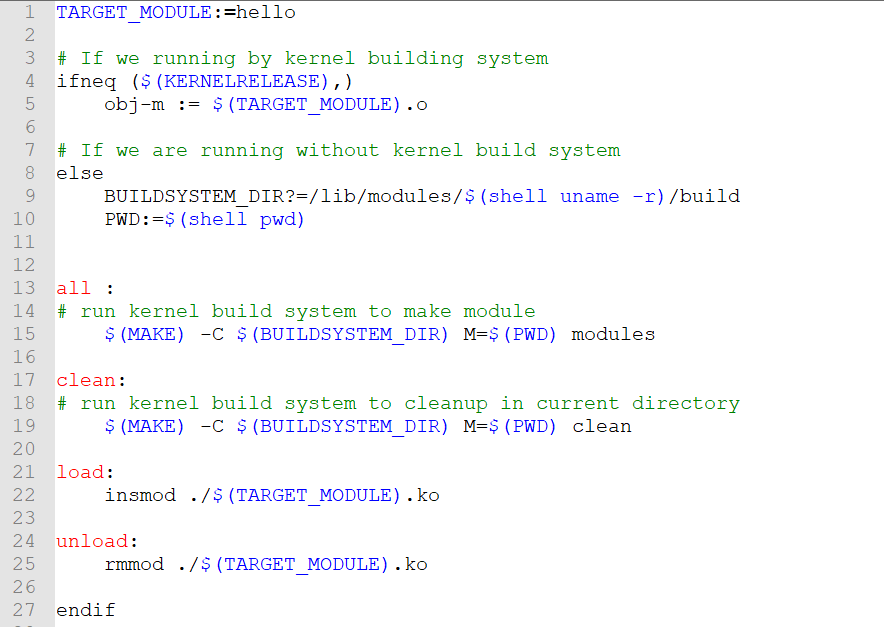
Tương tự, macro \_\_exit thường đi kèm với hàm kết thúc. Trong ví dụ trên, \_\_exit xuất hiện trước tên hàm exit\_hello. Macro này cho kernel biết, khi lắp module vào kernel thì chưa cần đưa hàm exit\_hello vào trong bộ nhớ RAM. Chỉ khi chuẩn bị tháo module ra khỏi kernel, hàm exit\_hello này mới cần 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.
  + 1. Biên dịch kernel module

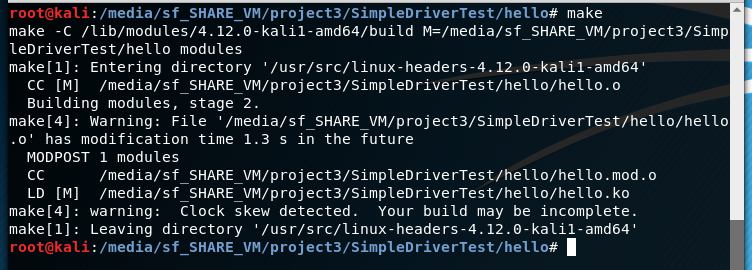
Để biên dịch cho các mô-đun nhân chúng ta sẽ tạo ra các Makefile. Trong tài liệu hướng dẫn của mã nguồn nhân Linux: linux/Documentation/kbuild/modules.txt có hướng dẫn chi tiết cách viết Makefile. Dưới đây là Makefile của chương trình hello.c



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.

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

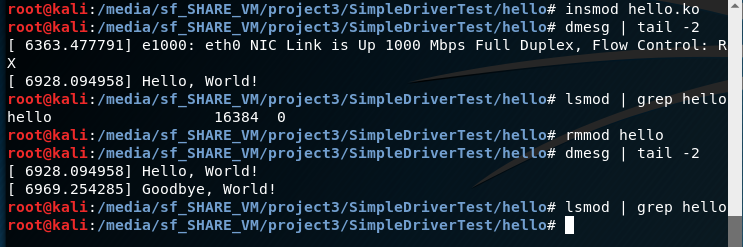


Sau khi biên dịch xong, ta sẽ thấy xuất hiện môt file có tên mở rộng là .ko (ko là viết tắt của kernel object). Đây chính là kernel module. Để biết được các thông tin về module, ta sử dụng lệnh modinfo



* + 1. Thêm/Xóa 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



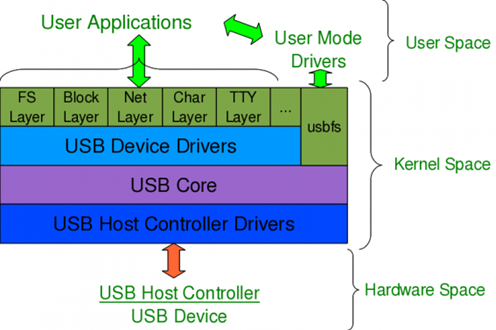
# CHƯƠNG 3: LẬP TRÌNH USB DEVICE DRIVER CHO THIẾT BỊ NANOPC-T3

## Lập trình USB Device Driver Cho Thiết Bị Nanopc-T3

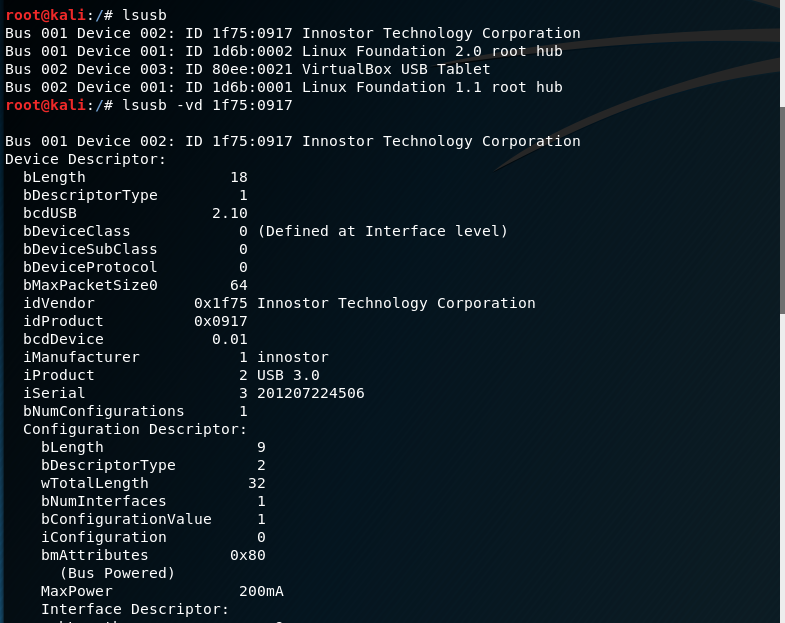
* + 1. Nhận dạng thiết bị USB trên Linux

Khi có một thiết bị usb hợp lệ được cắm vào hệ thống Linux, cho dù nó có driver hay không thì nó cũng vẫn được nhận diện (detect) bởi phần cứng ở tầng nhân (kernel space) của hệ thống Linux mà đã được hỗ trợ giao thức usb. Hệ thống có thể làm điều này là bởi vì khả năng của chính bản thân giao thức usb đã được thiết kế trong đặc tả của nó. Cụ thể, việc phát hiện ra thiết bị usb cắm vào được thực hiện bởi chip usb host controller (là thiết bị chủ động đường bus của giao thức usb). USB host controller này sẽ thu thập và diễn giải các thông tin ở tầng vật lý (low-level) đến các thông tin đặc tả giao thức USB ở tầng trên (high-level). Các thông tin về thiết bị theo khuôn dạng qui định của giao thức USB lại tiếp tục được đưa vào tầng usb core tổng quát (generic usb core) trong tầng nhân (được điều khiển bởi usbcore driver). Chính điều này giúp cho các thiết bị usb được hệ thống nhận diện ở tầng nhân, mặc dù nó có thể chưa có một driver cụ thể nào cho chức năng của nó.

Sau quá trình diễn ra ở tầng nhân này, sẽ đến nhiệm vụ của các drivers hoặc interfaces hoặc applications (cái mà phụ thuộc vào các bản Linux khác nhau) để tiếp tục nhận dạng ra thiết bị ở tầng người dùng (user space). Hình dưới minh họa cho kiến trúc phân tầng từ trên xuống của hệ thống USB trên Linux



Trên hệ điều hành Linux ta chỉ việc kết nối thiết bị tới máy tính, chạy lệnh lsusb trên terminal, tất cả các thiết bị USB đang kết nối với máy tính sẽ được liệt kê ra. Từ đó ta có thể biết được idVendor và idProduct của thiết bị. Tiếp tục, gõ lệnh lsusb –vd <idVendor>:<idProduct> để hiển thị các thông tin về cấu hình USB của thiết bị



Mô tả về phần cấu hình USB:

* USB có một Configuration
* Configuration này có một Interface
* Interface có 2 Endpoint là Bulk IN và Bulk OUT
  + 1. Khai báo danh sách các thiết bị có thể được điều khiển bởi Driver

Khi viết một USB Driver người lập trình viên cần chỉ định xem Driver này sẽ được sử dụng cho các thiết bị hoặc các lớp thiết bị nào. Cấu trúc usb\_device\_id cung cấp một kiểu thiết bị, nó có thể là một thiết bị cụ thể cũng có thể là một lớp thiết bị. Có các macro để khởi tạo cấu trúc này:

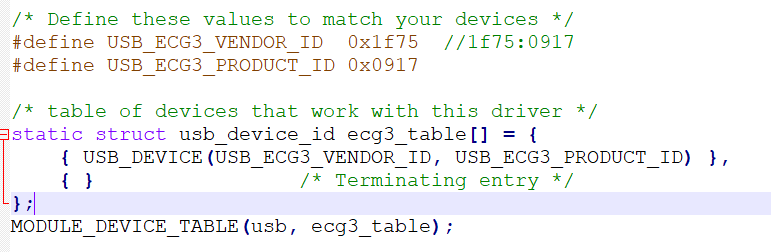
|  |  |
| --- | --- |
| Tạo ra một cấu trúc usb\_device\_id với IDvendor (số hiệu nhà sản xuất) và IDproduct (số hiệu sản phẩm) | USB\_DEVICE (vendor, product) |
| Tạo ra một cấu trúc usb\_device\_id với IDvendor, IDproduct và trong một dải phiên bản từ lo tới hi | USB\_DEVICE\_VER (vendor, product, lo, hi) |
| Tạo ra một cấu trúc usb\_device\_id phù hợp với lớp thiết bị USB được chỉ định bởi: lớp class, lớp con subclass, giao thức protocol | USB\_DEVICE\_INFO(class, subclass, protocol) |
| Tạo ra một cấu trúc usb\_device\_id phù hợp với một giao diện USB được chỉ định bởi: lớp class, lớp con subclass, giao thức protocol | USB\_INTERFACE\_INFO(class,  subclass, protocol) |

Sau khi đã tạo ra một cấu trúc usb\_device\_id ta cần phải khai báo cấu trúc này với USB Core, để làm việc này ta sử dụng macro:

MODULE\_DEVICE\_TABLE(usb, usb\_device\_id[]);

Trong đó usb\_device\_id[] là một mảng các cấu trúc usb\_device\_id đã khởi tạo trước đó.

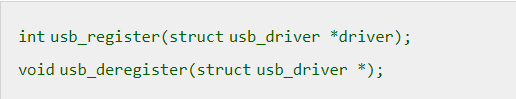
Trong ví dụ lần này, ta có đoạn code như sau:



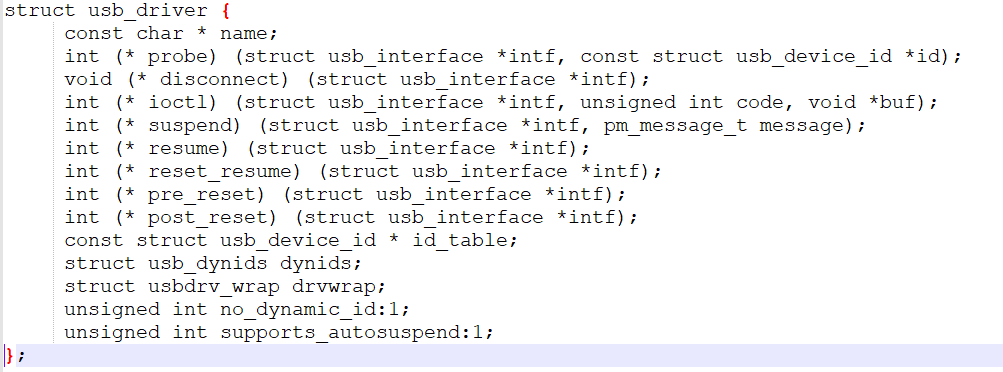
* + 1. Hàm đăng kí và hủy đăng kí USB Device Driver

Việc xây dựng driver này cũng giống như bất kỳ một driver thiết bị nào khác trên Linux, cần sử dụng một khuôn mẫu chung (có hàm tạo, hàm hủy). Tuy nhiên, nội dung chi tiết sẽ rất khác vì đây là một driver cho lớp giao thức phần cứng không giống như một character driver thông thường. Thay vì thực hiện đăng ký và hủy đăng ký với hệ thống file ảo (VFS – virtual file system) như của character driver thì ở đây, USB device driver cần thực hiện công việc tương ứng với lớp giao thức USB core; và thay vì cung cấp giao diện lên tầng người dùng (user space) giống như file thiết bị thì ở đây, usb driver cần thực hiện kết nối đến thiết bị thật trong tầng phần cứng (hardware-space).

Các hàm API của tầng USB core để thực hiện đăng ký và hủy đăng ký thiết bị như sau (nằm trong tệp tiêu đề <linux/usb.h>):



Hàm này thường được gọi trong hàm khởi tạo mô-đun Driver đang viết.Tham số cần truyền cho hàm này là một con trỏ tới cấu trúc usb\_driver. Cấu trúc này bao gồm các thông tin về Driver đang viết. Cấu trúc này được định nghĩa như sau:



Các thông tin quan trọng gồm:

* const char\* name : tên của Driver
* const struct usb\_device\_id\* id\_table : con trỏ tới bảng chứa các thiết bị sẽ được điều khiển bởi Driver này và đã được khai báo bằng macro MODULE\_DEVICE\_TABLE().
* int (\*probe) (struct usb\_interface \*intf, const struct usb\_device\_id\* id) : đây là một tham số rất quan trọng. Tham số này là một con trỏ tới một hàm (hàm thăm dò), hàm này sẽ được gọi khi thiết bị được kết nối tới hệ thống. Trong hàm này ta sẽ thực hiện các công việc quan trọng như xác định các Endpoint, cấp phát bộ nhớ…
* void (\*disconnect) (struct usb\_interface\* intf) : Một con trỏ tới một hàm (hàm ngắt kết nối), hàm này sẽ được gọi khi thiết bị được gỡ bỏ ra khỏi hệ thống. Trong hàm này lập trình viên cần phải thực hiện công tác dọn dẹp hệ thống như giải phóng bộ nhớ, hủy các công việc đang dở dang…
  + 1. Hàm thăm dò thiết bị (probe)

Khi thiết bị mới được kết nối tới hệ thống, nếu Driver được chỉ định cho điều khiển thiết bị đó thì hàm thăm dò của Driver sẽ được gọi. USB Core truyền tới hàm thăm dò một con trỏ tới cấu trúc usb\_interface mô tả Interface được chọn trên thiết bị.

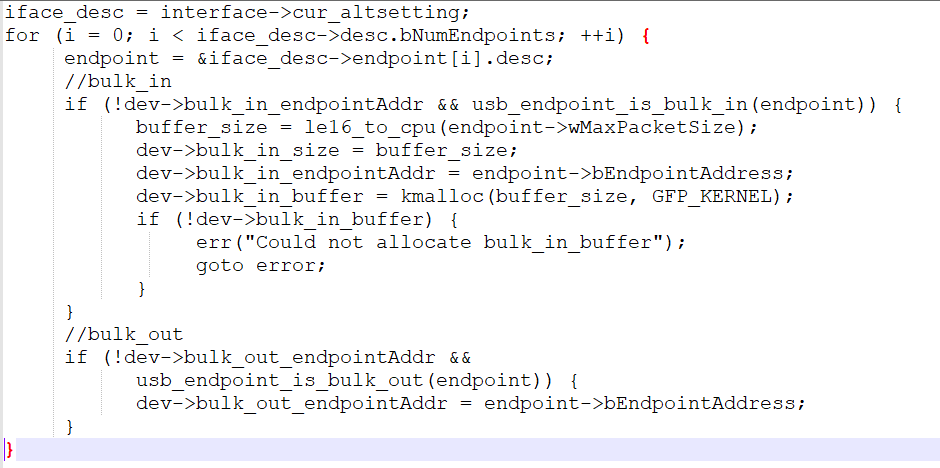
Nguyên mẫu hàm thăm dò như sau:



Trong hàm thăm dò, Driver cần thực hiện một số công việc sau:

* Lấy ra địa chỉ các Endpoint cần dùng, lấy ra kích thước các bộ đệm cho thiết bị
* Cấp phát bộ đệm
* Lưu lại các thông tin (địa chỉ Endpoint, kích thước bộ đệm, địa chỉ bộ đệm…)
* Đăng kí lớp thiết bị cho Driver
  + - 1. **Truy nhập các thông tin về Endpoint**

Đoạn mã sau thực hiện việc lấy các thông tin của một Endpoint Bulk IN và một Endpoint Bulk OUT:



* + - 1. **Lưu lại các thông tin đã truy nhập và cấp phát**

Để lưu trữ các dữ liệu liên quan tới thiết bị nhằm mục đích sử dụng sau, ta sử dụng hàm sau:

usb\_set\_intfdata(intf, dev);

Trong đó dev là con trỏ tới cấu trúc dữ liệu ta cần lưu trữ. (cấu trúc dữ liệu giao tiếp với thiết bị được người lập trình khai báo thích hợp với dữ liệu thiết bị cung cấp)

* + - 1. **Đăng kí lớp thiết bị**

Như ta đã thấy trong mô hình phân lớp của hệ thống USB trên Linux, phía trên lớp USB Device Driver là lớp Các hệ thống con nhân Linux (hay Các lớp thiết bị), bao gồm: Char, Block, TTY, Mem… Các thiết bị USB có thể được đối xử theo một trong các hệ thống con đó, chẳng hạn như chuột, bàn phím có thể coi như thuộc lớp Input. Tức là các ứng dụng người dùng sẽ sử dụng các API lớp Input để thực hiện giao tiếp với chuột, bàn phím. Do đó Driver cũng cần đăng kí thực hiện đăng kí lớp thiết bị mà nó quản lý. Tùy theo từng lớp  có các hàm đăng kí khác nhau:

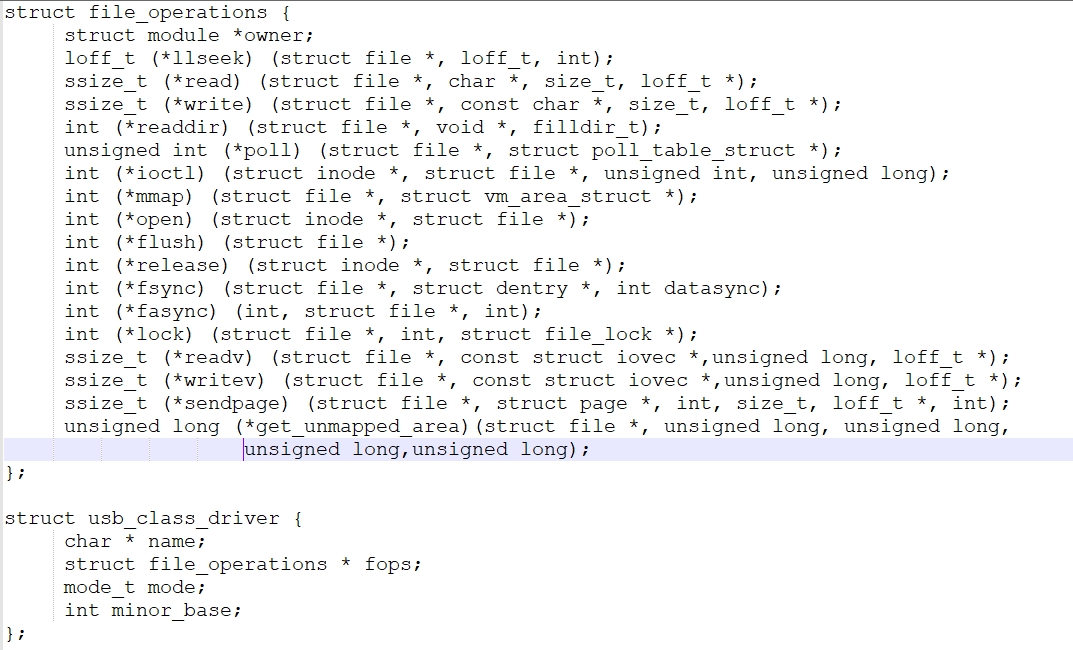
* Nếu Driver muốn đối xử với thiết bị như kiểu Input, ta sử dụng hàm:

input\_register\_device(struct input\_dev\* )

* Nếu Driver muốn đối xử với thiết bị như một thiết bị kiểu kí tự thông thường ta sử dụng hàm:

usb\_register\_dev(struct usb\_interface\*, struct usb\_class\_driver\* )

Tham số đầu tiên cần truyền cho hàm usb\_register\_dev là con trỏ tới cấu trúc chứa thông tin về Interface được chọn trên thiết bị, tham số thứ hai là con trỏ tới cấu trúc usb\_class\_driver. Cấu trúc này có một trường quan trọng là một con trỏ tới cấu trúc file\_operations định nghĩa các thao tác trên thiết bị được Driver hỗ trợ. Hai cấu trúc này được định nghĩa như dưới đây:

Trong cấu trúc file\_operations các trường thông tin chính là các con trỏ hàm. Các con trỏ này trỏ tới các hàm tương ứng với các thao tác trên tệp tin như: mở, đọc,ghi, ioctl, xóa… Chúng ta cần cài đặt các hàm này để khi có hành động tương ứng trên tệp tin, hàm của chúng ta sẽ thực thi các công việc cần thiết như: chuẩn bị dữ liệu, chuyển dữ liệu từ không gian nhân sang không gian người dùng và ngược lại, giải phóng dữ liệu…

* + 1. Hàm ngắt kết nối thiết bị

Khi thiết bị được gỡ bỏ ra khỏi hệ thống, hàm ngắt kết nối được gọi. Nguyên mẫu hàm như sau:

**void (\*disconnect) (struct usb\_interface\* intf);**

Trong hàm disconnect cần thực hiện hai công việc sau:

* Hủy các dữ liệu về thiết bị đã lưu trữ từ hàm thăm dò, để làm điều này ta sẽ thiết lập dữ liệu NULL cho interface intf:
* usb\_set\_intfdata(intf, NULL);
* Hủy đăng kí lớp thiết bị:
* usb\_deregister\_dev(struct usb\_interface\* , struct usb\_class\_driver\* );
  + 1. Các hàm mở/đọc/ghi thiết bị
       1. **Hành động mở tệp tin thiết bị**

**static int mydevice\_open(struct inode \*inode, struct file \*file);**

Hành động này có tác dụng chuẩn bị cho các hành động đọc, ghi sau đó. Trong hàm này ta sử dụng hàm usb\_get\_intfdata() để lấy ra các thông tin liên quan tới thiết bị đã lưu trữ từ hàm probe() (bằng hàm usb\_set\_intfdata), và thiết lập dữ liệu này cho cấu trúc file.

dev = usb\_get\_intfdata(interface);

file->private\_data = dev;

Cấu trúc file (được định nghĩa trong <linux/fs.h>) là một cấu trúc rất quan trọng trong Driver. Chúng ta cần chú ý rằng đây là một cấu trúc dữ liệu trong không gian nhân và cấu trúc này sẽ không liên quan gì tới con trỏ FILE\* trong thư viện của ngôn ngữ C trong không gian người dùng. Cấu trúc file thể hiện một tệp tin đang mở trong hệ thống Linux. Khi một tệp tin được mở, một thể hiện của cấu trúc này được tạo ra và liên kết với tệp tin đó. Khi bất kỳ hàm nào (đọc, ghi…) thao tác trên tệp tin, thể hiện của cấu trúc này sẽ được truyền cho hàm đó.

* + - 1. **Hành động đọc / ghi**

1. **Hành động đọc**

static ssize\_t mydevice\_read(struct file \*file, char \_\_user \*buffer, size\_t count, loff\_t \*ppos);

Hàm này có hai việc chính, đầu tiên là tạo ra các URB (USB request block) yêu cầu dữ liệu và xác nhận nó tới tầng USB Core, thứ hai là chuyển  dữ liệu nhận được từ các URB sang không gian người dùng bằng hàm:

unsigned long copy\_to\_user(void \_\_user \* to, const void \* from, unsigned long size);

1. **Hành động ghi**

static ssize\_t my\_device  write(struct file \*file, const char \_\_user \*buffer, size\_t count, loff\_t \*ppos);

Hàm này sẽ thực hiện hai công việc ngược lại với hàm đọc. Đầu tiên nó phải chuyển dữ liệu từ không gian người dùng sang không gian nhân hệ  điều hành, sau đó nó cần đóng gói dữ liệu này trong các URB và xác nhận các URB này tới tầng USB Core.

Để copy dữ liệu người dùng vào không gian nhân, ta sử dụng hàm:

unsigned long copy\_from\_user(void \* to, const void \_\_user \* from, unsigned long size);

## Biên dịch module trên Thiết Bị Nanopc-T3

# Refer: https://vimentor.com/vi/lesson/gioi-thieu-ve-linux-kernel-1

<https://sites.google.com/site/embedded247/ddcourse/usb-device-drivers-phan-1-usb-driver-tren-linux-1>

https://sites.google.com/site/embedded247/ddcourse/qui-trinh-viet-usb-device-driver-tren-linux