

25 YEARS ANNIVERSARY

SOKT

The graphic consists of the number "25" in large, bold, white letters. A curved line arches over the top of the "2" containing the text "YEARS ANNIVERSARY". Below the "25" is the acronym "SOKT" in a large, bold, white, sans-serif font.

**ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG**



ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

IT4425

Phát triển phần mềm nhúng thông minh

Giảng viên: TS. Phạm Ngọc Hưng
Viện Công nghệ Thông tin và Truyền thông
hungpn@soict.hust.edu.vn

Giới thiệu học phần

- IT4425 Phát triển phần mềm nhúng thông minh
(Smart Embedded System Software Development)
- Khối lượng (Credits): 2(2-0-1-4)

Mục tiêu

- Nắm được kiến thức tổng quan về phát triển phần mềm nhúng
- Nắm được kiến trúc hệ nhúng nền tảng ARM Linux
- Nắm được các kỹ thuật cơ bản phát triển ứng dụng cho hệ nhúng ARM Linux:
 - Giao tiếp GPIO cho phần mềm nhúng
 - Giao tiếp chuẩn RS232, USB cho phần mềm nhúng
- Hiểu và khai thác các kỹ thuật lập trình nâng cao cho phát triển phần mềm nhúng:
 - Tiến trình (process), luồng (Thread), phát triển ứng dụng đa luồng, đồng bộ luồng, giao tiếp dùng socket
- Nắm được cơ chế xây dựng trình điều khiển thiết bị (device driver)

Nội dung

- Chương 1. Tổng quan về phát triển phần mềm nhúng
- Chương 2. Phát triển phần mềm nhúng nền tảng Arm Linux
- Chương 3. Các kỹ thuật nâng cao cho phát triển phần mềm nhúng
- Chương 4. Cơ chế xây dựng Device Driver

Chương 1

Tổng quan về phát triển

Phần mềm nhúng

Nội dung

- 1.1. Giới thiệu phát triển phần mềm nhúng
- 1.2. Qui trình phát triển phần mềm nhúng
- 1.3. Hệ nhúng nền tảng Arm Linux

1.1. Giới thiệu phát triển phần mềm nhúng

- *Phát triển phần mềm trên hệ thống nhúng phụ thuộc vào nền tảng phần cứng, phần mềm của hệ thống nhúng đó.*
- **Hệ nhúng không dùng hệ điều hành:**
 - Thường sử dụng các vi điều khiển hiệu năng tương đối thấp (8 bit) (8051, ATMega, PIC, ARM7, ...)
 - Phần mềm (firmware) gồm 1 chương trình, phát triển thường bằng C, Assembly
 - Môi trường, công cụ phát triển tùy theo từng dòng vi điều khiển (CodeVision, AVR Studio, Keil, Arduino...)
 - Phù hợp các ứng dụng điều khiển vào/ra cơ bản, các giao tiếp ngoại vi cơ bản.



Giới thiệu phát triển phần mềm nhúng

▪ Hệ nhúng có hệ điều hành:

- Dựa trên các vi điều khiển, vi xử lý (CPU) có hiệu năng cao (Ví dụ: AVR32, ARM 9, ARM 11, ...) (32 bit, 64 bit)
- Nhiều nền tảng hệ điều hành nhúng : FreeRTOS, uCLinux, **Embedded Linux**, Android, ...
- Môi trường, công cụ phát triển tùy thuộc nền tảng hệ điều hành: C/C++, Qt SDK, .Net Compact Framework (Microsoft), ...
- Ứng dụng nhiều bài toán phức tạp: giao tiếp thiết bị, giao tiếp mạng, xử lý ảnh, nhận dạng, ...

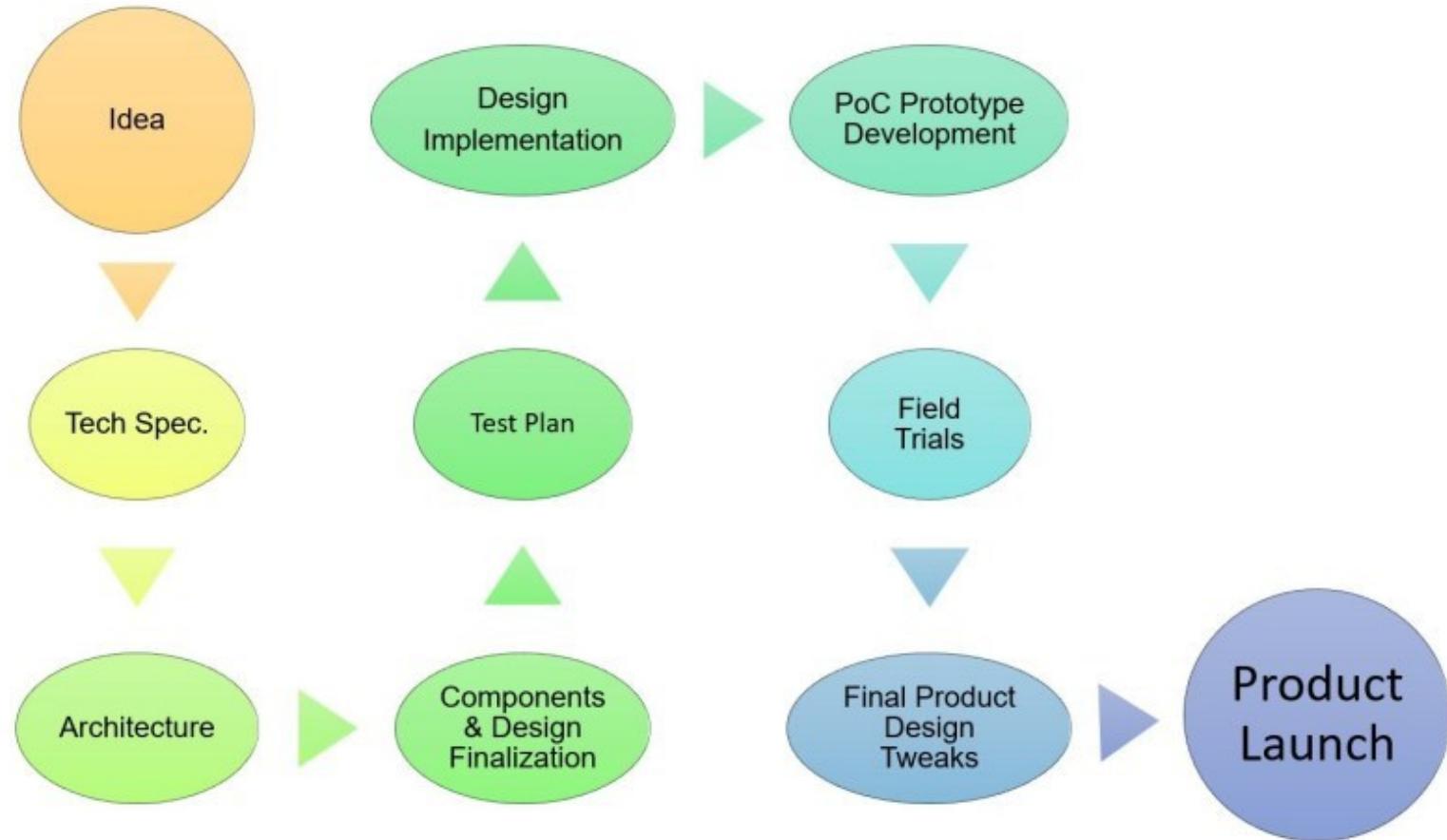
Giới thiệu phát triển phần mềm nhúng

- Học phần này hướng tới
 - Phát triển phần mềm nhúng trên nền tảng **ARM + Linux**
 - Sử dụng C trên Arm Linux, Java (Android)
- Lý do:
 - **ARM** ? Rất phổ biến trong thị trường hệ nhúng hiệu năng cao
 - **Embedded Linux** ? Mã nguồn mở, khả năng can thiệp, hiểu sâu hệ thống. Nhiều OS khác (iOS, Android) dựa trên Linux kernel



1.2. Qui trình phát triển phần mềm nhúng

- 10 steps embedded software development process

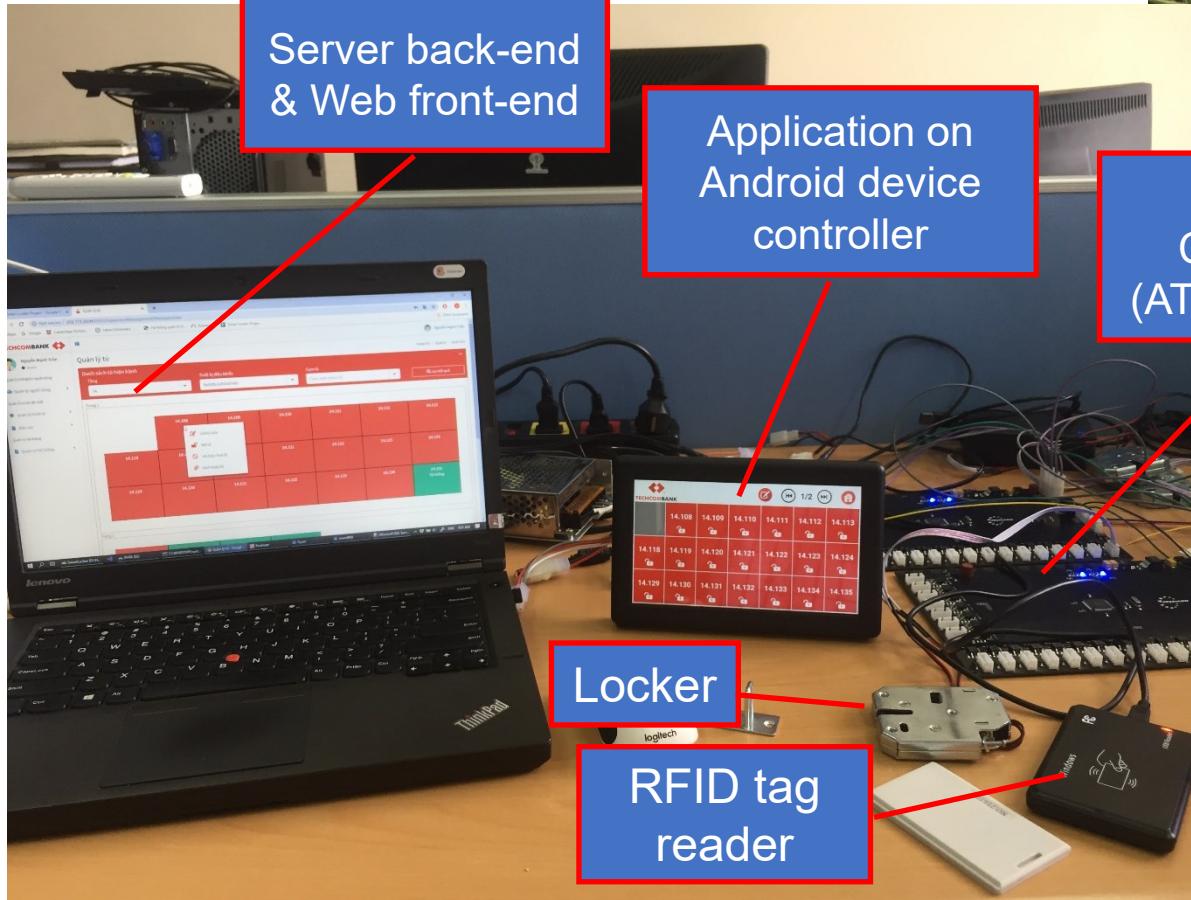


Qui trình phát triển phần mềm nhúng

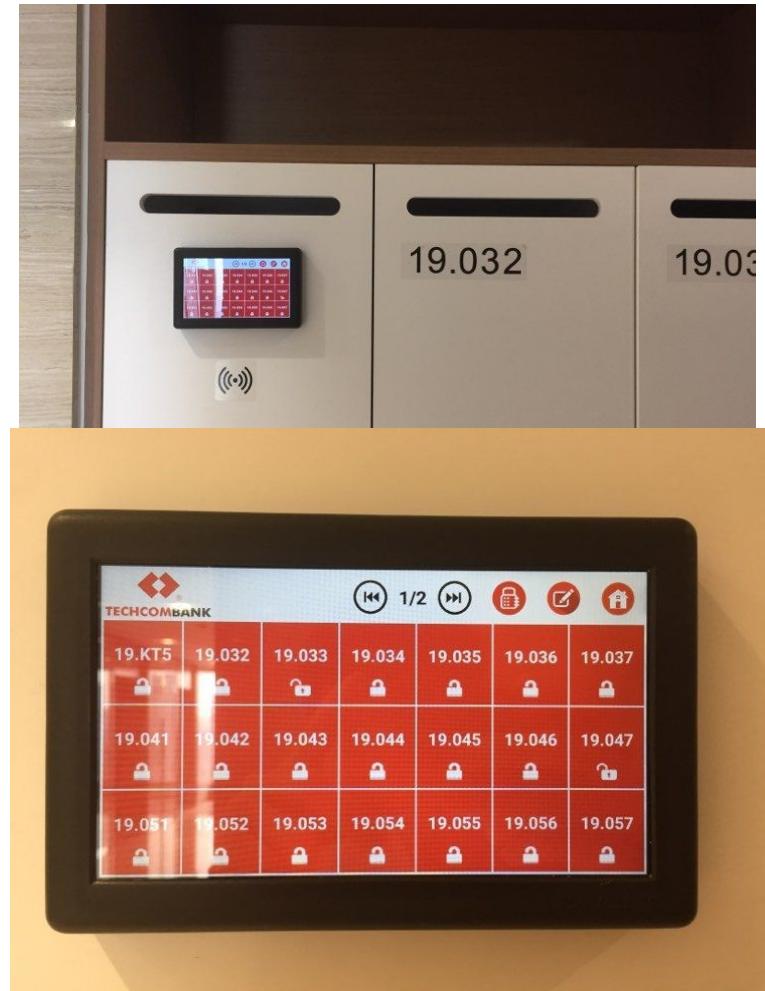
- 10 steps embedded software development process:
 - 1. Ideation/purpose of the product: Lập ý tưởng/mục đích của sản phẩm
 - 2. Technical specification: Phân tích các yêu cầu kỹ thuật
 - 3. Architecting the Solution: Mô hình hóa kiến trúc giải pháp
 - 4. Component section & design finalization: Thiết kế các thành phần hệ thống
 - 5. Test Plan: Kiểm thử đánh giá thiết kế
 - 6. Design implementation: Triển khai thiết kế
 - 7. PoC (Proof-of-Concept/ Prototype Development): Chế tạo mẫu
 - 8. Field Trials: Kiểm thử thực tế tại hiện trường
 - 9. Final Product Improvements: Cải tiến sản phẩm
 - 10. Product Release: Triển khai sản phẩm

Ví dụ 1. Phần mềm hệ thống Smart Locker

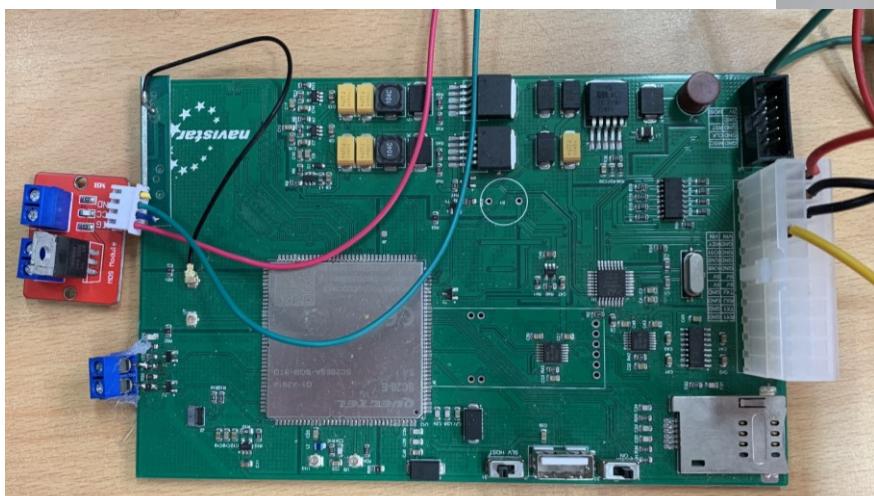
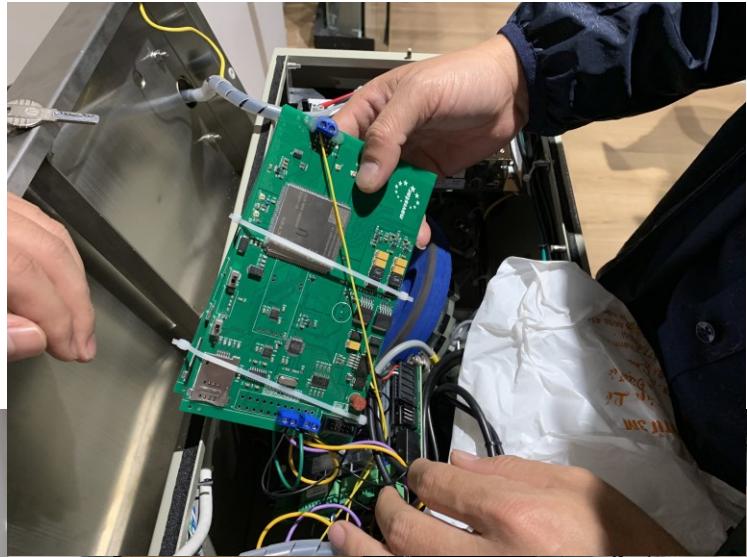
- Mô tả các thành phần phần hệ thống



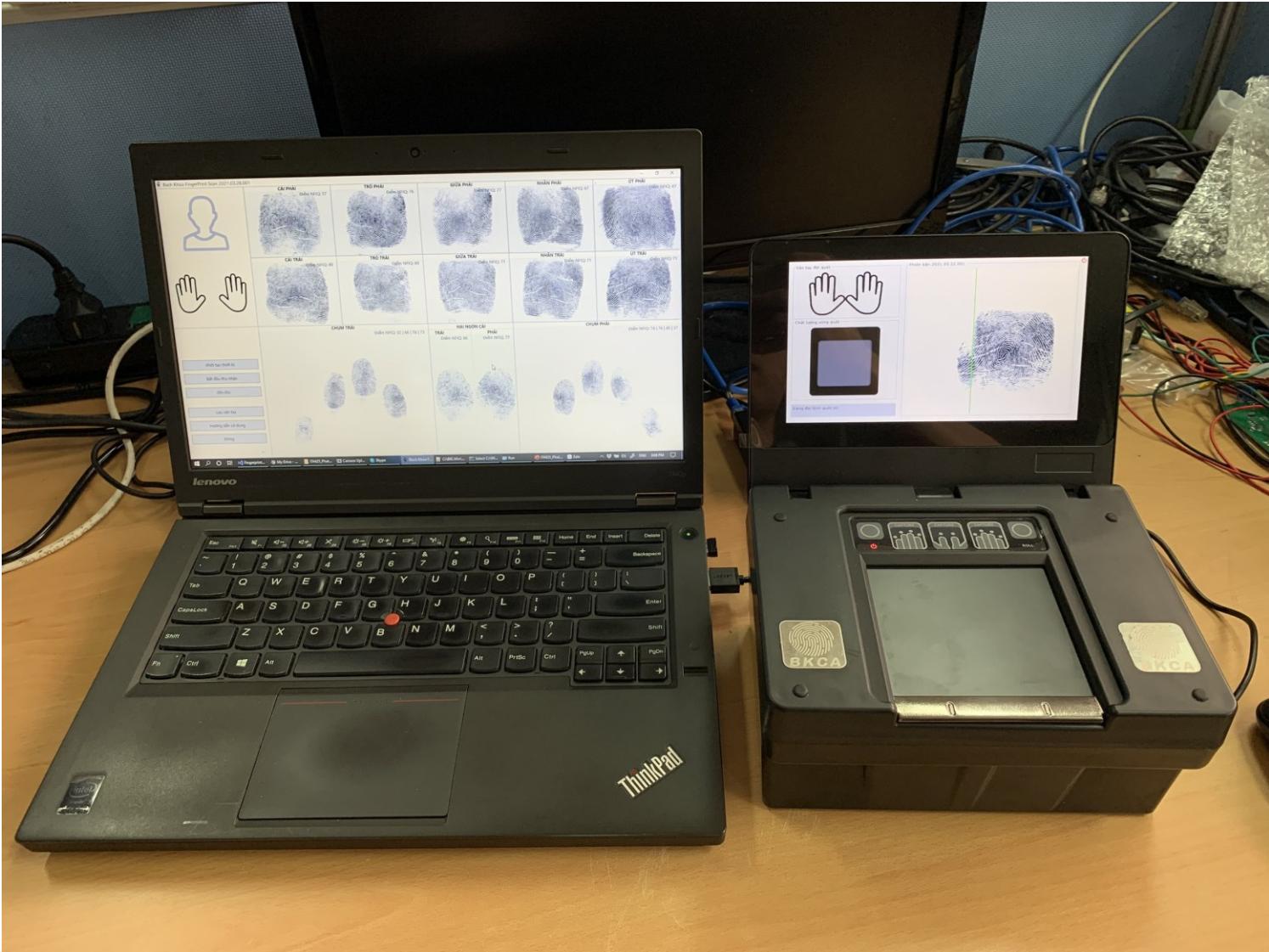
Phần mềm hệ thống Smart Locker



Ví dụ 2. Hệ thống kiểm soát ra vào, chấm công



Ví dụ 3. Phần mềm máy vân tay BKCA



1.3. Hệ thống nhúng nền tảng Arm Linux

- 1.3.1. Giới thiệu kiến trúc ARM
- 1.3.2. Hệ điều hành cho hệ thống nhúng
- 1.3.3. Hệ điều hành nhúng Embedded Linux

1.3.1. Giới thiệu kiến trúc ARM



- ARM:
 - Advanced RISC Machine
 - Acorn RISC Machine
- ARM có kiến trúc tập lệnh RISC 32 bit (Instruction Set Architecture – ISA) phát triển bởi Arm Holdings (1983).
- Phổ biến nhất trong các kiến trúc tập lệnh 32 bit.
- Được sử dụng rộng rãi trong các hệ thống nhúng: mobile phones, PDAs, tablets, digital media, music players, calculators, routers, ...

Các dòng kiến trúc ARM

https://en.wikipedia.org/wiki/ARM_architecture

| Architecture | Core bit-width | Cores | |
|--------------|----------------|--|--|
| | | Arm Ltd. | Third-party |
| ARMv4T | 32 | ARM7TDMI , ARM9TDMI , SecurCore SC100 | |
| ARMv5TE | 32 | ARM7EJ , ARM9E , ARM10E | XScale , FA626TE, Feroceon, PJ1/Mohawk |
| ARMv6 | 32 | ARM11 | |
| ARMv6-M | 32 | ARM Cortex-M0 , ARM Cortex-M0+ , ARM Cortex-M1 , SecurCore SC000 | |
| ARMv7-M | 32 | ARM Cortex-M3 , SecurCore SC300 | Apple M7 |
| ARMv7E-M | 32 | ARM Cortex-M4 , ARM Cortex-M7 | |
| ARMv8-M | 32 | ARM Cortex-M23 ^[61] ARM Cortex-M33 ^[62] | |
| ARMv7-R | 32 | ARM Cortex-R4 , ARM Cortex-R5 , ARM Cortex-R7 , ARM Cortex-R8 | |
| ARMv8-R | 32 | ARM Cortex-R52 | |
| | 64 | ARM Cortex-R82 | |
| ARMv7-A | 32 | ARM Cortex-A5 , ARM Cortex-A7 , ARM Cortex-A8 , ARM Cortex-A9 , ARM Cortex-A12 , ARM Cortex-A15 , ARM Cortex-A17 | Qualcomm Scorpion/Krait , PJ4/Sheeva, Apple Swift (A6 , A6X) |
| ARMv8-A | 32 | ARM Cortex-A32 ^[67] | |
| | 64/32 | ARM Cortex-A35 ^[68] ARM Cortex-A53 , ARM Cortex-A57 ^[69] ARM Cortex-A72 ^[70] ARM Cortex-A73 ^[71] | X-Gene , Nvidia Denver 1/2 , Cavium ThunderX , AMD K12 , Apple Cyclone (A7)/Typhoon (A8 , A8X) (A10 , A10X), Qualcomm Kryo , Samsung M1/M2 |
| | 64 | ARM Cortex-A34 ^[78] | |
| ARMv8.1-A | 64/32 | TBA | Cavium ThunderX2 |
| ARMv8.2-A | 64/32 | ARM Cortex-A55 ^[80] ARM Cortex-A75 ^[81] ARM Cortex-A76 ^[82] ARM Cortex-A77 , ARM Cortex-A78 , ARM Cortex-X1 , ARM Neoverse N1 | Nvidia Carmel , Samsung M4 ("Cheetah"), Fujitsu A64FX (ARMv8 SVE 512-bit) |
| | 64 | ARM Cortex-A65, ARM Neoverse E1 with simultaneous multithreading (SMT), ARM Cortex-A65AE ^[83] | Apple Monsoon+Mistral (A11) (September 2017) |
| ARMv8.3-A | 64 | TBA | Apple Vortex+Tempest (A12 , A12X , A12Z), Marvell ThunderX3 (v8.3+) ^[87] |
| ARMv8.6-A | 64 | TBA | Apple Firestorm+Icestorm (A14 , M1) |

ARM: Tối ưu với lệnh điều kiện

Pseudo assembly

```
int gcd(int a, int b) {  
    while (a != b)  
        if (a > b)  
            a -= b;  
        else  
            b -= a;  
    return a;  
}
```

C language

Arm assembly language

```
loop:  
// Compare a and b  
GT = a > b;  
LT = a < b;  
NE = a != b;  
//Perform operations based on flag results  
if (GT) a -= b;  
if (LT) b -= a;  
if (NE) goto loop;  
return a;
```

;assign a to register r0, b to r1

```
loop : CMP    r0, r1  
        ;  
        ;  
        ;  
SUBGT  r0, r0, r1  
        ;  
        ;  
SUBLT  r1, r1, r0  
        ;  
        ;  
BNE    loop  
        ;  
        ;  
        ;  
B      lr
```

;set condition "NE" if (a != b),
; "GT" if (a > b),
; or "LT" if (a < b)
; if "GT" (Greater Than), a = a - b;
; if "LT" (Less Than), b = b - a;
; if "NE" (Not Equal), then loop
; if the loop is not entered, we can
safely return

ARM: Tối ưu lệnh số học, logic

- Ví dụ lệnh C:

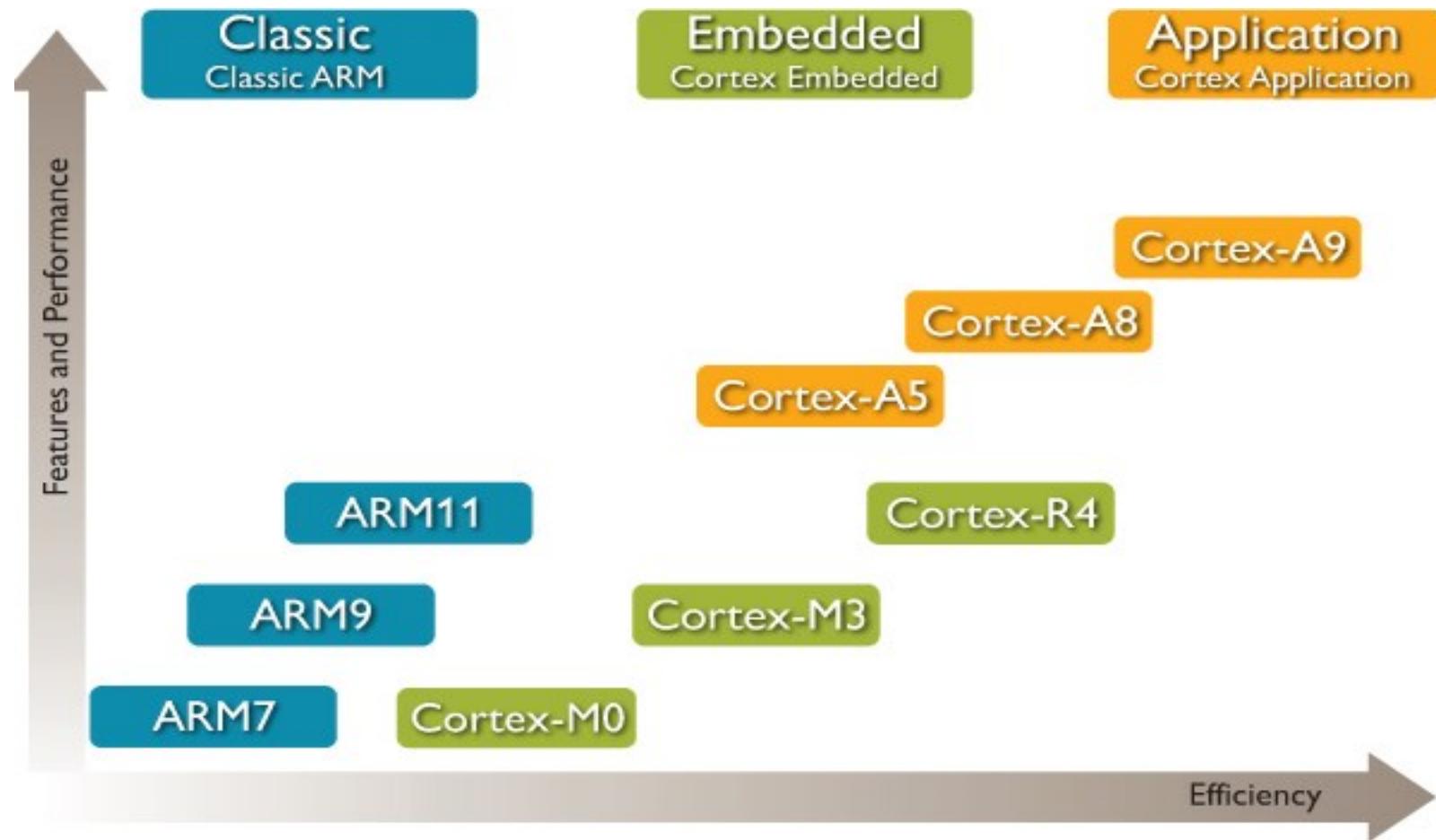
```
a += (j << 2);
```

- Complied into ARM instruction:

```
ADD Ra, Ra, Rj, LSL #2
```

single-word, single-cycle instruction

So sánh các dòng ARM

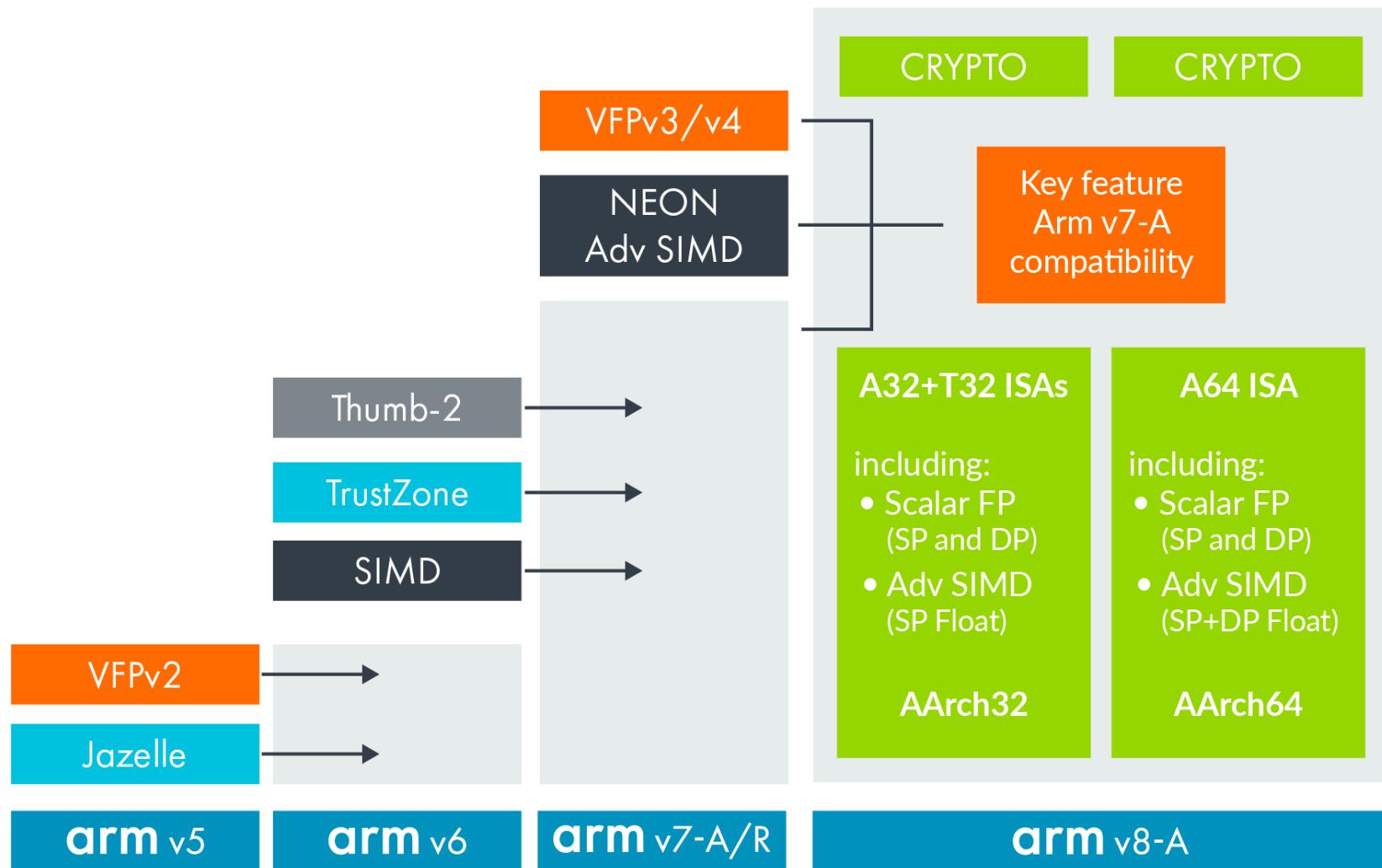


Phân nhóm theo hiệu năng và tính hữu dụng

Một số công nghệ trong ARM

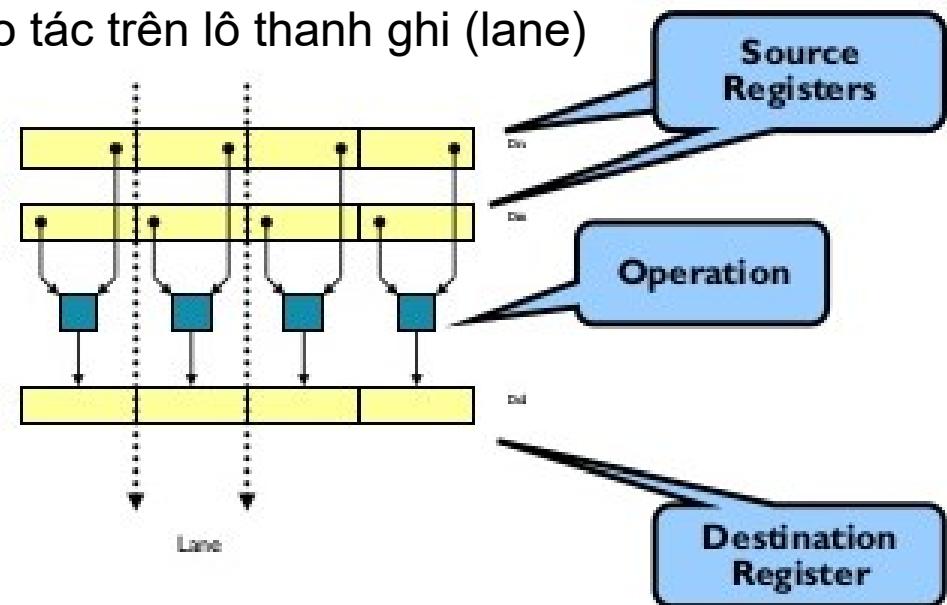
- Thumb Instruction Set: tập lệnh 16 bit cho phép tăng mật độ lệnh
- Jazelle: công nghệ cho phép tăng tốc các ứng dụng viết bằng Java
- SIMD, NEON:
 - SIMD = Single Instruction Multiple Data
 - NEON: wide SIMD data processing architecture
- TrustZone: công nghệ nâng cao tính bảo mật

Một số công nghệ trong ARM



NEON ?

- NEON: kiến trúc xử lý dữ liệu wide SIMD (single instruction multiple data)
 - Mở rộng của kiến trúc tập lệnh ARM
 - 32 registers, 64-bit wide (AArch64: 128-bit wide)
- Các lệnh NEON thực hiện xử lý dữ liệu theo lô “Packet SIMD”
 - Các kiểu dữ liệu: signed/unsigned 8/16/32/64 bit, single precision float (AArch64: double precision float)
 - Các lệnh thực hiện cùng thao tác trên lô thanh ghi (lane)



Các ứng dụng của công nghệ NEON



Watch any video in any format



Game processing



Edit & Enhance captured videos
Video stabilization



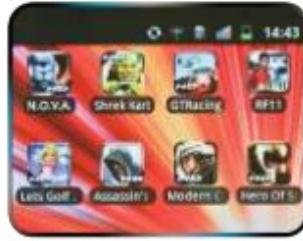
Process megapixel photos quickly



Antialiased rendering & compositing



Voice recognition



Advanced User Interfaces



Powerful multichannel hi-fi audio processing

Kiến trúc ARM và lịch sử phát triển

- ARM được rất nhiều hãng phát triển và sản xuất, ở Việt Nam phổ biến chip ARM của các hãng
 - ATTEL: AT91SAM7, AT91SAM9...
 - NXP: LPC2138, LPC2148, LPC2300...
 - TI (Texas Instrument): TMS470, TMS570...
 - SAMSUNG: S3C2440
 - ...

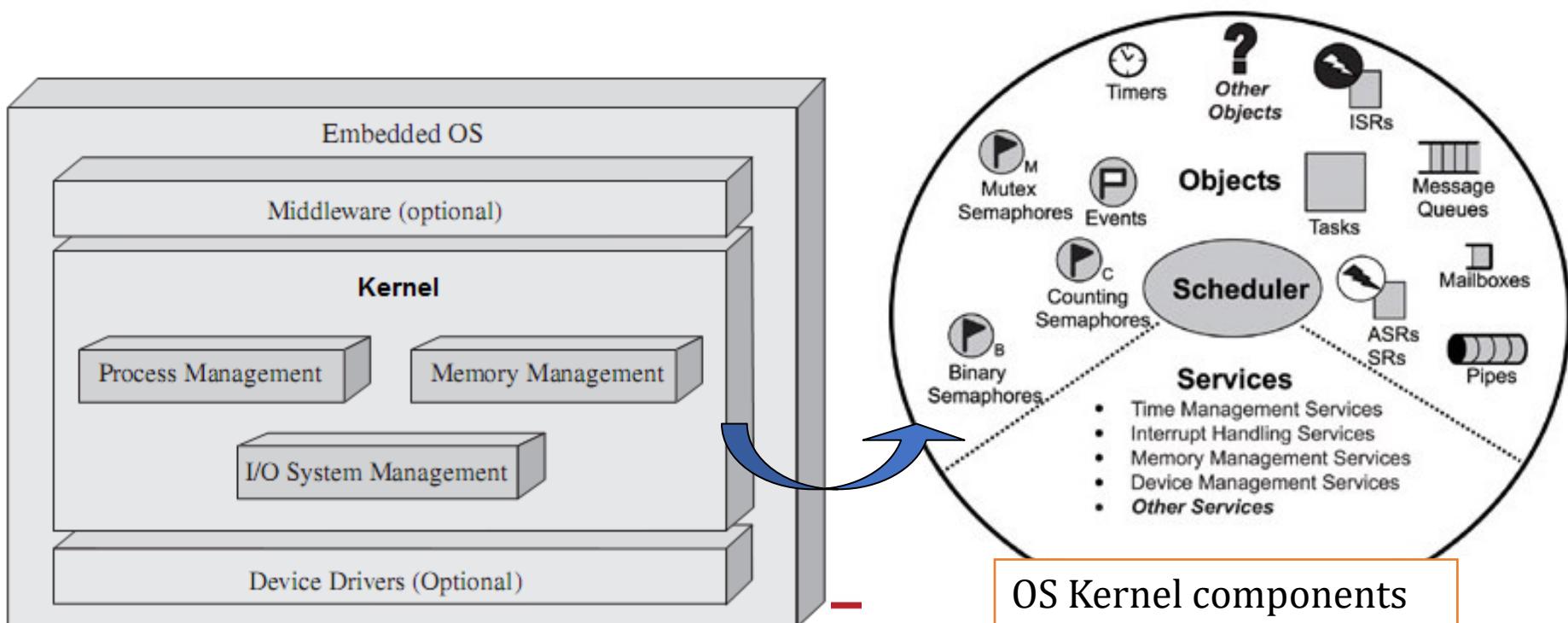
1.3.2. Hệ điều hành cho hệ thống nhúng

- **Hệ điều hành có thể:**

- Cho phép nhiều chương trình chạy đồng thời (multi-tasking)
- Quản lý tài nguyên phần cứng, cung cấp dịch vụ cho các chương trình khác nhau
- Cho phép phát triển các phần mềm phức tạp (cung cấp thư viện nền tảng lập trình, API)

OS Kernel

- Kernel (“Nhân”) hệ điều hành: thành phần chứa các chức năng chính:
 - Quản lý tiến trình (Process Management)
 - Quản lý bộ nhớ (Memory Management)
 - Quản lý hệ thống vào ra (I/O System Management)

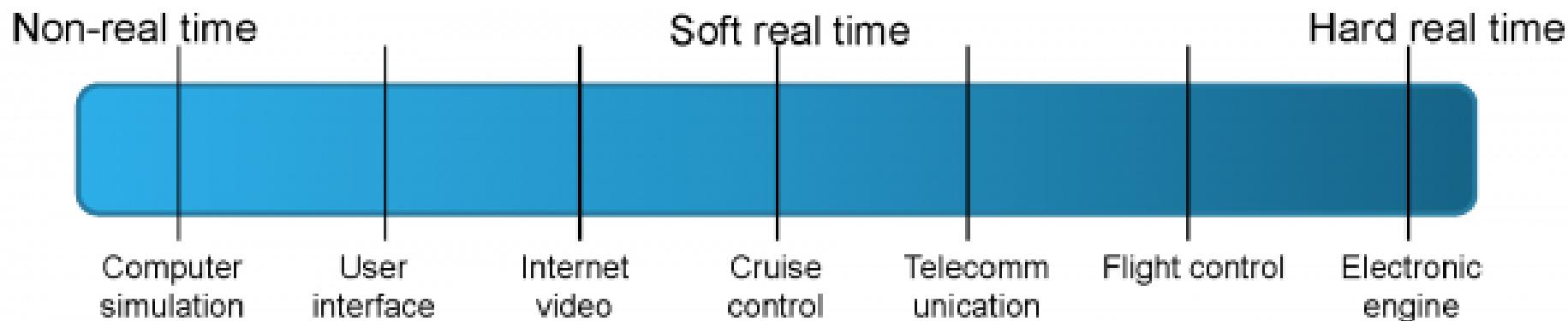


Các đặc trưng hệ điều hành nhúng

- Tính tin cậy (reliability)
- Tính khả chuyển (portability)
- Tính linh hoạt (flexibility): dễ dàng tùy chỉnh (nâng cấp hay thu gọn) để tương thích với nền tảng hệ thống.
- Khả năng nén, nhỏ gọn (compact), đòi hỏi ít bộ nhớ
- Cơ chế lập lịch hỗ trợ thời gian thực

RTOS – Real Time Operating System

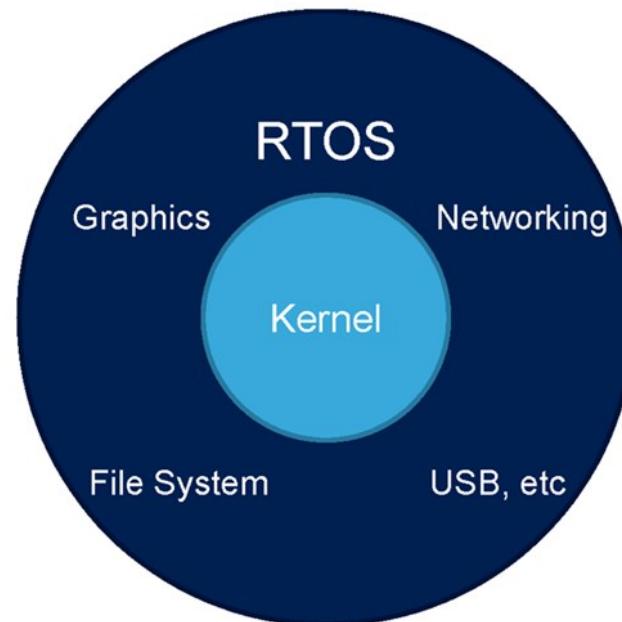
- Real time là gì ?
 - Một hệ thống có phản ứng thích hợp, đúng thời điểm với môi trường của nó
 - “Đủ nhanh” trong một ngữ cảnh
- Ví dụ cấp độ từ Non-real time đến Hard real time



RTOS – Real Time Operating System

- Hệ điều hành thời gian thực:

- Thường dùng trong các ứng dụng hệ thống nhúng có giới hạn tài nguyên hạn chế và yêu cầu ngặt nghèo về thời gian đáp ứng, tính sẵn sàng và khả năng tự kiểm soát một cách chính xác.



RTOS – Real Time Operating System

- Một hệ thống nhúng có thể chạy nhiều task (tác vụ) khác nhau.
- Một task là một chương trình thường chạy trong một vòng lặp vô hạn
- Ví dụ máy bán nước tự động:
 - Task quản lý việc lựa chọn của người dùng
 - Task để kiểm tra đúng số tiền người dùng đã trả
 - Task để điều khiển động cơ

Một số hệ điều hành nhúng RTOS

■ FreeRTOS:

The screenshot shows the official FreeRTOS website. At the top left is the FreeRTOS logo. To its right is a navigation bar with links: Quick Start, Supported MCUs, Books & Kits, Trace Tools, Ecosystem, TCP & FAT, Training, About, Contact, Support, FAQ, and Download. A green box on the right contains the text "FreeRTOS Tutorial Books and Reference Manuals" and "Become an expert while supporting the FreeRTOS project". Below the navigation bar, there's a sidebar with links to Home, FreeRTOS Books and Manuals, and FreeRTOS Interactive. The main content area features the text "FreeRTOS™" and "The Market Leading, De-facto Standard and Cross Platform Real Time Operating System (RTOS). Don't Let Your RTOS Lock You In." It also mentions that FreeRTOS is developed in partnership with leading chip companies over a 12-year period. On the far right, there's a "Buildable Examples" section with options for FreeRTOS+TCP and FreeRTOS+FAT, each accompanied by a small icon.

■ uLinux

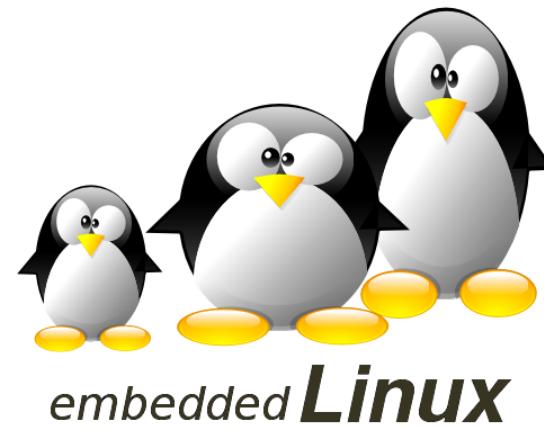
The screenshot shows the uClinux website. At the top left is a penguin wearing a t-shirt with "μC" on it. To its right is a navigation bar with links: Home, What is uClinux?, and Status. The main content area features the text "The Embedded Linux/Microcontroller project is a port of Linux to systems without a Memory Management Unit (MMU)." Below this, there's a detailed description of the project, mentioning its pronunciation ("you-see-linux"), origin ("mu" and "C"), and first target system ("Motorola MC68328: DragonBall Integrated Microprocessor"). It also notes that the first boot was on a "PalmPilot" using a "TRG SuperPilot Board". On the right side, there's a large "μClinux" logo and the text "Embedded Linux/Microcontroller Project".

■ TinyOS



1.3.3. Hệ điều hành nhúng Embedded Linux

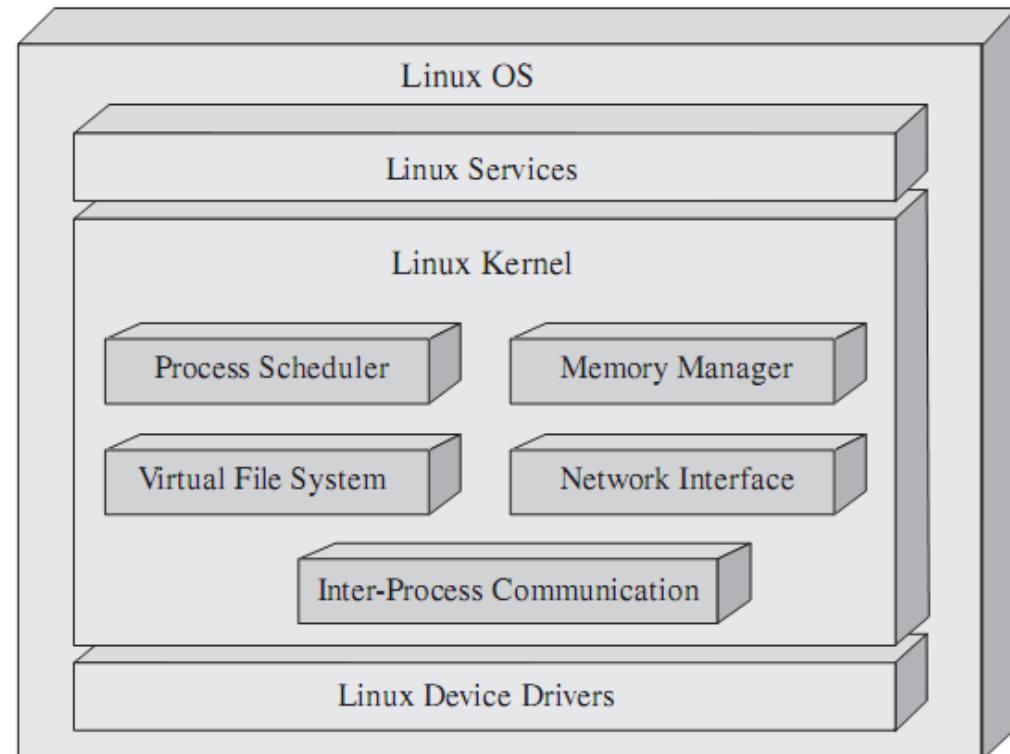
- Linux đã được porting vào nhiều nền tảng kiến trúc:
 - X86, ARM, PowerPC, MIPS, AVR32, ...
 - 32 bit, 64 bit
- Mã nguồn mở (open source), có thể tùy chỉnh cho phù hợp với yêu cầu riêng
- Hỗ trợ đủ chức năng thiết yếu của hệ điều hành



Hệ điều hành nhúng Embedded Linux

- Các thành phần chính:

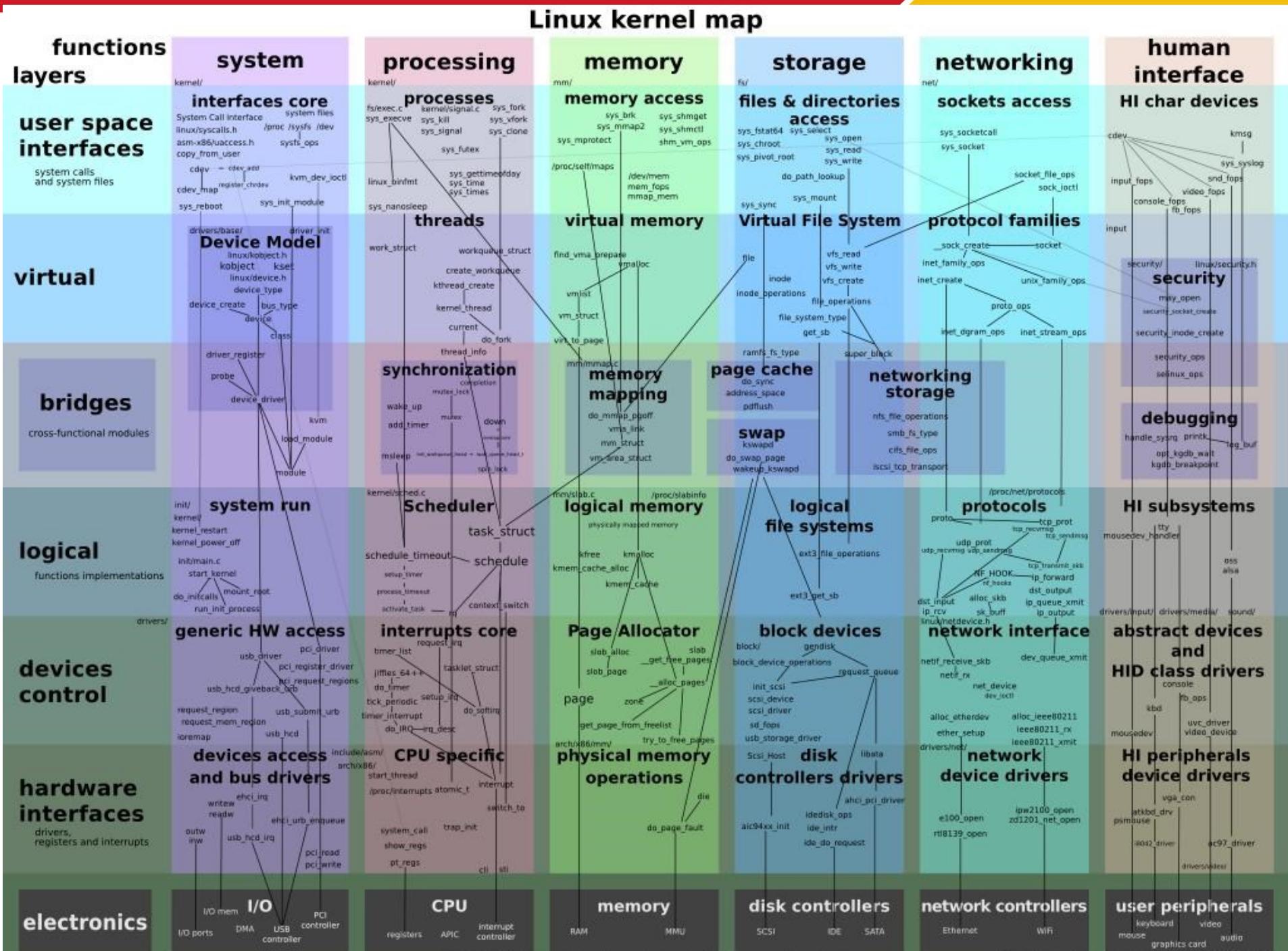
- Memory Manager
- Scheduler
- File System
- Network Interface
- Inter Process Communication (IPC)
- Device Drivers
- Services



Embedded Linux Model

Hệ điều hành nhúng Embedded Linux

- **Linux Kernel:** Nhiệm vụ nói chuyện với phần cứng và phần mềm, quản lý các tài nguyên của hệ thống tốt nhất có thể
- Kernel giao tiếp với phần cứng thông qua drivers, là thành phần trong nhân hoặc được cài đặt bổ sung theo cơ chế kernel modules



Linux Kernel history

- 2.4 has been around forever, was “end-of-lifed” Dec 2011.
- 2.6 is still active, but no new releases planned
- 3.0 isn’t a radical change from 2.6, instead a 2.6 upgrade was move to 3.0 for Linux’s 20th anniversary.
 - [<https://lkml.org/lkml/2011/5/29/204>]
 - And 4.0 is more of the same.

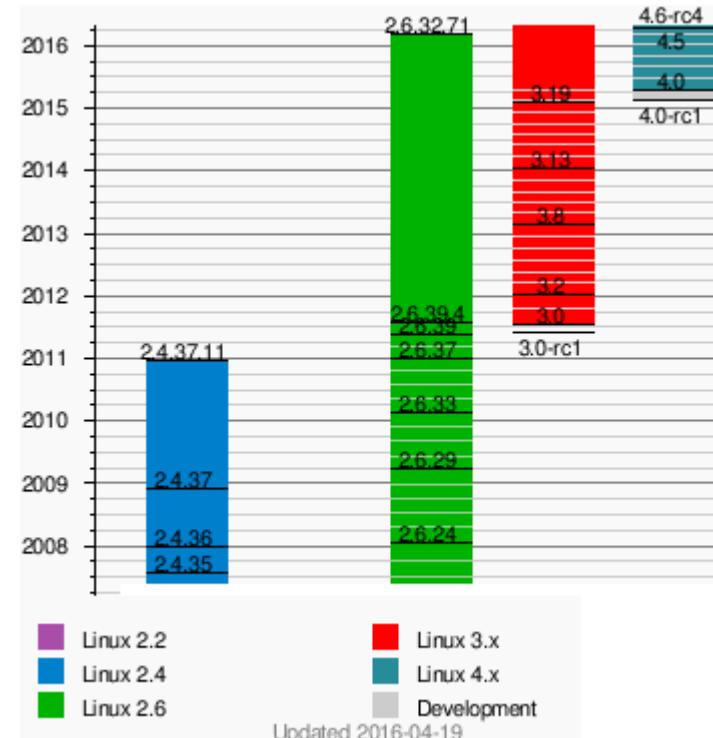


Figure modified from “Linux kernel” article on Wikipedia

<http://www.itworld.com/article/2887558/linus-torvalds-bumps-linux-kernel-to-version-4x.html>

Chương 2

Phát triển phần mềm nhúng trên nền tảng ARM Linux

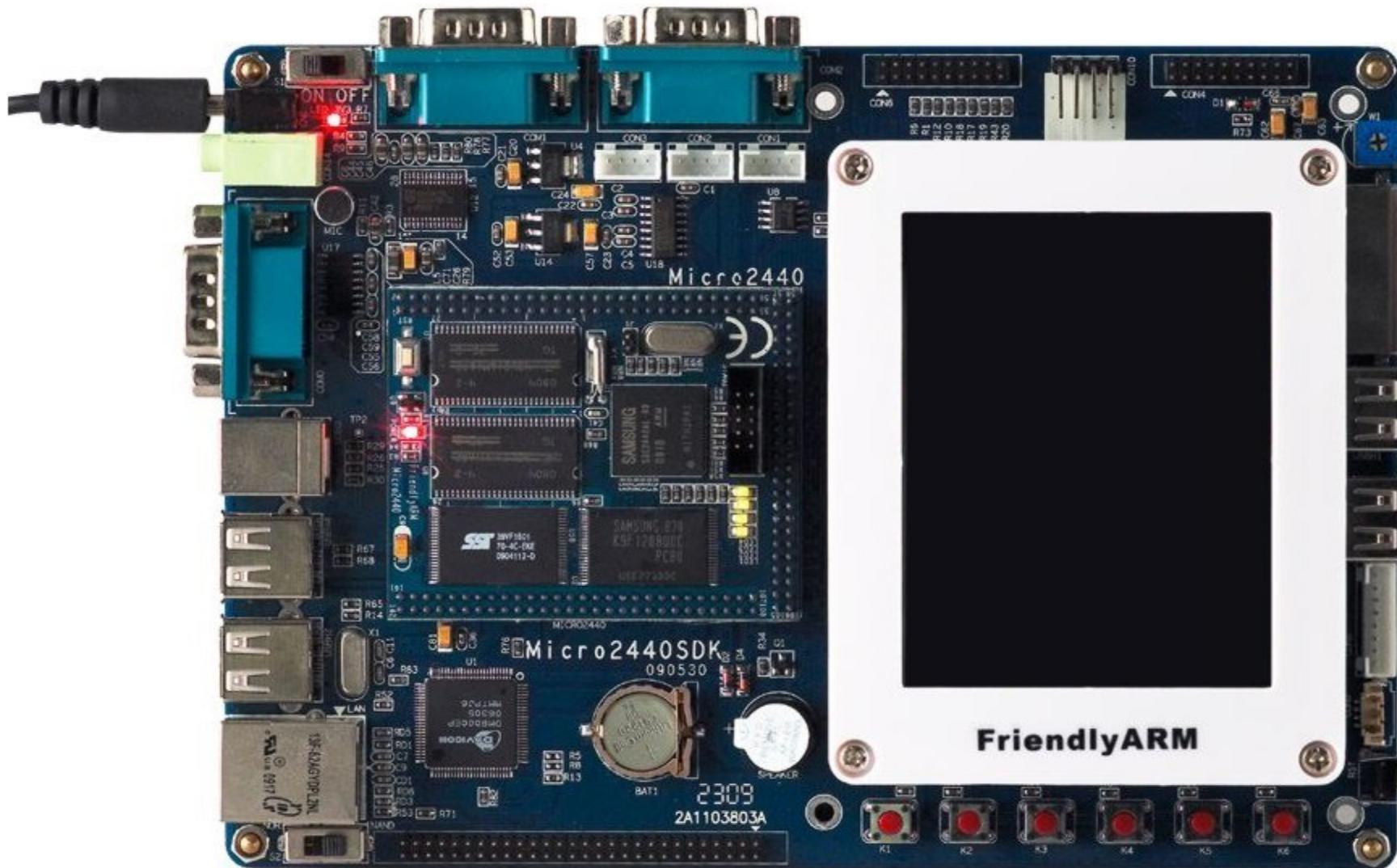
Nội dung

- 2.1. KIT phát triển
- 2.2. Môi trường phát triển ứng dụng
- 2.3. Cơ chế giao tiếp thiết bị
- 2.4. Giao tiếp vào ra cơ bản GPIO
- 2.5. Giao tiếp thiết bị chuẩn RS232
- 2.6. Giao tiếp thiết bị chuẩn USB

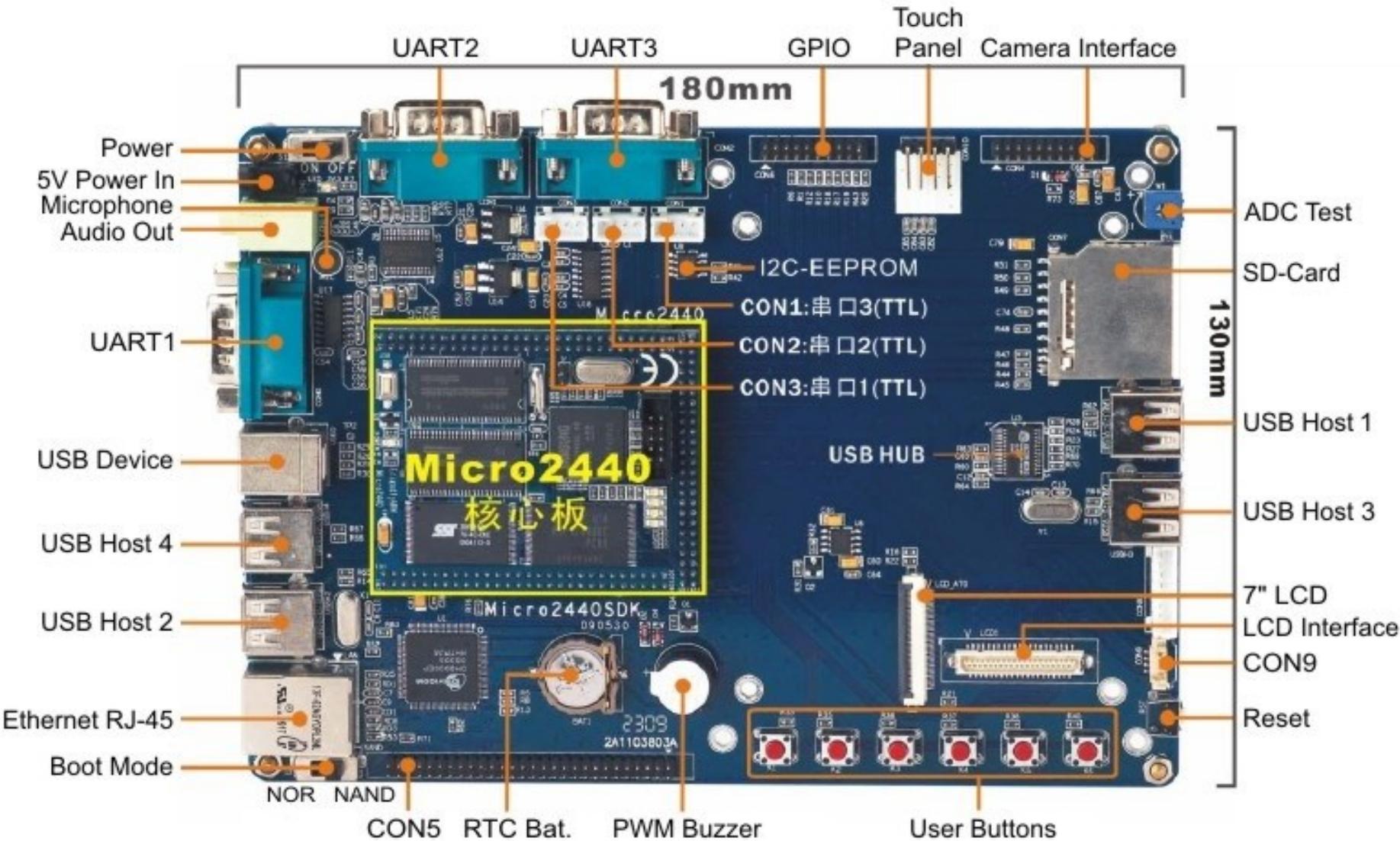
2.1. KIT phát triển

- A. Sử dụng KIT FriendlyARM (ARM + Embedded Linux)
 - Biên dịch nhân Embedded Linux
 - Cài đặt Embedded Linux
- B. Sử dụng Raspberry Pi 4 (ARM + Raspbian OS)

KIT FriendlyARM Micro2440



KIT FriendlyARM Micro2440



KIT FriendlyARM Micro2440

Core Board: CPU S3C2440, NAND & NOR Flash, SDRAM



Biên dịch nhân Embedded Linux

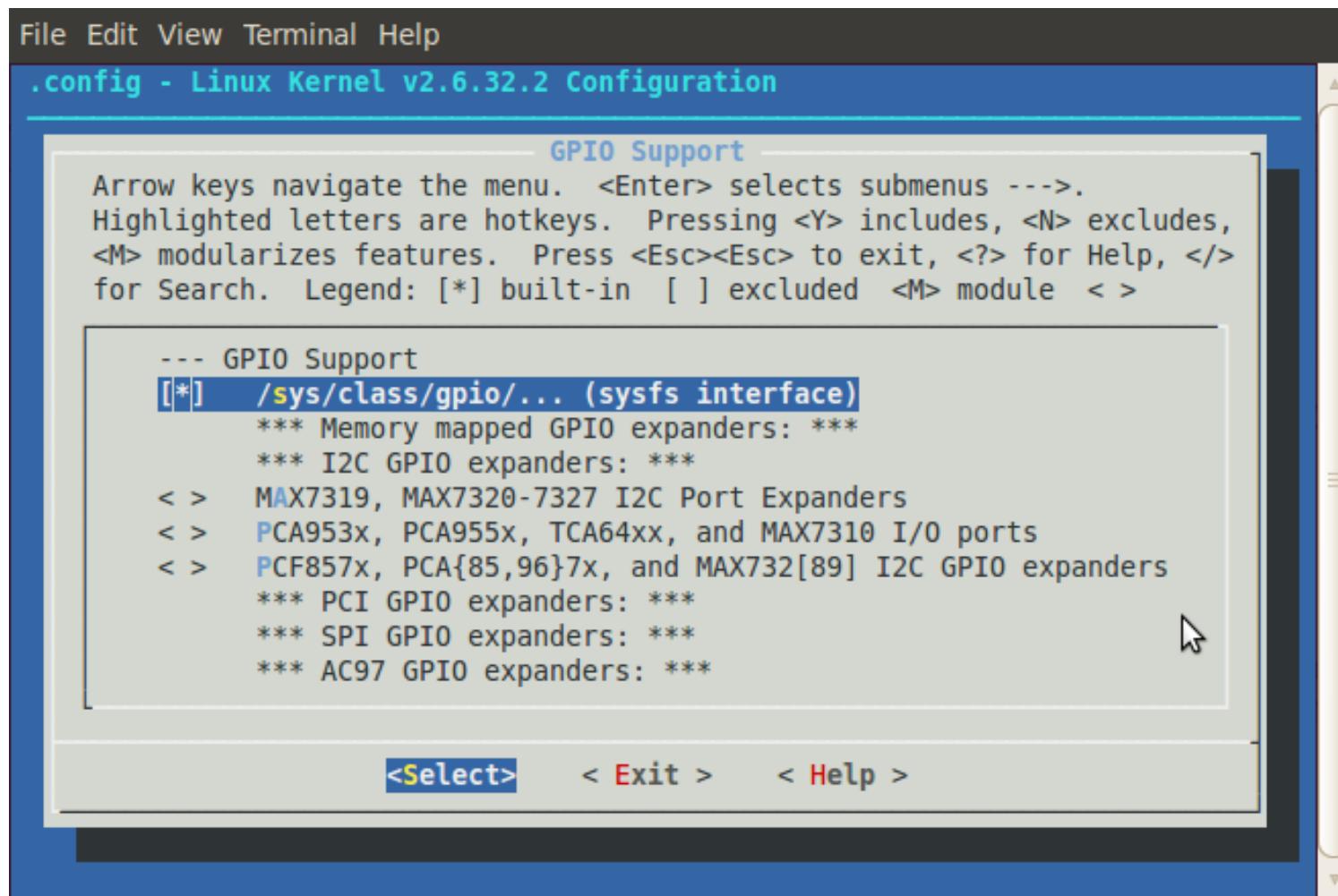
- Linux Kernel:
 - Mã nguồn mở, được phân phối tại kernel.org, gồm nhiều phiên bản.
 - Có thể sử dụng phiên bản đã được phát triển mẫu đi kèm hệ nhúng (Linux Kernel 2.6.32 for ARM micro2440)
- Khi nào cần biên dịch nhân ?
 - Cần cấu hình tùy biến, chỉnh sửa các thành phần (gỡ bỏ, thêm mới) cho phù hợp với nhu cầu sử dụng, với hệ thống phần cứng đang có.

Biên dịch nhân Embedded Linux

- Qui trình biên dịch nhân Linux cho KIT ARM micro2440:
 - Giải nén file mã nguồn nhân linux-2.6.32.2.tar được thư mục tương ứng
 - Cấu hình trước khi biên dịch:
 - Thông tin cấu hình lưu trong file .config (file ẩn)
 - Có thể sử dụng file cấu hình mẫu có sẵn bằng cách chép đè lên file .config qui định. Lệnh:
cp mini2440_config_x35 .config
 - Sử dụng giao diện cấu hình để tùy biến các thành phần. Lệnh: **make menuconfig**
 - Thực hiện biên dịch bằng lệnh: **make zImage**
- Biên dịch thành công kết quả sẽ là file zImage (trong thư mục linux-2.6.32.2/arch/arm/mach-s3c2440), sẽ được nạp (porting) xuống bộ nhớ NAND Flash của Micro2440

Biên dịch nhân Embedded Linux

▪ Giao diện đồ họa cấu hình biên dịch nhân Linux



Cài đặt Embedded Linux

- Quá trình boot Linux trên PC và trên hệ nhúng:

Boot Linux trên PC

Boot Linux trên hệ nhúng

Power-up / Reset

System startup

BIOS / BootMonitor

Stage 1 bootloader

Master Boot Record

Stage 2 bootloader

LILO, GRUB, etc.

Kernel

Linux

Init

User-space

Operation

Boot Loader
(U-Boot, Supervivi...)

Kernel

Init

Cài đặt Embedded Linux

- **Bootloader:** chương trình mồi, thực hiện kiểm tra phần cứng hệ thống và nạp nhân (kernel) của hệ điều hành
- **Kernel:** nhân hệ điều hành, chứa các thành phần cơ bản nhất
- **Root file system:** hệ thống file, chứa các modules bổ sung và các phần mềm ứng dụng

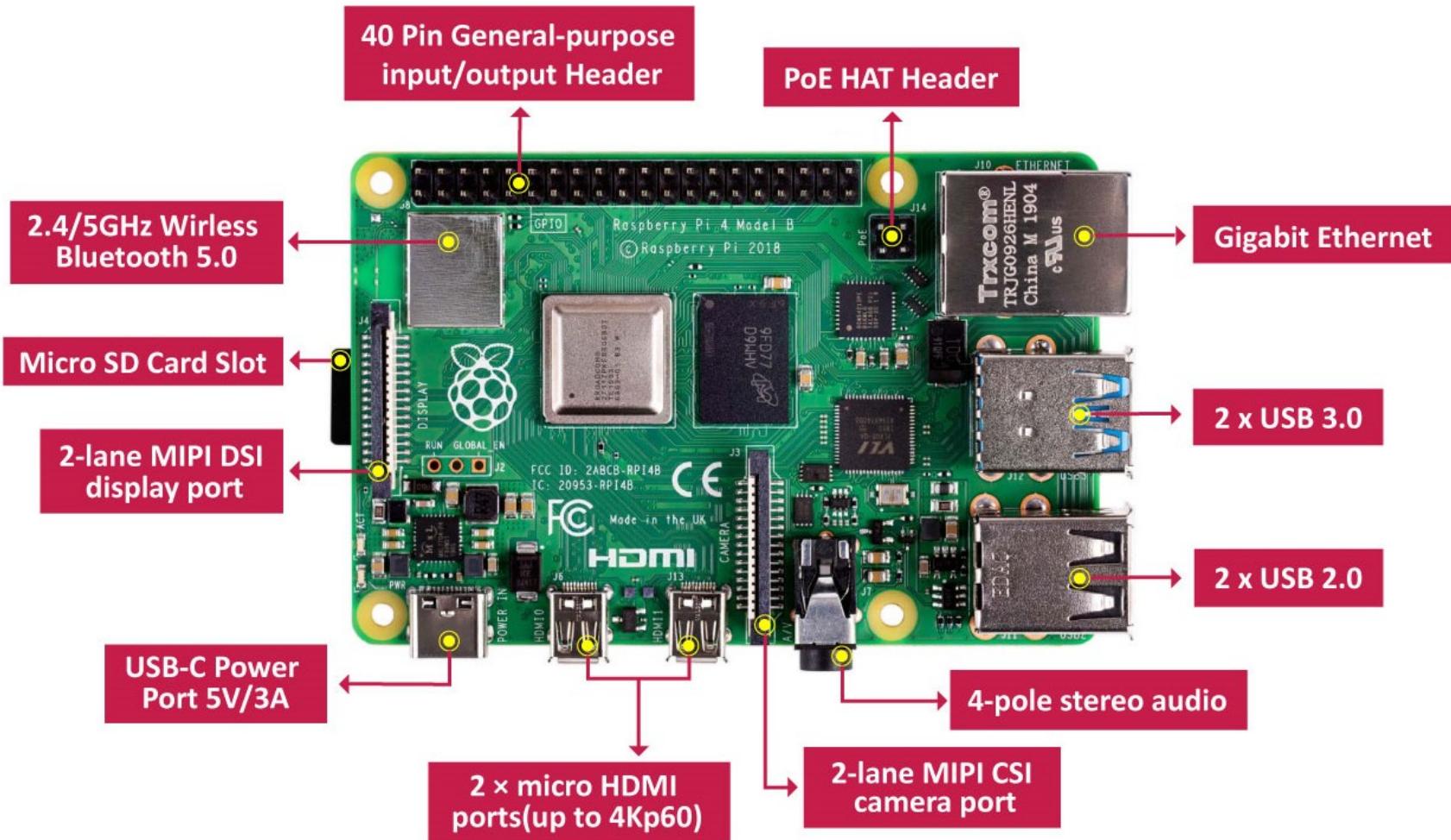
Cài đặt Embedded Linux

- Quá trình cài đặt Embedded Linux cho Micro2440:
 - Bước 1. Chép chương trình bootloader lên bộ nhớ NAND Flash (địa chỉ bắt đầu 0x00000000)
 - Bước 2. Chép kernel Image lên bộ nhớ NAND Flash
 - Bước 3. Chép root files system
- Quá trình cài đặt (chép các thành phần) được điều khiển thông qua một chương trình điều khiển chứa trong NOR Flash

Cài đặt Embedded Linux

- Công cụ cài đặt Embedded Linux cho micro2440:
 - Phần mềm giao tiếp cổng com giữa PC và KIT:
 - GtkTerm, hoặc minicom (trên Ubuntu PC) hoặc Hyper Terminal (trên Windows PC)
 - Nhiệm vụ giao tiếp với chương trình điều khiển trên NOR Flash, ra lệnh trong quá trình cài đặt
 - Phần mềm truyền file từ PC xuống KIT (qua usb):
 - Nhiệm vụ chép các file cần thiết lên bộ nhớ NAND Flash
 - usbpush trên Ubuntu hoặc DNW trên Windows

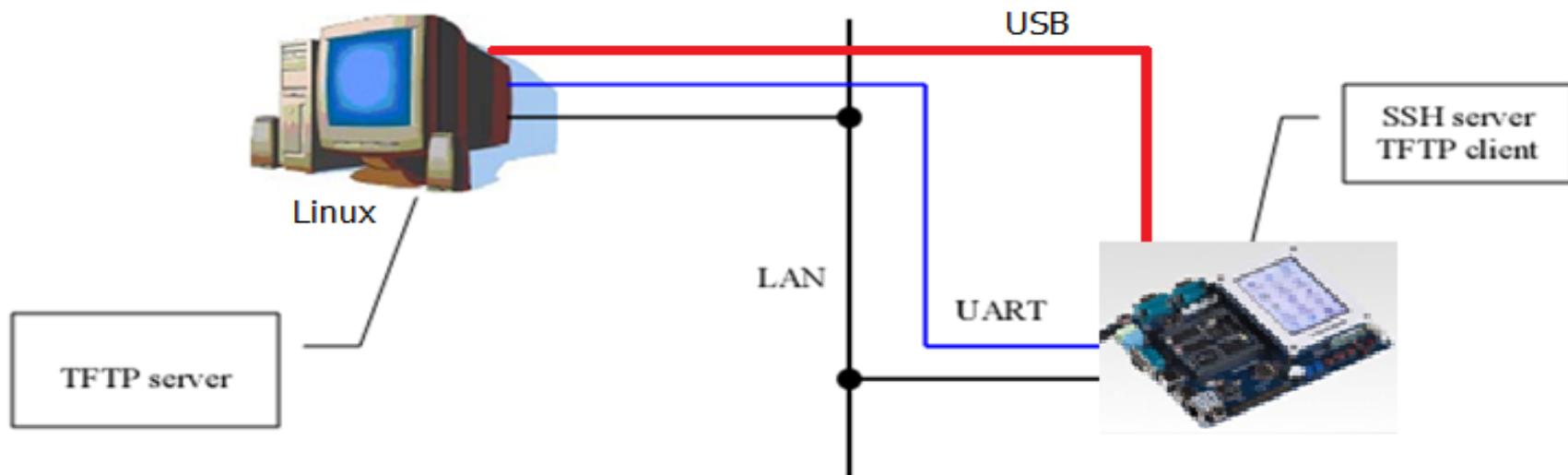
Raspberry Pi



<https://raspberrypi.vn/huong-dan-cai-dieu-hanh-cho-raspberry-pi-2457.pi>

2.2. Môi trường phát triển ứng dụng

- Máy host cài hệ điều hành Ubuntu
- Trình biên dịch chéo Cross toolchains (arm-linux-gcc 4.4.3): biên dịch ứng dụng (viết bằng C/C++)
- Công cụ viết mã nguồn chương trình C (dùng gedit, eclipse, ...)
- gFTP: truyền nhận file Host<->KIT qua giao thức FTP
- Telnet: kết nối KIT qua Ethernet (sử dụng cross cable)



Môi trường phát triển ứng dụng

- Cài đặt trình biên dịch chéo arm-linux-gcc
 - **Bước 1:** Giải nén arm-linux-gcc-4.4.3.tar.gz
tar -zxvf arm-linux-gcc-4.4.3.tar.gz
 - **Bước 2:** Cập nhật biến môi trường PATH
 - Thêm đường dẫn tới thư mục **bin** của arm-linux-gcc-4.4.3 (**Cập nhật biến môi trường PATH trong file .bashrc trong thư mục người dùng \$HOME**)
 - **Bước 3:** Kiểm tra trình biên dịch bằng lệnh
arm-linux-gcc --version
- Cấu hình mạng LAN (host + KIT) qua cáp chéo và sử dụng IP cùng dải:
 - Linux host: 192.168.1.30
 - Linux target: 192.168.1.230 (default)

Ví dụ chương trình Hello

- Soạn thảo mã nguồn chương trình (sử dụng gedit, eclipse, v.v...)

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    printf("Ten chuong trinh la '%s'.\n", argv[0]);
    printf("Chuong trinh co %d tham so \n", argc - 1);
    /* Neu co bat cu tham so dong lenh nao*/
    if (argc > 1) {
        /* Thi in ra*/
        int i;
        printf("Cac tham so truyen vao la : \n");
        for (i = 1; i < argc; ++i)
            printf(" Tham so %d: %s\n", i, argv[i]);
    }
    return 0;
}
```

Biên dịch chéo chương trình

- **Cách 1:** Sử dụng lệnh của cross compiler
 - VD: `arm-linux-gcc -o Hello Hello.c`
 - Kết quả: biên dịch ra một file thực thi có tên là Hello từ một file mã nguồn là Hello.c, file này có hỗ trợ khả năng debug
- **Cách 2:** Tạo và sử dụng Makefile
 - make là một tool cho phép quản lý quá trình biên dịch, liên kết ... của một dự án với nhiều file mã nguồn.
 - Tạo Makefile lưu các lệnh biên dịch theo định dạng của Makefile
 - Sử dụng lệnh `make` để chạy Makefile và biên dịch chương trình

Cấu trúc Makefile

- Makefile cấu thành từ các target, variables và comments
- Target có cấu trúc như sau:

target: dependencies

[tab] system command

- target: make target
- Dependencies: các thành phần phụ thuộc (file mã nguồn, các file object...)
- System command: các câu lệnh (lệnh biên dịch, lệnh linux)

Ví dụ Makefile đơn giản

```
CC=arm-linux-gcc
```

```
all: Hello.c
```

```
$(CC) -o Hello Hello.c
```

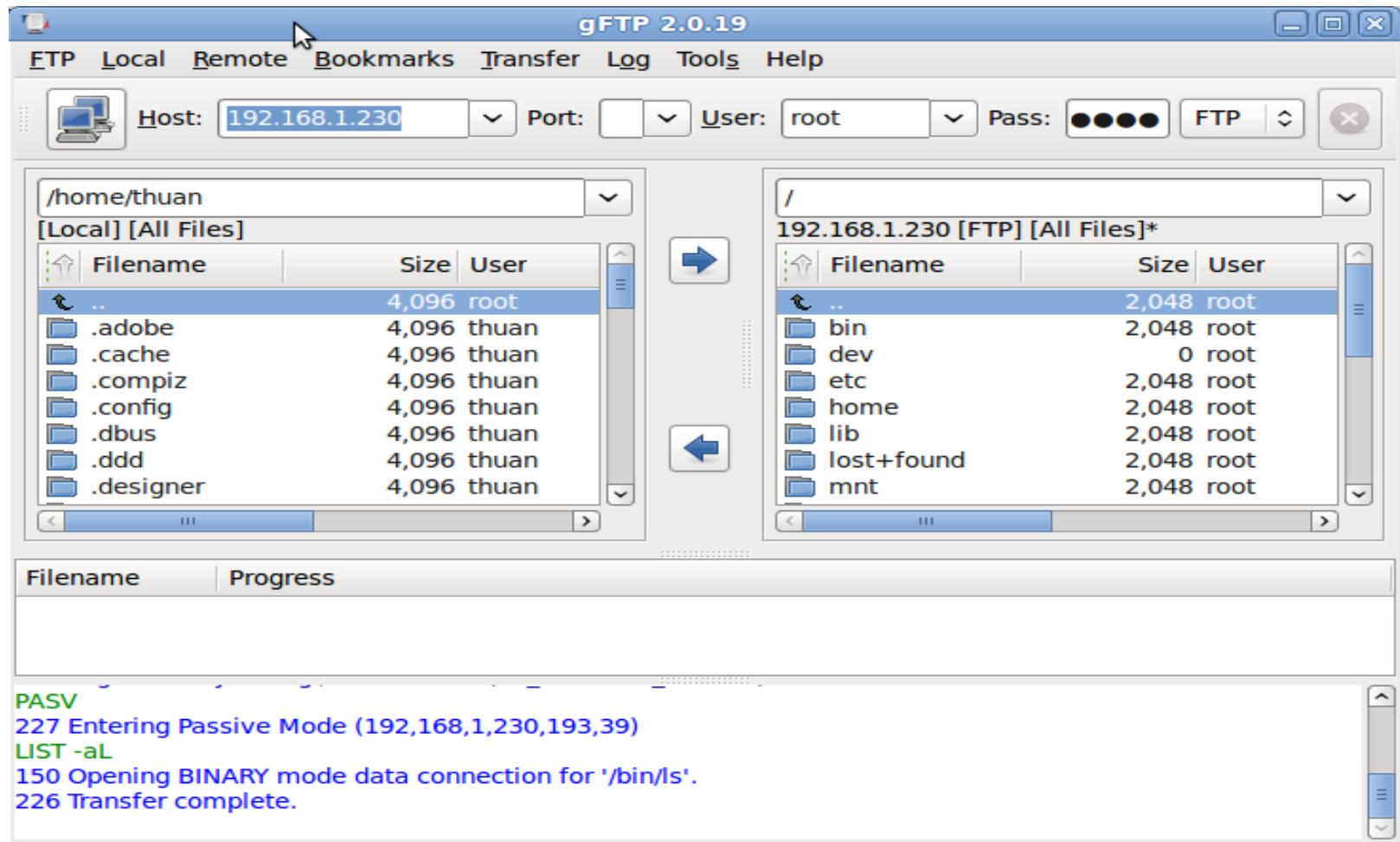
```
clear:
```

```
rm Hello
```

- Biên dịch chương trình: **make all**
- Xóa file sinh ra trước đó: **make clear**

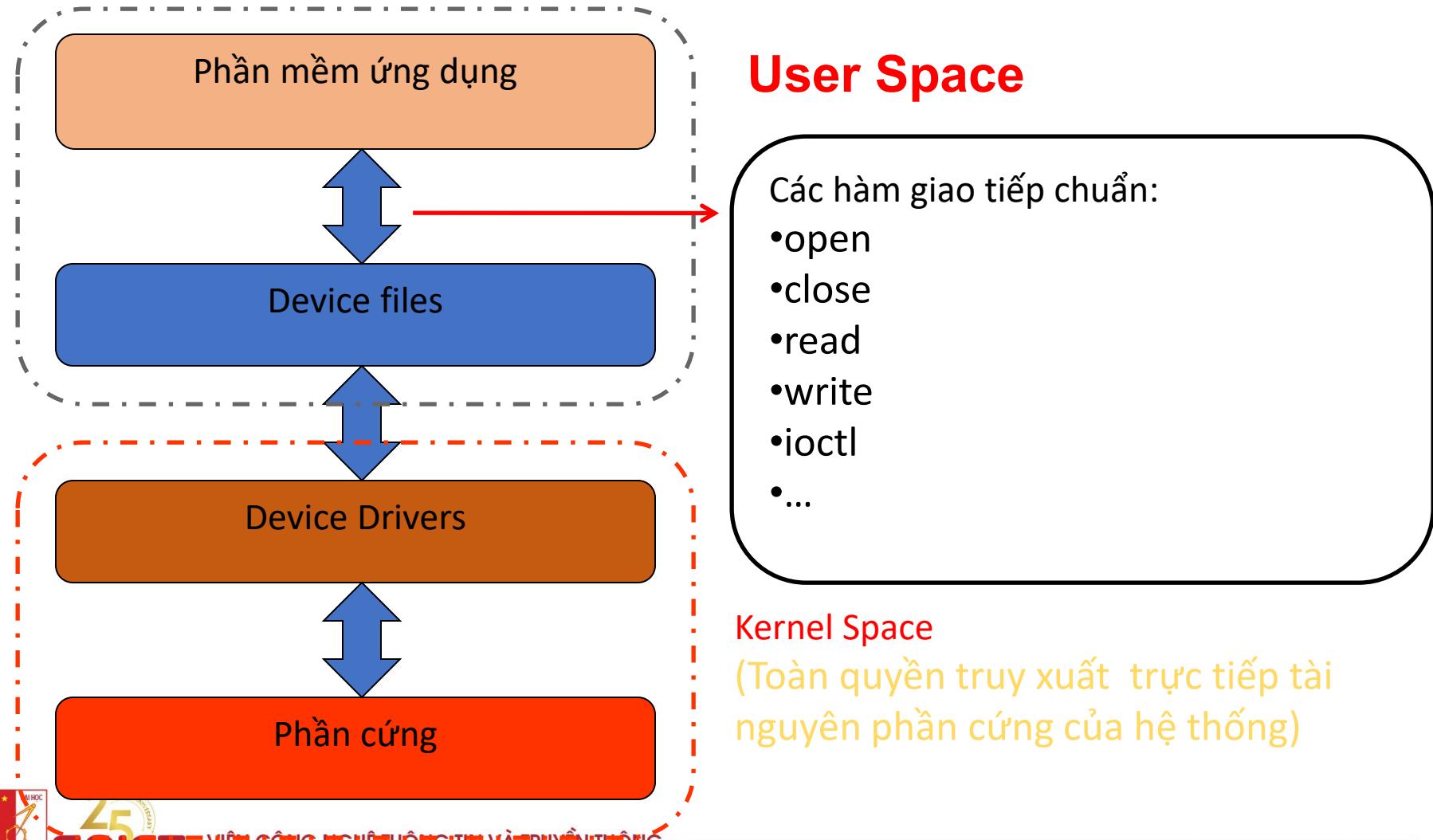
Đưa chương trình lên KIT

- Sử dụng phần mềm gFTP kết nối host và KIT



2.3. Cơ chế giao tiếp thiết bị

Mô hình giao tiếp ứng dụng – thiết bị trên hệ điều hành



Cơ chế giao tiếp thiết bị

- File thiết bị (**device file**): Hệ điều hành đối xử với các thiết bị ngoại vi phần cứng tương tự như các file dữ liệu thông thường.
- Quá trình ứng dụng giao tiếp với thiết bị là quá trình đọc/ghi file thiết bị đó thông qua driver.
- Linux quản lý các file thiết bị trong thư mục /dev (Sử dụng lệnh **ls -l /dev** để xem)
- Phân loại device files
 - **Character device**: thiết bị phần cứng đọc, ghi một chuỗi các byte dữ liệu
 - **Block device**: thiết bị phần cứng đọc, ghi một khối dữ liệu
(Đứng trên phương diện giao tiếp của driver)

Cơ chế giao tiếp thiết bị

- Để quản lý các file thiết bị, hệ điều hành sử dụng số hiệu định danh device number cho mỗi thiết bị. Số hiệu này gồm 2 giá trị:
 - Major device number: Xác định thiết bị này sử dụng driver nào
 - Minor device number: phân biệt giữa các thiết bị khác nhau sử dụng cùng một driver

Cơ chế giao tiếp thiết bị

- Kiểm tra danh sách các thiết bị
 - Các file thiết bị (device file) nằm trong thư mục /dev
 - Gõ lệnh **ls -al /dev**

```
brw-rw---- 1 root disk 3, 0 May 5 1998 /dev/hda
brw-rw---- 1 root disk 3, 1 May 5 1998 /dev/hda1
```

- Các thông tin:
 - Loại thiết bị: character (c) hay block (b) device
 - Tài khoản người dùng
 - Tên thiết bị
 - Major number và Minor number
 - Mount point

Cơ chế giao tiếp thiết bị

- Cơ chế lập trình giao tiếp:
 - Sử dụng cơ chế giao tiếp file và các hàm do driver cung cấp
 - Các hàm mở đóng file thiết bị: open, close
 - Hàm điều khiển cổng vào ra: ioctl
 - Hàm đọc/ghi file thiết bị: read, write
 - ...
 - Được qui định trong cấu trúc file operations của driver.

2.4. Lập trình giao tiếp vào ra cơ bản

Lập trình giao tiếp GPIO

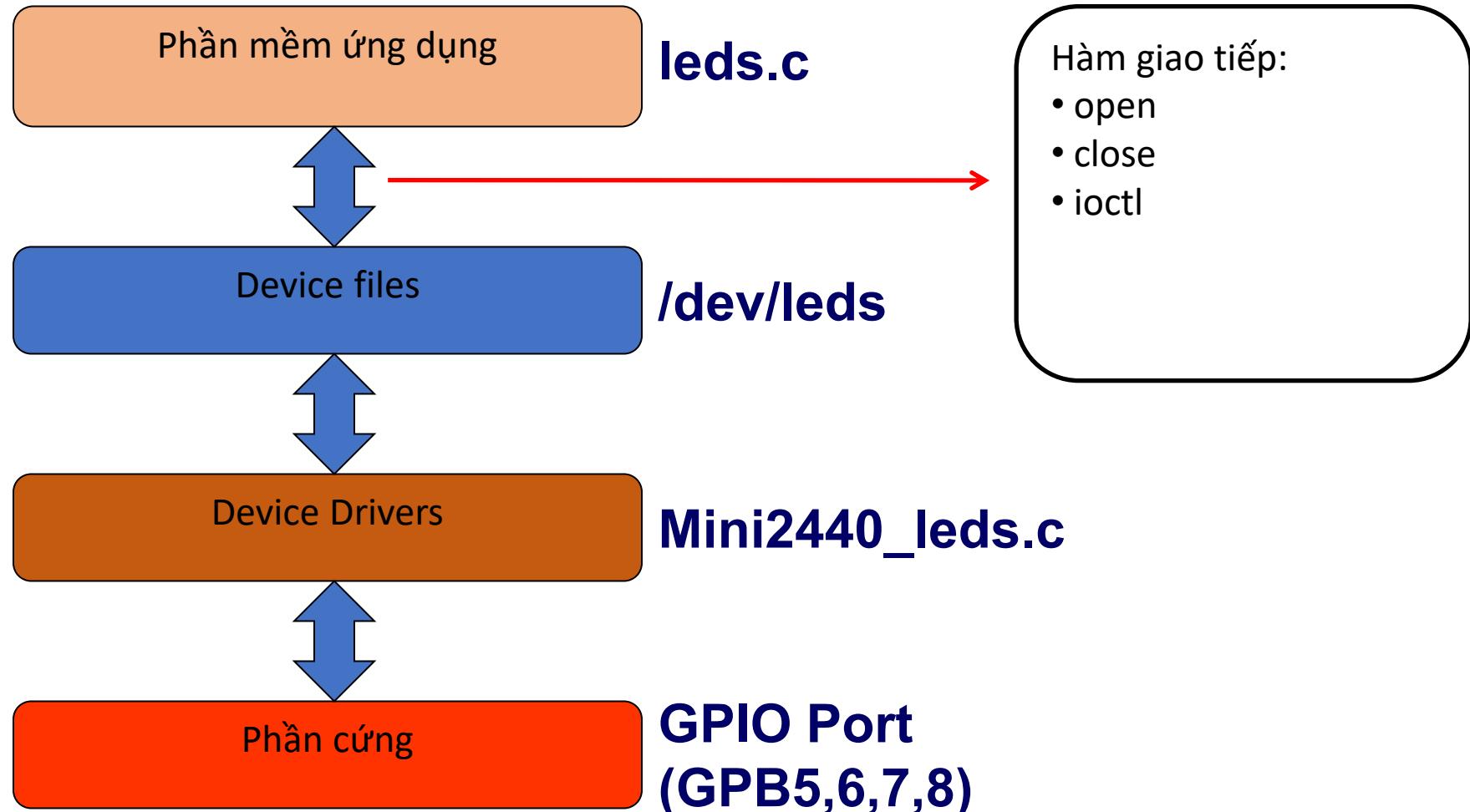
A. Lập trình giao tiếp LED

- Sử dụng led driver đã có
- 4 led đơn, ghép nối qua GPB5,6,7,8
- Điều khiển led on/off, tạo hiệu ứng: nhấp nháy, chạy đuôi, ...
- Cần sử dụng hàm trễ (delay): sleep, usleep (thư viện sys/time.h)

<https://sites.google.com/site/embedded247/escourse/gpioprogramming>



Mô hình giao tiếp điều khiển led



Lập trình điều khiển led

- **Mở file thiết bị: `fd=open("/dev/leds", 0)`**

- fd: file description (int, định danh file thiết bị)
- /dev/leds: device file name (tên file thiết bị)
- 0: WRITE_ONLY (chế độ mở để ghi)

- **Hàm điều khiển `ioctl(fd, onoff, led_no)`**

- ioctl: input output control (out 0, 1)
- Điều khiển bật/tắt led đơn có số hiệu led_no

- Mã nguồn driver cho led đơn:

[linux-2.6.32.2/drivers/char/mini2440_leds.c](#)

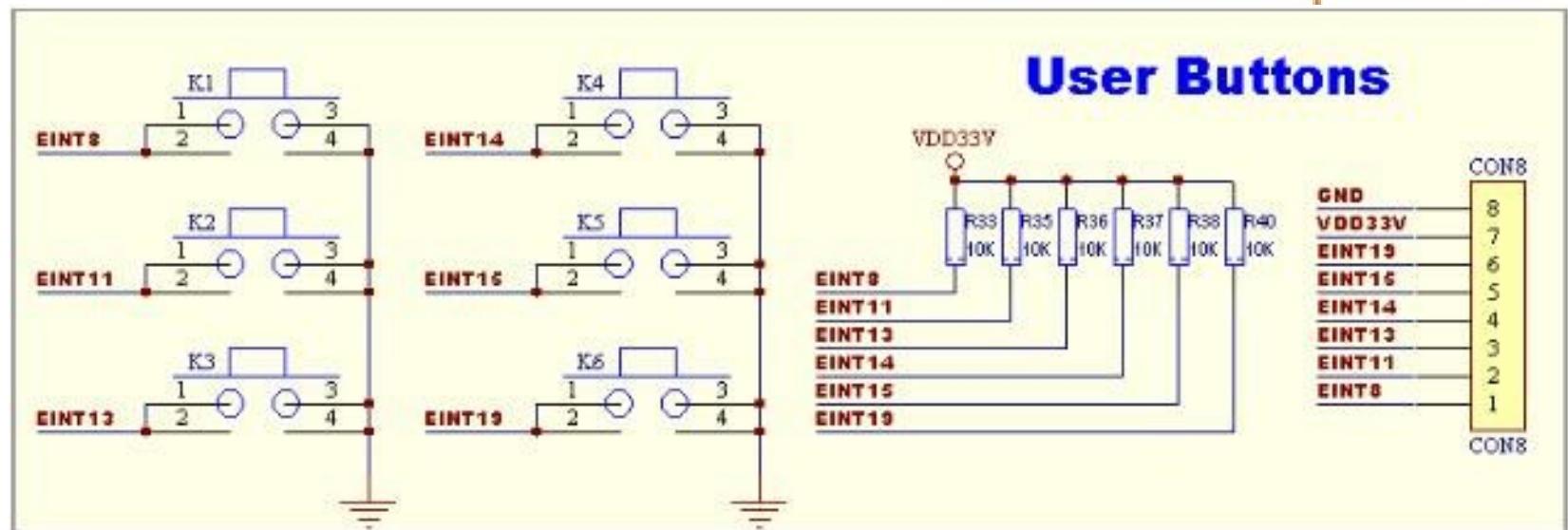
Mã nguồn minh họa

```
int main(int argc, char **argv) {
    int on;
    int led_no;  int fd;
    //Nhận tham số từ dòng lệnh, kiểm tra điều kiện tham số hợp lệ
    if (argc != 3 || sscanf(argv[1], "%d", &led_no) != 1 ||
        sscanf(argv[2], "%d", &on) != 1 || on < 0 || on > 1 || led_no < 0
        || led_no > 3) {
        fprintf(stderr, "Usage: leds led_no 0|1\n");
        exit(1);
    }
    //Mở file thiết bị leds
    fd = open("/dev/leds", 0);
    if (fd < 0) {
        perror("error open device leds\n");
        exit(1);
    }
    ioctl(fd, on, led_no); //Điều khiển bật/tắt led có số hiệu led_no
    close(fd); //Đóng file thiết bị
    return 0;
}
```

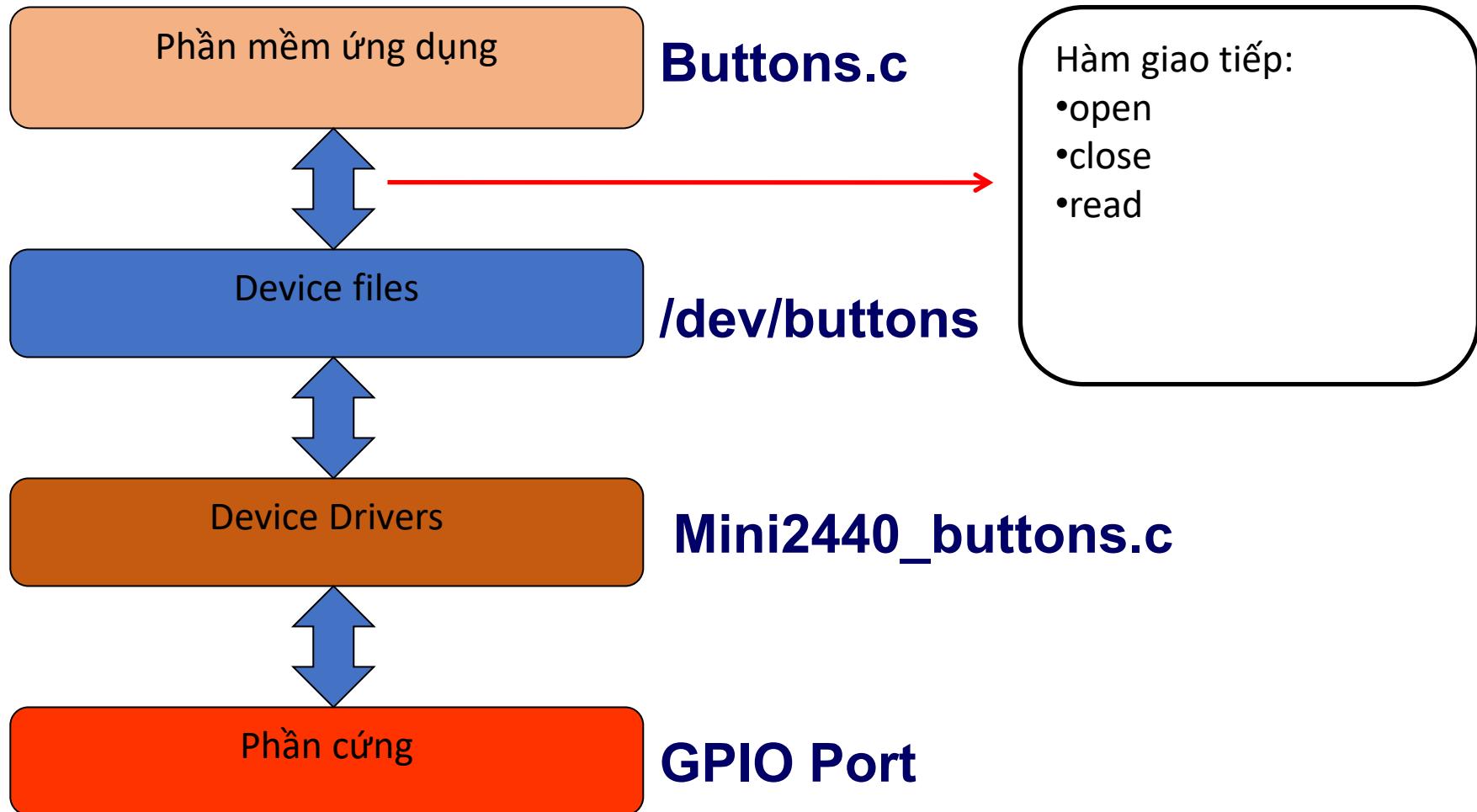
Minh họa giao tiếp vào ra cơ bản

B. Lập trình giao tiếp nút bấm

- Giao tiếp qua driver đã có
- Đọc trạng thái các nút bấm sử dụng phương pháp thăm dò (polling)



Mô hình giao tiếp nút bấm



Lập trình giao tiếp nút bấm

▪ Mở file thiết bị:

`buttons_fd=open("/dev/buttons", O_RDONLY)`

- buttons_fd: file description
- /dev/buttons: device file

▪ Hàm đọc dữ liệu

`read(buttons_fd, current_buttons,
 sizeof(current_buttons))`

- Đọc trạng thái các nút bấm

▪ Hàm đóng file thiết bị: `close(buttons_fd)`

▪ Mã nguồn driver cho nút bấm

`linux-2.6.32.2/drivers/char/mini2440_buttons.c`

Mã nguồn minh họa

```
#include ...
int main(void) {
    int buttons_fd;
    char buttons[6] = { '0', '0', '0', '0', '0', '0' };
    buttons_fd = open("/dev/buttons", O_RDONLY);
    if (buttons_fd < 0) {
        perror("open device buttons"); exit(1);
    }
    for (;;) {
        char current_buttons[6]; int count_of_changed_key; int i;
        if (read(buttons_fd, current_buttons, 6) != 6) {
            perror("read buttons:"); exit(1);
        }
        count_of_changed_key = 0;
        for (i = 0; i < 6; i++) {
            if (buttons[i] != current_buttons[i]) {
                buttons[i] = current_buttons[i];
                printf("key %d is %s", i + 1, buttons[i] == '0'?"up":"down");
                count_of_changed_key++;
            }
        }
        if (count_of_changed_key) printf("\n");
    }
    close(buttons_fd); return 0;
}
```

Giao tiếp GPIO sử dụng gpiolib

- 2 cách sử dụng giao tiếp gpio (từ Linux user space)
 - Cách 1: Viết gpio driver (trên không gian nhân hệ điều hành, kernel space), giao tiếp qua driver này.
(Ví dụ với led, button đã làm)
 - Cách 2: giao tiếp các chân gpio trực tiếp từ không gian người dùng (user space) dựa trên API thư viện gpiolib cung cấp. Linux cung cấp giao diện GPIO sysfs cho phép thao tác với bất kỳ chân GPIO từ userspace.

Giao tiếp GPIO sử dụng gpiolib

- Tất cả các giao diện điều khiển GPIO thông qua sysfs nằm trong thư mục /sys/class/gpio
- Kiểm tra bằng lệnh: ls /sys/class/gpio

```
File Edit View Terminal Help

Kernel 2.6.32.2-FriendlyARM on (/dev/pts/0)
FriendlyARM login: root tiep gpio:
Password:
[root@FriendlyARM /]# ls
bin          home      linuxrc    opt        sbin      usr
dev          kmtt      lost+found  proc      sys       var
etc          lib       mnt        root      tmp       www
[root@FriendlyARM /]# cd /sys/class/gpio
[root@FriendlyARM gpio]# ls
export      gpiochip128  gpiochip192  gpiochip32  gpiochip96
gpiochip0   gpiochip160  gpiochip224  gpiochip64  unexport
[root@FriendlyARM gpio]# echo 161 > /sys/class/gpio/export
[root@FriendlyARM gpio]# ls
export      gpiochip0   gpiochip160  gpiochip224  gpiochip64  unexport
gpio161    gpiochip128  gpiochip192  gpiochip32  gpiochip96
[root@FriendlyARM gpio]# echo 161 > /sys/class/gpio/unexport
[root@FriendlyARM gpio]# ls
export      gpiochip128  gpiochip192  gpiochip32  gpiochip96
gpiochip0   gpiochip160  gpiochip224  gpiochip64  unexport
[root@FriendlyARM gpio]#
```

PIO sysfs interface cần cấu hình nhân hệ điều hành cho phép sử dụng giao
cấu hình trước khi biên dịch nhân (nếu hệ điều hành cài đặt chưa có)

Giao tiếp GPIO sử dụng gpiolib

- Giao diện này cung cấp các files điều khiển sau đây:

export Make a specific GPIO pin available for userspace control. Write the pin number N (e.g. "55", ASCII); the gpioN directory should appear.

gpioN/direction Write "in" or "out" to set pin direction. Write "high" or "low" to set direction to output, with initial value, atomically.

gpioN/value Read the current pin status in input. For output, write "0" or "1" to set the pin status. To get change notification (interrupt) lseek() to end of file, and either poll() for POLLPRI and POLLERR, or select() with the file descriptor in exceptfds.

Giao tiếp GPIO sử dụng gpiolib

- Minh họa giao tiếp gpiolib sử dụng linux command

Cấu hình chân GPF5 (micro2440) output, và xuất giá trị 0 ra chân này

echo 165 > /sys/class/gpio/export

echo "out" > /sys/class/gpio/gpio165/direction

echo 0 > /sys/class/gpio/gpio165/value

Chi tiết:

<https://sites.google.com/site/embedded247/ddcourse/giao-tiep-gpio-tu-userspace-1>

Minh họa lập trình giao tiếp gpiolib

■ File gpio.h

```
#ifndef GPIO_H
#define GPIO_H
int gpio_export(unsigned int gpio); //export pin to user space
int gpio_unexport(unsigned int gpio); //unexport pin from user
space
int gpio_dir_out(unsigned int gpio); //Config pin as output
int gpio_dir_in(unsigned int gpio); //Config pin as input
int gpio_set_value(unsigned int gpio, unsigned int value); //Ghi
gia tri cua mot pin gpio output
int gpio_get_value(unsigned int gpio, unsigned int *value); //Doc
gia tri cua mot pin gpio input
int gpio_set_edge(unsigned int gpio, char *edge);
#endif
```

Minh họa lập trình giao tiếp gpiolib

- File gpio.c (ví dụ hàm đăng ký chân gpio sử dụng)

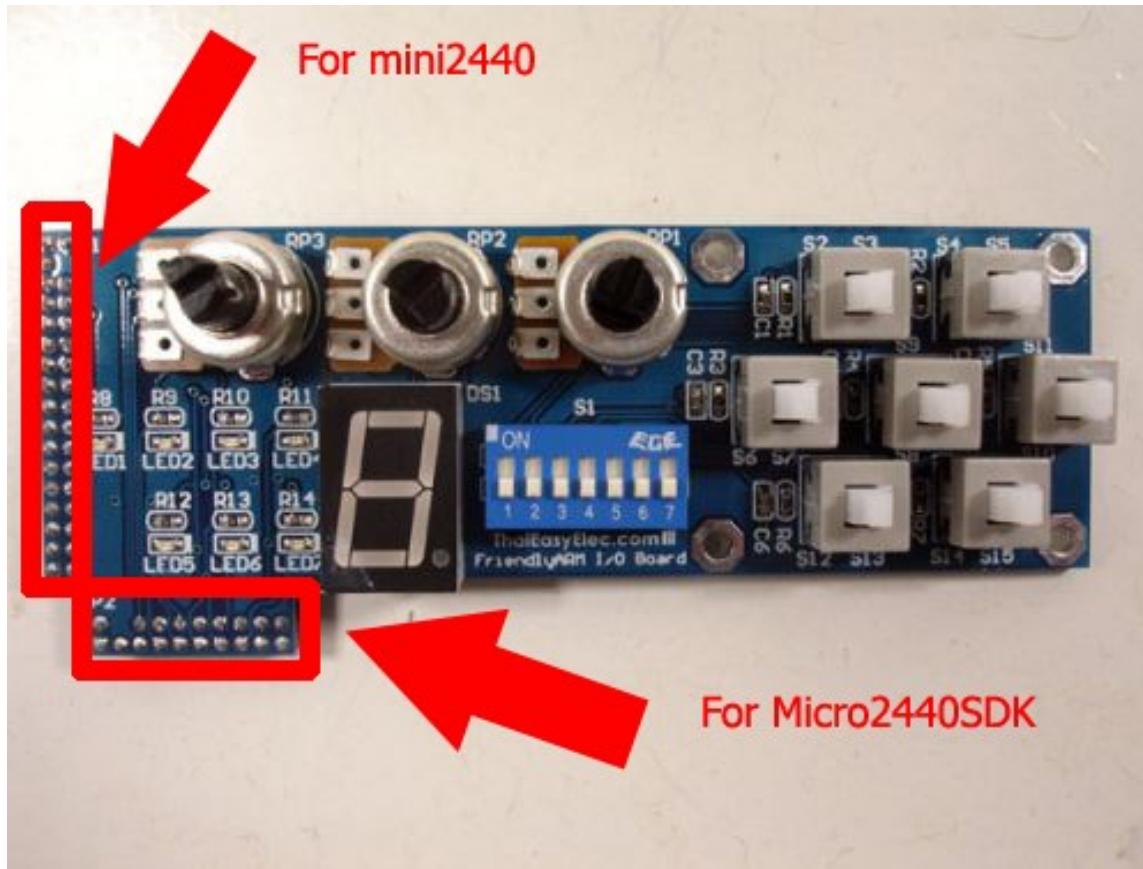
```
int gpio_export(unsigned int gpio) {
    int fd, len;
    char buf[64];
    fd = open("/sys/class/gpio/export", O_WRONLY);
    if (fd < 0) {
        perror("gpio/export");
        return fd;
    }
    len = snprintf(buf, sizeof(buf), "%d", gpio);
    write(fd, buf, len);
    //Ghi gpio <number> vào file export
    close(fd);
    return 0;
}
```

Hàm ghi dữ liệu ra chân gpio

```
int gpio_set_value(unsigned gpio, unsigned value)
{
    int fd, len;
    char buf[64];
    len = snprintf(buf, sizeof(buf),
    "/sys/class/gpio/gpio%d/value", gpio);
    fd = open(buf, O_WRONLY);
    if (fd < 0) {
        perror("gpio/value");
        return fd;
    }
    //Output value 1 or 0 by write "1" or "0" into file value
    if (value)
        write(fd, "1", 2);
    else
        write(fd, "0", 2);
    close(fd);
    return 0;
}
```

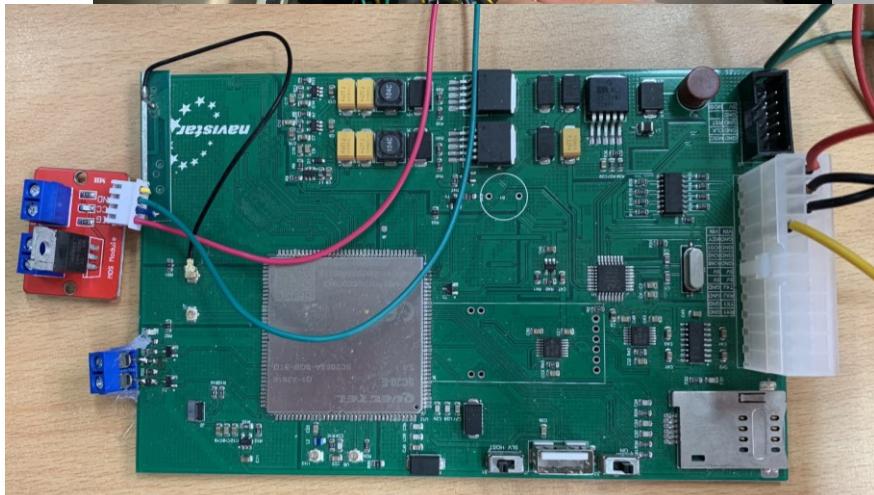
Minh họa lập trình giao tiếp gpiolib

- Demo: Ứng dụng giao tiếp Board mở rộng, điều khiển led đơn, 7 seg, buttons.



Giao tiếp GPIO trên Android

- Ví dụ: điều khiển khóa cửa on/off
- Cơ chế ghi file thiết bị (gpio pins)



Điều khiển khóa cửa (on/off)

```
public int openDoor(String openType) {  
    String gpioNo = "";  
    if(openType.equals("IN")) {  
        gpioNo = "gpio947";  
    }  
    else if(openType.equals("OUT")) {  
        gpioNo = "gpio1000";  
    }  
    try {  
        FileOutputStream gpioFile = new FileOutputStream(new  
File("/sys/class/gpio/" + gpioNo + "/value"));  
        gpioFile.write(new byte[] {49}); //out 1  
        Thread.sleep(500);  
        gpioFile.write(new byte[] {48}); //out 0  
        Thread.sleep(3000);  
        gpioFile.write(new byte[] {49}); //out 1  
        gpioFile.flush();  
        gpioFile.close();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    return 0;  
}
```

Lập trình Java trên Android

Điều khiển buzzer phát âm thanh

```
public void playBuzzer(int delay, int repeat) {
    try {
        FileOutputStream gpioFile = new FileOutputStream(new
File("/sys/class/gpio/gpio934/value"));
        for(int i = 0; i < repeat; i++) {
            try {
                gpioFile.write(new byte[] { 49 });
                Thread.sleep(delay);
                gpioFile.write(new byte[] { 48 });
                Thread.sleep(delay);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        gpioFile.flush();
        gpioFile.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

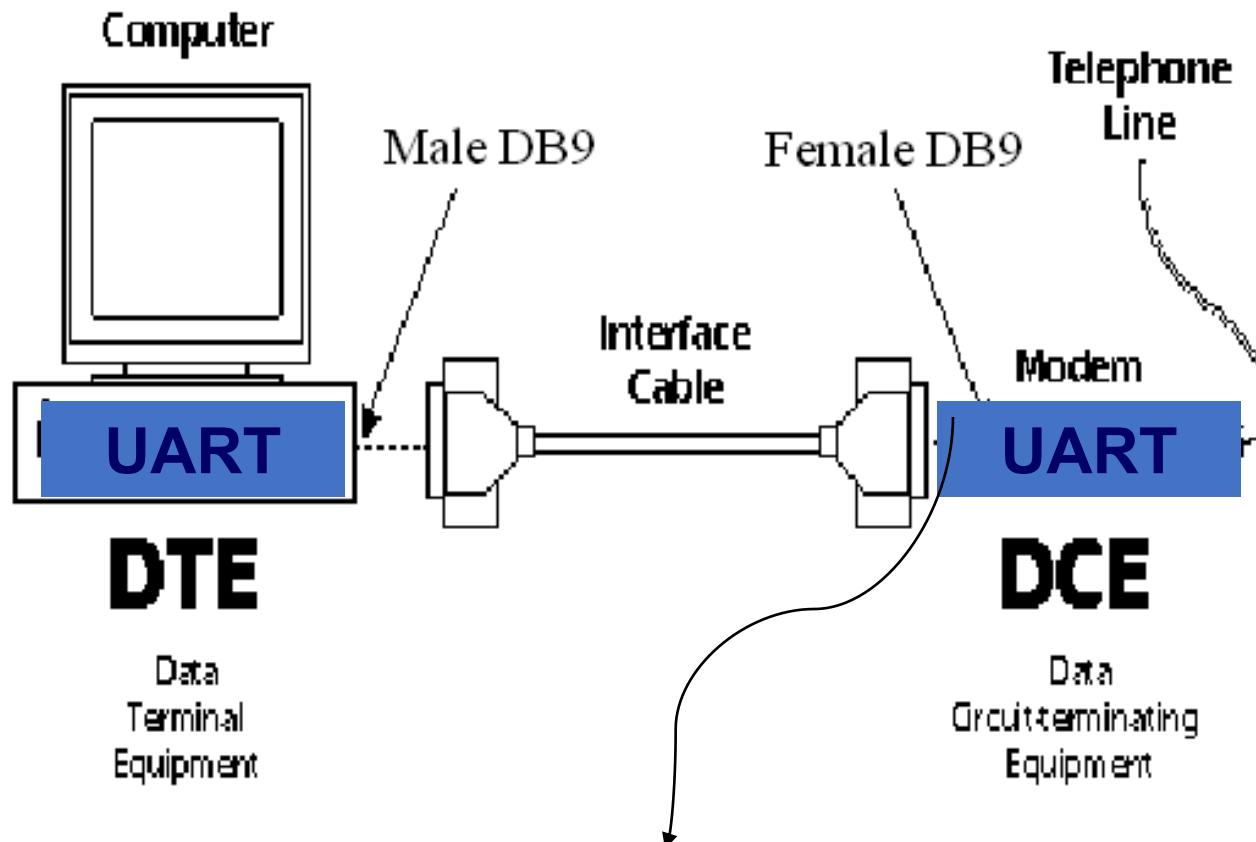
2.5. Lập trình giao tiếp RS232

Giới thiệu chuẩn RS232

Lập trình giao tiếp RS232 trên Linux

Chuẩn RS232

- Chuẩn đầu nối trên PC

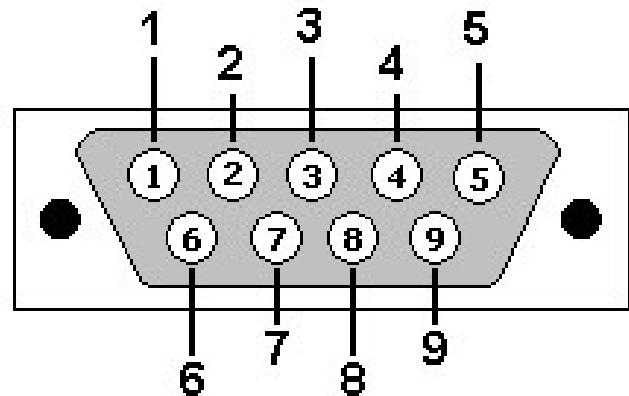


UART (Universal Asynchronous receiver/transmitter)

Chuẩn RS232

- Chuẩn đầu nối trên PC

- Chân 1 (DCD-Data Carrier Detect): phát hiện tín hiệu mang dữ liệu
- Chân 2 (RxD-Receive Data): nhận dữ liệu
- Chân 3 (TxD-Transmit Data): truyền dữ liệu
- Chân 4 (DTR-Data Terminal Ready): đầu cuối dữ liệu sẵn sàng
- Chân 5 (Signal Ground): đất của tín hiệu
- Chân 6 (DSR-Data Set Ready): dữ liệu sẵn sàng
- Chân 7 (RTS-Request To Send): yêu cầu gửi
- Chân 8 (CTS-Clear To Send): Xóa để gửi
- Chân 9 (RI-Ring Indicate): báo chuông



Chuẩn RS232

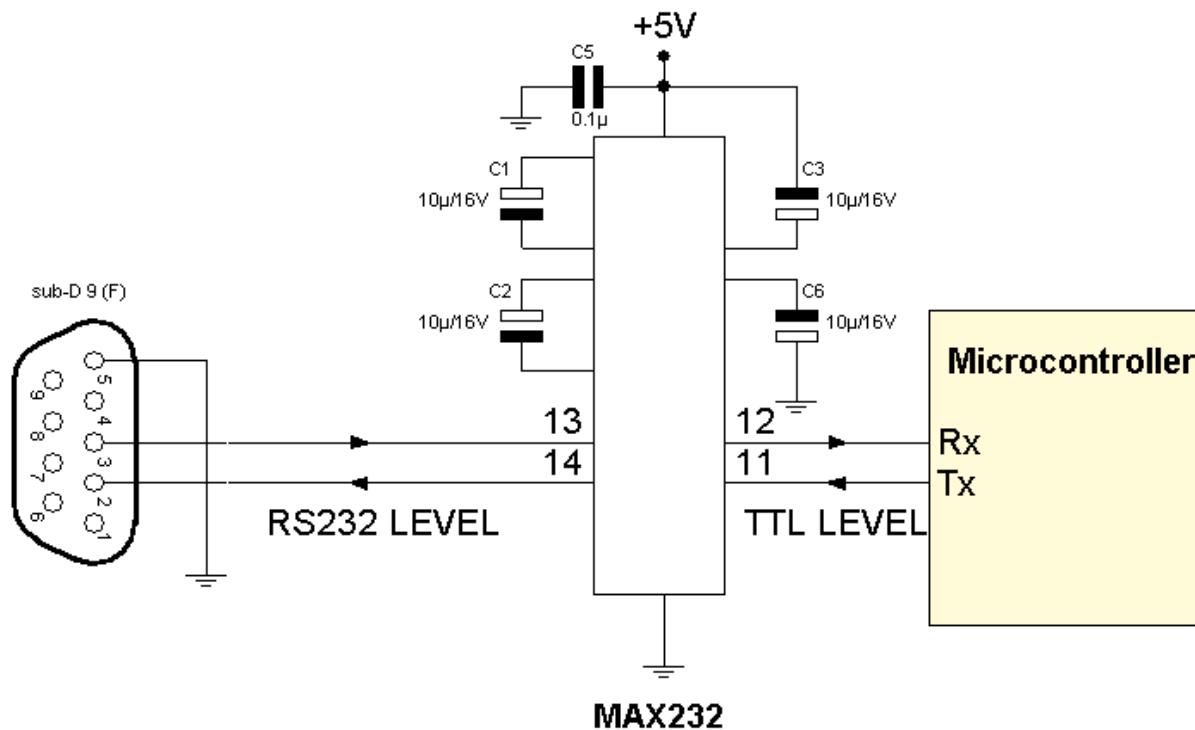
- **Khuôn dạng khung truyền**

- PC truyền nhận dữ liệu qua cổng nối tiếp RS-232 thực hiện theo kiểu không đồng bộ (Asynchronous)
- Khung truyền gồm 4 thành phần
 - 1 Start bit (Mức logic 0): bắt đầu một gói tin, đồng bộ xung nhịp clock giữa DTE và DCE
 - Data (5,6,7,8 bit): dữ liệu cần truyền
 - 1 parity bit (chẵn (even), lẻ (odd), mark, space): bit cho phép kiểm tra lỗi
 - Stop bit (1 hoặc 2 bit): kết thúc một gói tin



Chuẩn RS232

- Kịch bản truyền



**Ghép nối không bắt tay giữa hai thiết bị
(Khác nhau về mức điện áp)**

Lập trình giao tiếp chuẩn RS232 trên Linux

- **Các bước cơ bản gồm:**

- **Khởi tạo:** Khai báo thư viện
- **Bước 1:** Mở cổng COM
- **Bước 2:** Thiết lập tham số cấu hình cổng COM
- **Bước 3:** Đọc/ghi cổng (gửi/nhận dữ liệu)
- **Bước 4:** Đóng cổng

Khai báo thư viện

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h> // UNIX standard function
#include <fcntl.h> // File control definitions
#include <errno.h> // Error number definitions
#include <termios.h> // POSIX terminal control
#include <time.h> // time calls
```

Bước 1: Mở cổng

- Sử dụng lệnh mở file

int fd = open ("/dev/ttySAC0", O_RDWR);

- “/dev/ttySAC0”: Tên file thiết bị cổng COM muốn mở
- $fd > 0$ nếu mở file thành công
- $fd < 0$ nếu mở file thất bại

Bước 2: Thiết lập tham số

- Sử dụng cấu trúc termios

struct termios port_settings;

- Thiết lập tham số (9600, 8, n, 1)

```
cfsetispeed(&port_settings, B9600);
cfsetospeed(&port_settings, B9600);
port_settings.c_cflag &= ~PARENB;
port_settings.c_cflag &= ~CSTOPB;
port_settings.c_cflag &= ~CSIZE;
port_settings.c_cflag |= CS8;
tcsetattr(fd, TCSANOW, &port_settings);
```

Bước 3: Đọc, ghi cổng

- Đọc cổng: sử dụng lệnh đọc file

```
n=read(fd,&result,sizeof(result));
```

➤ N: số ký tự đọc được

➤ Result: chứa kết quả

- Ghi cổng: sử dụng lệnh ghi file

```
n=write(fd,"Hello World\r",12);
```

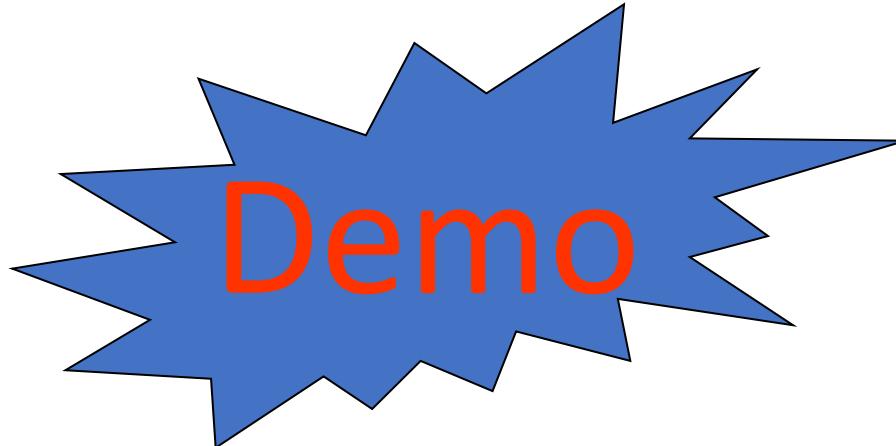
➤ N:số ký tự đã ghi

➤ fd: file id (có được từ thao tác mở file thành công)

Bước 4: Đóng cổng

- Đóng cổng: sử dụng lệnh đóng file
`close (fd);`

fd: file ID (có được từ thao tác mở file thành công)



<https://sites.google.com/site/embedded247/escourse/rs232programming>

Ví dụ giao tiếp serial port trên Android

- Thiết bị Android gửi lệnh điều khiển cho mạch điều khiển AT mega2560 qua cổng serial (RS232)



Thiết bị Android

Mạch điều khiển
ATMega2560

Ví dụ giao tiếp serial port trên Android (Java)

- Giao tiếp qua driver thiết bị, có device file name: /dev/ttyHSL1
- Sử dụng các hàm read, write file của thư viện
- Mở cổng

```
File mSerialPort;
FileInputStream mSerR;
OutputStream mSerW;
try {
    mSerialPort = new File("/dev/ttyHSL1");
    mSerR = new FileInputStream(mSerialPort);
    mSerW = new FileOutputStream(mSerialPort);
}
catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

Ví dụ giao tiếp serial port trên Android (Java)

- Ghi một gói dữ liệu command ra cổng com

```
public void sendCommandToLocker (char cmdType, int portno) {  
    byte[] data = new byte[5];  
    data[0] = 0x01; //start byte  
    data[1] = (byte) cmdType;  
    data[2] = (byte) (portno + 0x40);  
    data[3] = 0x00;  
    data[4] = 0x02; //end byte  
    try {  
        if (mSerW != null) mSerW.write(data);  
    }  
    catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

Ví dụ giao tiếp serial port trên Android (Java)

■ Đọc dữ liệu từ cổng com

```
byte[] recvdata = new byte[5];
int index = 0;
while (mSerR.available() > 0) {
    byte tmp = (byte) mSerR.read();
    if (index < 5) {
        recvdata[index++] = tmp;
    } else {
        for (int i = 0; i < 4; i++) {
            recvdata[i] = recvdata[i + 1];
        }
        recvdata[4] = tmp;
    }
}
```

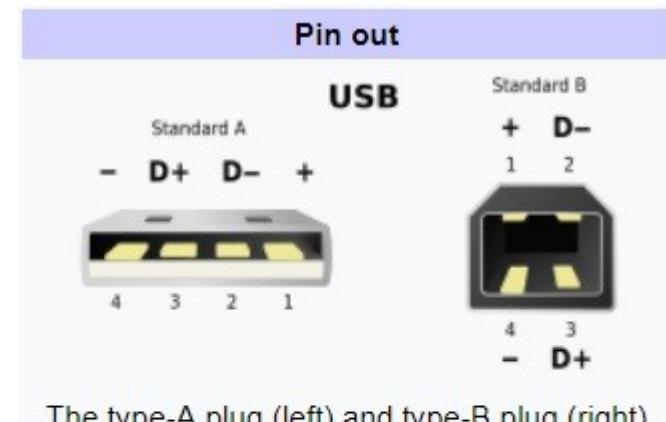
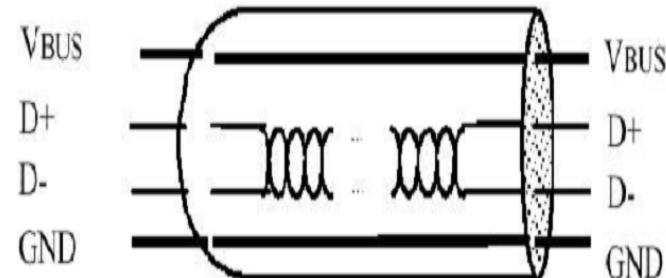
2.6. Giao tiếp thiết bị chuẩn USB

- Tìm hiểu giao tiếp USB
- Lập trình giao tiếp thiết bị usb

Tìm hiểu giao tiếp USB

Giới thiệu chuẩn USB:

- Năm 1995: USB 1.0
 - Tốc độ Low-Speed: 1.5 Mbps
 - Tốc độ tối đa (Full-Speed): 12 Mbps
- Năm 1998: USB 1.1 (Sửa lỗi của USB 1.0)
 - Tốc độ tối đa (Full-Speed): 12 Mbps
- Năm 2001: USB 2.0
 - Tốc độ tối đa (High-Speed): 480 Mbps
- Năm 2008: USB 3.0
 - Tốc độ tối đa (Super-Speed): 4.8 Gbps
- Tín hiệu
 - Truyền kiểu nối tiếp
 - Tín hiệu trên hai đường D+ và D- là tín hiệu vi sai

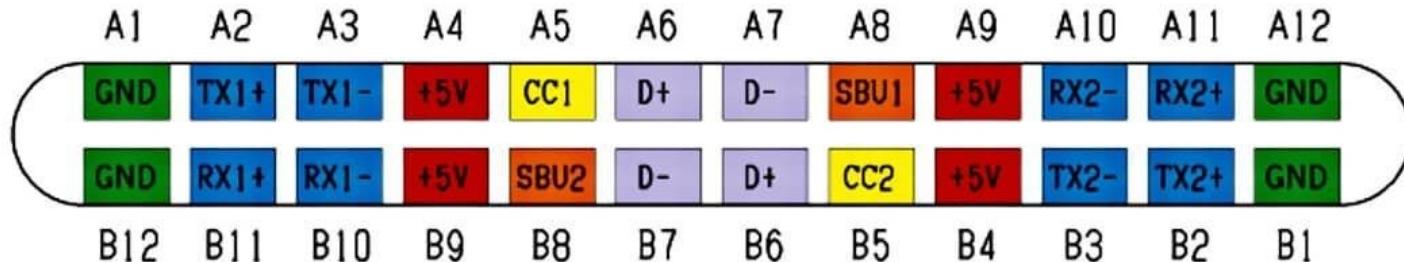


The type-A plug (left) and type-B plug (right)

| | |
|-------|-------------------------|
| Pin 1 | V _{BUS} (+5 V) |
| Pin 2 | Data- |
| Pin 3 | Data+ |
| Pin 4 | Ground |

Tìm hiểu giao tiếp USB

USB Type-C Connector Pin Assign

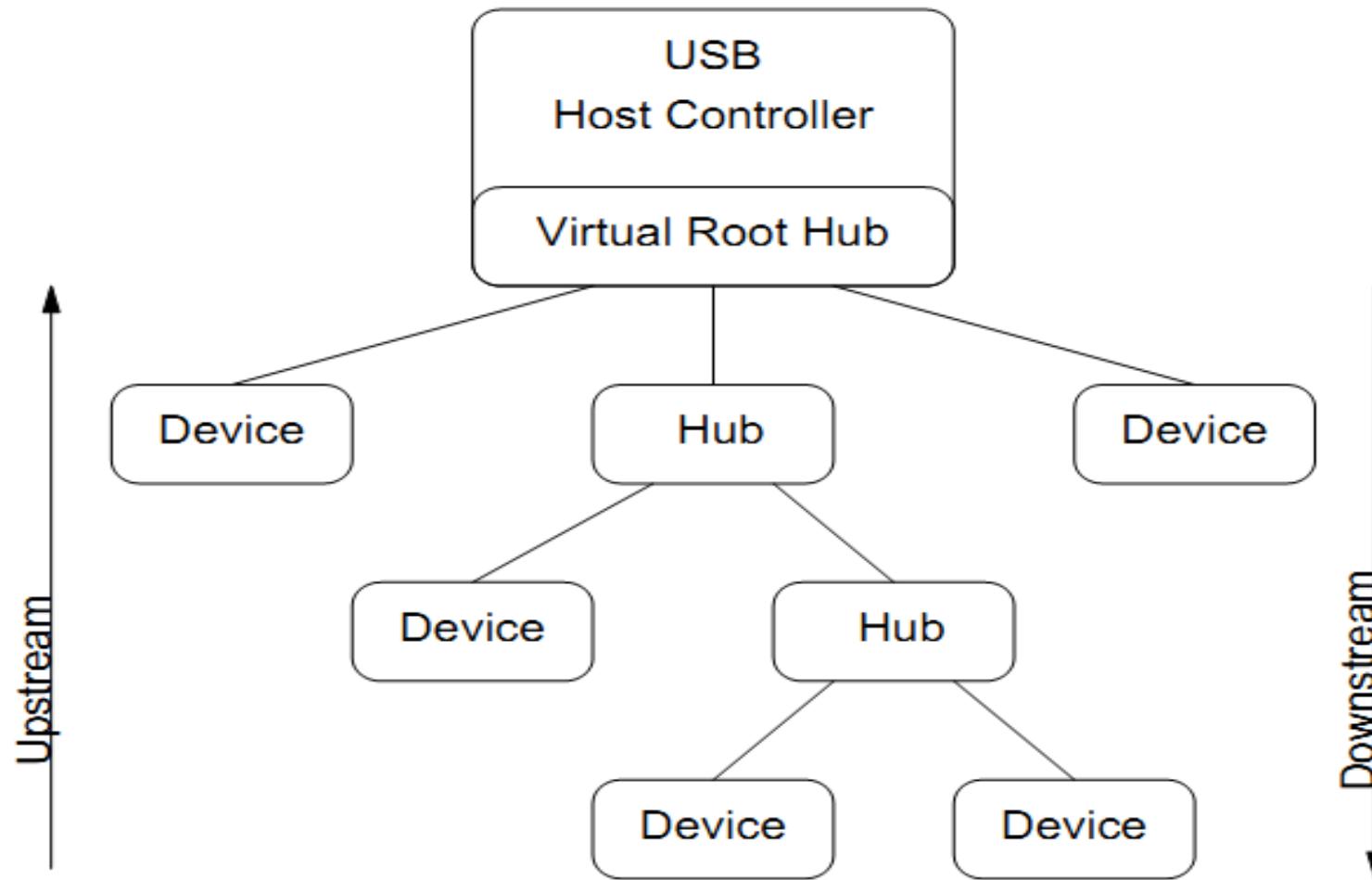


█ USB3.1 Super speed+ 10Gbps █ Secondary Bus
█ USB2.0 High speed 480Mbps █ USB Power Delivery Communication

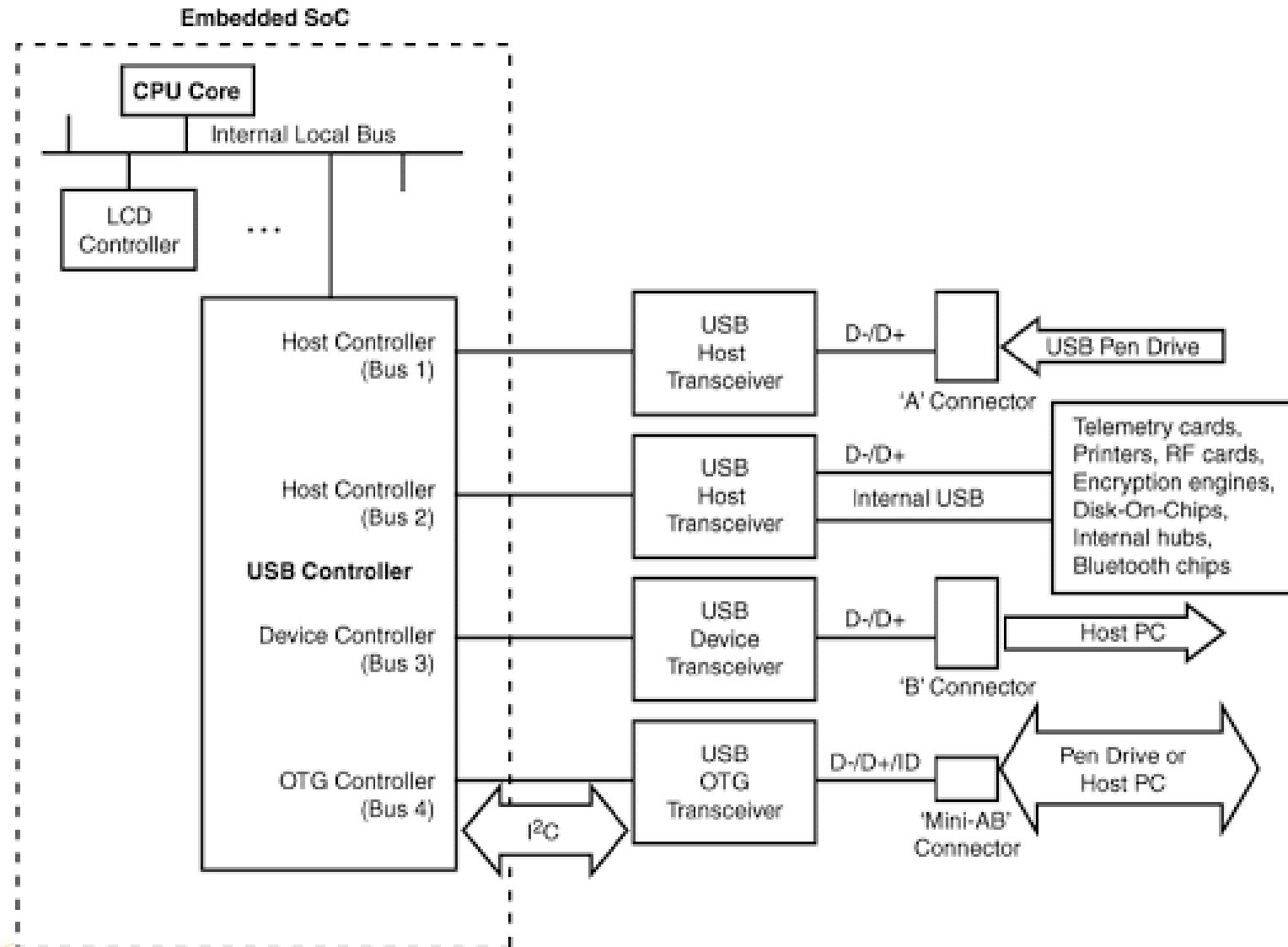


USB Type C

Mô hình bus USB



Mô hình kiến trúc USB



Vai trò của các thành phần

- Vai trò của USB host:

- Trao đổi dữ liệu với các thiết bị ngoại vi
- Điều khiển bus:
 - Quản lý được các thiết bị kết nối vào đường bus và khả năng của mỗi thiết bị đó: sử dụng cơ chế điểm danh (Enumeration)
 - Phân xử, quản lý luồng dữ liệu trên bus, đảm bảo các thiết bị đều có cơ hội trao đổi dữ liệu
- Kiểm tra lỗi: thêm các mã kiểm tra lỗi vào gói tin cho phép phát hiện lỗi và yêu cầu truyền lại gói tin
- Cung cấp nguồn điện cho tất cả các thiết bị

Vai trò của các thành phần

- Vai trò của thiết bị ngoại vi
 - Trao đổi dữ liệu với host
 - Phát hiện các gói tin hay yêu cầu (request) được gửi tới thiết bị để xử lý phù hợp
 - Kiểm tra lỗi: tương tự như Host, các thiết bị ngoại vi cũng phải chèn thêm các bit kiểm tra lỗi vào gói tin gửi đi
 - Quản lý nguồn điện: các thiết bị có thể sử dụng nguồn điện ngoài hay nguồn từ bus. Nếu sử dụng nguồn từ bus, thường có chế độ tiết kiệm điện năng.

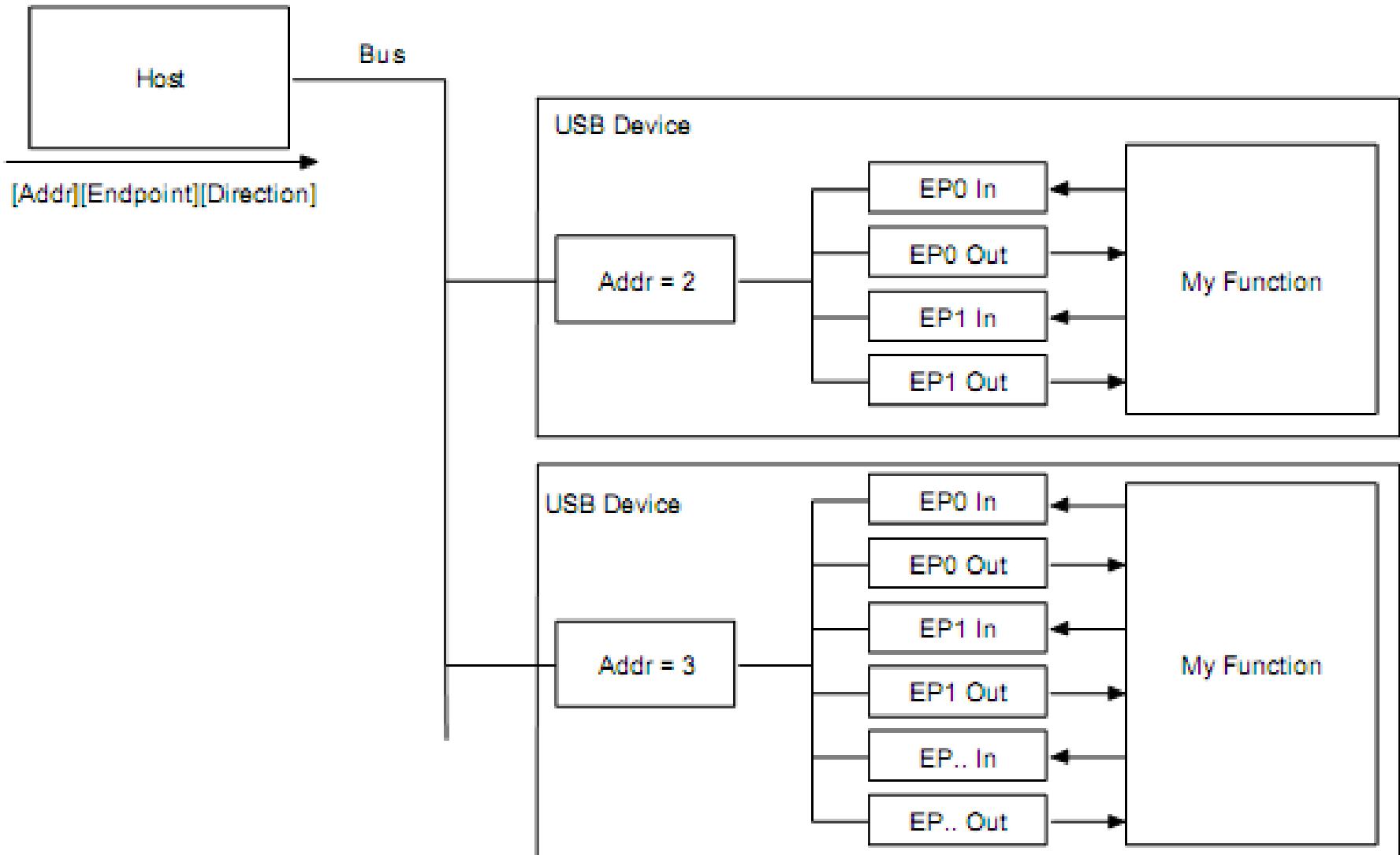
Endpoint & pipes

- Mỗi quá trình truyền nhận dữ liệu bao gồm một hay nhiều giao dịch (transactions), mỗi giao dịch gồm một hay nhiều packets
- Để hiểu được các giao dịch, các packet và nội dung của chúng -> cần tìm hiểu hai khái niệm Endpoint và Pipes

Endpoint

- Endpoint của thiết bị:
 - Là “điểm cuối” trong kênh giao tiếp giữa host và usb device.
 - Endpoint là bộ đệm (gửi, nhận) nằm trên thiết bị.
 - Các Endpoint được đánh địa chỉ và xác định hướng
 - In Endpoint: bộ đệm gửi (host lấy dữ liệu về)
 - Out Endpoint: bộ đệm nhận (host gửi dữ liệu vào)
 - Tất cả các thiết bị đều phải có Endpoint 0, đây là endpoint mặc định để gửi các thông tin điều khiển

Endpoint



Pipes

- Pipes: kết nối Endpoint của thiết bị tới Host
 - Phải thiết lập pipe trước khi muốn trao đổi dữ liệu
 - Host thiết lập pipe trong quá trình điểm danh (Enumeration)
 - Các Pipe sẽ được hủy khi thiết bị ngắt kết nối khỏi bus
 - Tất cả các thiết bị đều có một đường ống điều khiển (control pipe) mặc định sử dụng Endpoint 0

Ví dụ lsusb -l

```
pi@raspberrypi: ~
MaxPower          100mA
Interface Descriptor:
  bLength           9
  bDescriptorType   4
  bInterfaceNumber  0
  bAlternateSetting 0
  bNumEndpoints     1
  bInterfaceClass   9 Hub
  bInterfaceSubClass 0
  bInterfaceProtocol 0 Full speed (or root) hub
  iInterface        0
Endpoint Descriptor:
  bLength           7
  bDescriptorType   5
  bEndpointAddress 0x81 EP 1 IN
  bmAttributes      3
    Transfer Type    Interrupt
    Synch Type       None
    Usage Type        Data
  wMaxPacketSize    0x0001 1x 1 bytes
```

Device Classes

- Các thiết bị ngoại vi chuẩn (chuột, máy in, ổ nhớ flash...) có đặc tính truyền nhận dữ liệu chung -> Hệ điều hành có thể cung cấp driver chung cho các nhóm, các nhà sản xuất thiết bị không cần viết driver riêng.
- Các thiết bị đặc thù riêng: cần có driver từng loại
- Các nhóm thiết bị đã được định nghĩa
 - Audio
 - Communication devices
 - **Human interface (HID)**
 - IrDA Bridge
 - **Mass Storage**
 - Cameras and scanners
 - Video

Quá trình trao đổi dữ liệu

- Các thiết bị USB có thể trao đổi dữ liệu với Host theo 4 kiểu hoàn toàn khác nhau, cụ thể:
 - Truyền điều khiển (control transfer)
 - Truyền ngắt (interrupt transfer)
 - Truyền theo khối (bulk transfer)
 - Truyền đẳng thời (isochronous transfer)

Các kiểu truyền

- **Truyền điều khiển (control transfer):**
 - Truyền các yêu cầu điều khiển thiết bị (control packet)
 - Mức độ ưu tiên cao, khả năng kiểm soát lỗi tự động, tốc độ cao
- **Truyền ngắn (interrupt transfer):**
 - Các thiết bị cung cấp một lượng dữ liệu nhỏ, lặp lại, ví dụ như chuột, bàn phím
 - USB Host sẽ hỏi theo chu kỳ, ví dụ 10ms một lần xem có các dữ liệu mới gửi đến

Các kiểu truyền

- **Truyền theo khối (bulk transfer):**
 - Khi có lượng dữ liệu lớn cần truyền và cần kiểm soát lỗi truyền nhúng không có yêu cầu nghiêm ngặt về thời gian truyền
 - Ví dụ: máy in, máy quét
- **Truyền đẳng thời (isochronous transfer):**
 - Khi có khối lượng dữ liệu lớn với tốc độ dữ liệu đã được qui định
 - Ví dụ: thiết bị âm thanh, video record
 - Một giá trị tốc độ xác định được duy trì.
 - Việc hiệu chỉnh lỗi không được thực hiện vì lỗi truyền lẻ tẻ không gây ảnh hưởng đáng kể.

Ví dụ 1 giao tiếp thiết bị usb RFID, QR code reader

- Giao tiếp thiết bị usb RFID reader, QR code reader: là các thiết bị vào chuẩn trên nền tảng hệ điều hành (Windows, Android, Linux, ...)



Ví dụ giao tiếp thiết bị usb RFID, QR code reader

```
//Xử lý sự kiện trên Android OS
@Override
public boolean dispatchKeyEvent(KeyEvent event) {
    String readerID = event.getDevice().getDescriptor();
    msQRCodeID = readerID;
    if(event.getAction() == KeyEvent.ACTION_UP)
    {
        //Log.d("KeyEvent:", event.getKeyCode() + " " + event.getUnicodeChar());
        byte scancode = (byte) event.getUnicodeChar();
        if(scancode != 13 && scancode != 10 && scancode != 0 && numQRCode <
MAX_QRCODE) {
            bQRCode[numQRCode] = scancode;
            numQRCode++;
        }
        if(scancode == 10 || scancode == 13) {
            String sQRCode = "";
            for(int i = 0; i < numQRCode; i++) {
                sQRCode = sQRCode + String.format("%c", bQRCode[i]);
            }
            numQRCode = 0;
            msQRCodeValue = sQRCode;
            Log.d("VINID_TAG: ", "Android ID = " + msDeviceID + " Reader ID = " +
msQRCodeID + " " + " Code = " + msQRCodeValue);
            new AuthenticateTask().execute(msDeviceID, msQRCodeID, sQRCode);
        }
    }
    return super.dispatchKeyEvent(event);
}
```

Ví dụ 2: Giao tiếp thiết bị usb joystick

- Là thiết bị vào chuẩn (HID)



Cấu trúc js_event trên Linux

- Linux định nghĩa cấu trúc js_event để lưu các thông tin khi có phát sinh sự kiện (khởi tạo thiết bị, người dùng bấm nút chức năng, nút chỉnh hướng)
- Định nghĩa trong **include/linux/joystick.h**

```
struct js_event {  
    unsigned int time; /* event timestamp in milliseconds */  
    short value; /* value */  
    unsigned char type; /* event type */  
    unsigned char number; /* axis/button number */  
};
```

Cấu trúc js_event

▪ Nội dung các trường dữ liệu

- **Time:** nhãn thời gian phát sinh sự kiện
- **Value:** giá trị, phụ thuộc vào nút chức năng hay nút chỉnh hướng
 - Nếu là nút chức năng: 0/1 (1: pressed, 0: released)
 - Nếu là nút chỉnh hướng: -32768 -> 32767
(Phân biệt hướng x, y phụ thuộc vào giá trị number=0/1)
- **Type:** loại sự kiện
 - Khởi tạo thiết bị: 0x80
 - Nút chỉnh hướng: 0x02
 - Nút chức năng: 0x01
- **Number:** xác định nút được nhấn (trường hợp type =1 hoặc xác định nút chỉnh hướng x, y trong trường hợp type =2)

Lập trình kết nối joystick

- Mở file thiết bị:

```
joystick_fd = open(JOYSTICK_DEVNAME, O_RDONLY |  
O_NONBLOCK);
```

- JOYSTICK_DEVNAME: tên của file thiết bị, thường là /dev/input/js0
- O_RDONLY | O_NONBLOCK: mở file chỉ đọc ở chế độ NONBLOCK

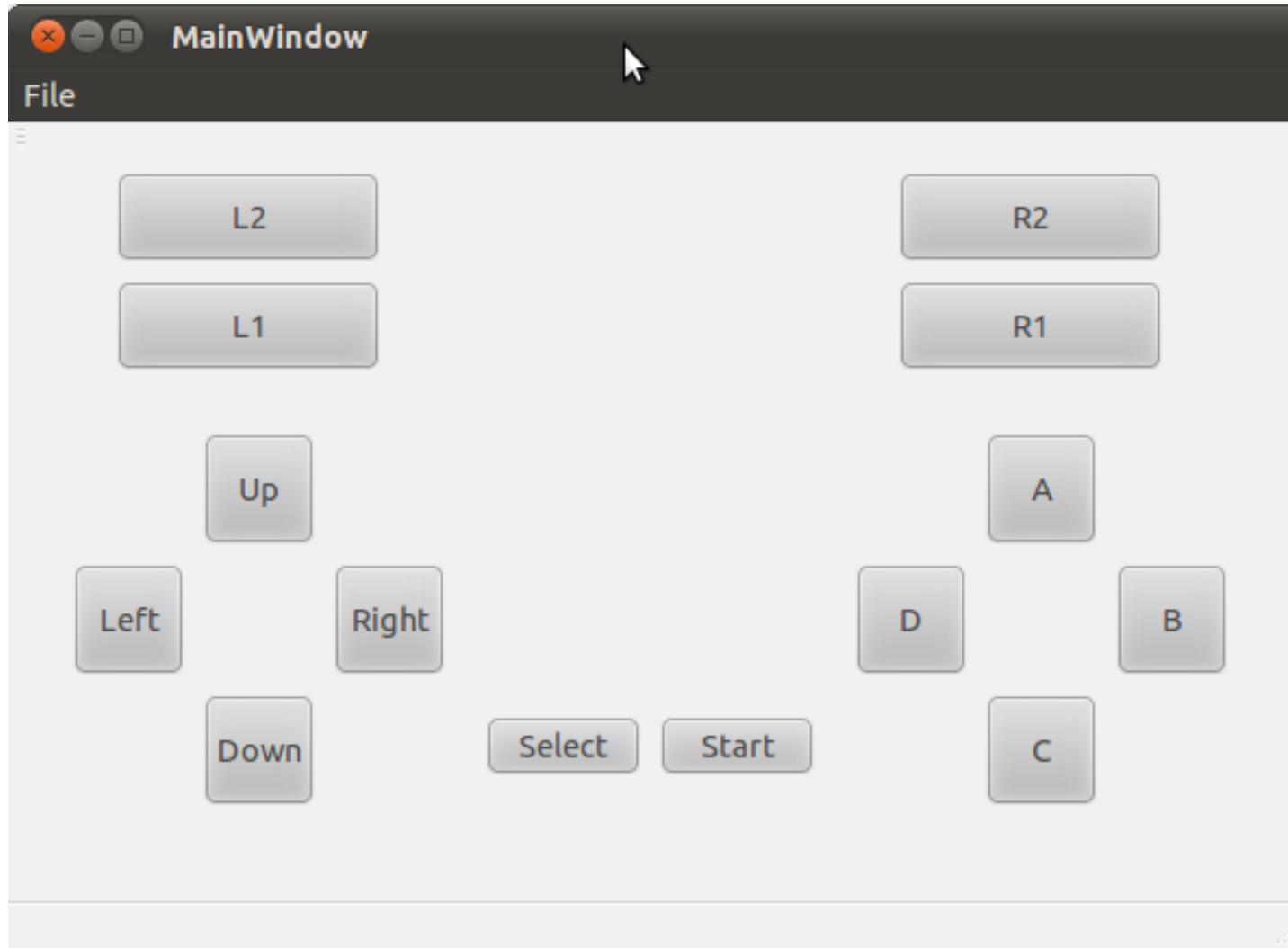
Lập trình kết nối joystick

- Đọc dữ liệu từ thiết bị (khi có phát sinh sự kiện)

bytes = read(joystick_fd, jse, sizeof(*jse));

- joystick_fd: con trỏ file có được khi mở file
- jse: biến cấu trúc js_event
- bytes: Tổng số file đọc được, nếu số này bằng kích thước của cấu trúc js_event thì quá trình đọc thành công

QT Joystick Demo



Chương 3

Các kỹ thuật nâng cao cho phát triển phần mềm nhúng

Nội dung

- Phát triển phần mềm hệ thống nhúng thông minh trên nền tảng hệ điều hành thường yêu cầu các kỹ thuật lập trình nâng cao:
 - Một hệ thống có nhiều tiến trình (chương trình), giao tiếp với nhau
 - Một tiến trình (chương trình) chia thành nhiều luồng xử lý (lập trình đa luồng)
 - Vấn đề đồng bộ đa luồng
 - Giao tiếp mạng (lập trình socket, http, ...)

Nội dung

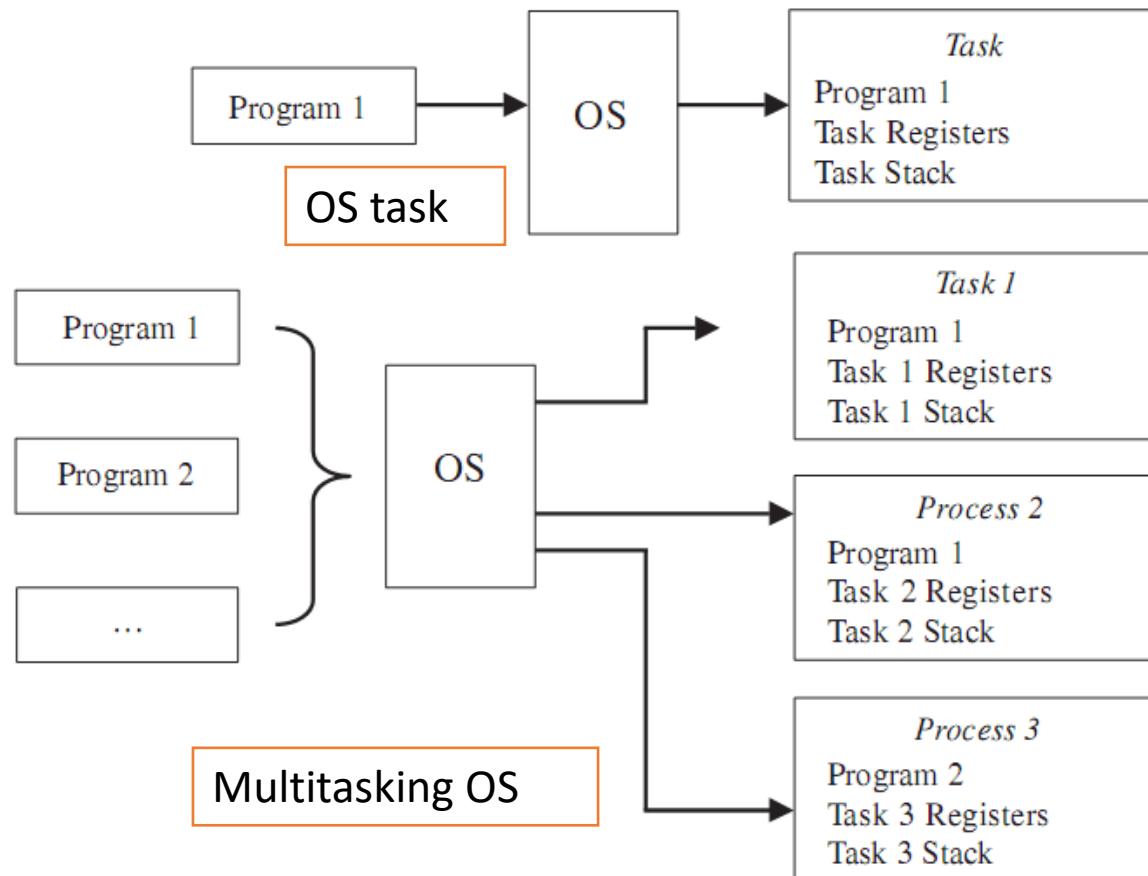
- 3.1. Tiến trình (process) và liên lạc giữa các tiến trình
- 3.2. Luồng (thread) và lập trình ứng dụng đa luồng
- 3.3. Lập trình socket trên Linux

3.1.1. Tiến trình (Process)

- Tiến trình (process) được tạo ra khi ta thực thi một chương trình (program)
- Kỹ thuật lập trình nâng cao thường sử dụng nhiều tiến trình kết hợp trong một ứng dụng để xử lý nhiều công việc.
- Ví dụ: Mở nhiều chương trình Word (mỗi cửa sổ soạn thảo nội dung một file khác nhau), mỗi thẻ hiện (instance) là một tiến trình (process)

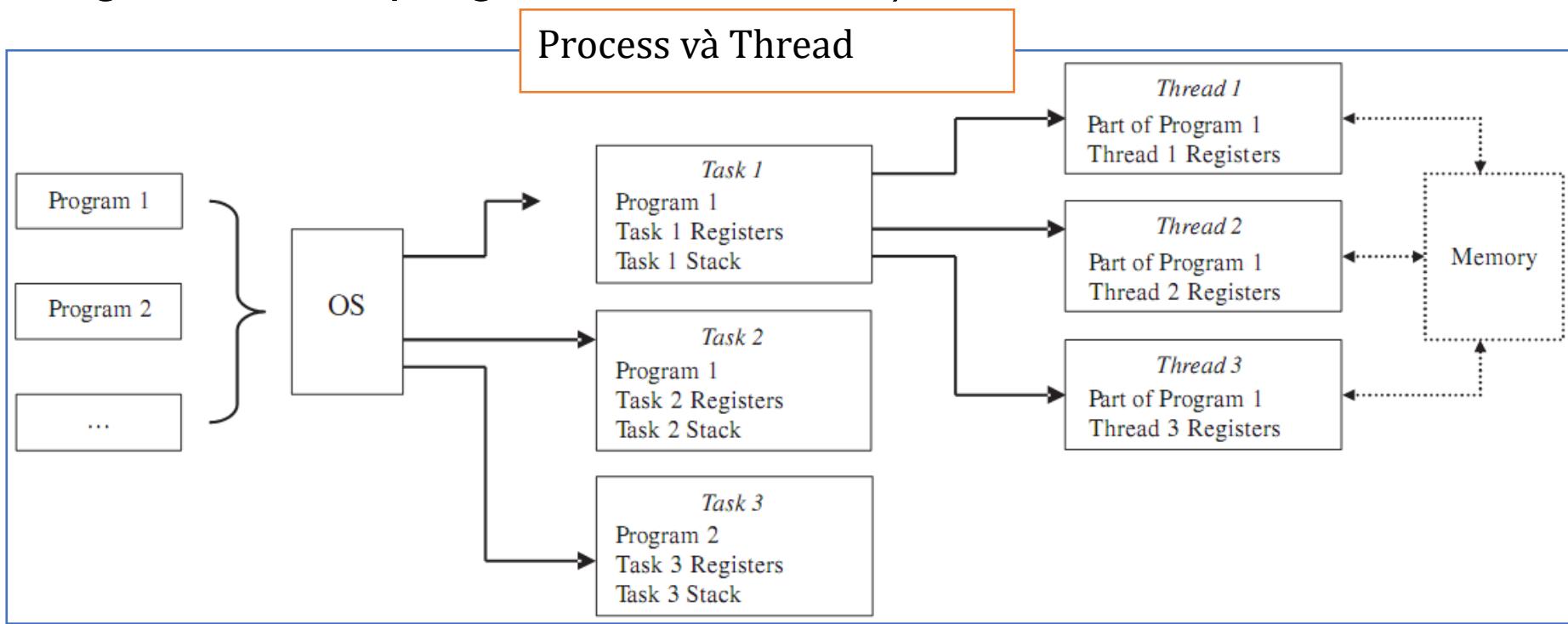
Process

- **Process ? (or task)** Được tạo ra bởi OS, đóng gói tất cả các thông tin liên quan đến việc thực thi một chương trình (program) như: stack, PC, source code, data, ...



Process và Thread

- Thread (luồng) được tạo ra trong một tiến trình.
- Một tiến trình có thể có 1 hoặc nhiều luồng thực thi
- Các luồng có thể chia sẻ cùng tài nguyên (files, io devices, global data, program code, v.v...)



Số hiệu tiến trình

- Làm thế nào để phân biệt các tiến trình ?
- Mỗi tiến trình được xác định bởi số hiệu (định danh):
 - PID (Process ID): số hiệu tiến trình
 - PPID (Parent Process ID): số hiệu tiến trình cha
- Lập trình C/C++ cung cấp hàm hệ thống để:
 - Lấy về PID: sử dụng hàm `getpid()`
 - Lấy về PPID: sử dụng hàm `getppid()`

```
#include<stdio.h>
#include<unistd.h>
int main() {
    printf("Process ID is %d\n", (int)getpid());
    printf("Parent process ID is %d\n", (int)getppid());
    return 0;
}
```

Xem tiến trình đang chạy

- Lệnh Linux: ps
- Xem nhiều thông tin hơn:

ps -e -o pid, ppid, command

Trong đó:

- e để xem tất cả tiến trình
- o pid, ppid, command để xem các thông tin tương ứng: pid, ppid và command (lệnh để thực thi tiến trình)

ps STAT (status)

```
osboxes@osboxes: ~
File Edit View Search Terminal Help
osboxes@osboxes:~$ ps -ax
 PID TTY      STAT   TIME COMMAND
  1 ?        Ss     0:02 /sbin/init splash
  2 ?        S      0:00 [kthreadd]
  4 ?        S<    0:00 [kworker/0:0H]
  6 ?        S<    0:00 [mm_percpu_wq]
  7 ?        S      0:00 [ksoftirqd/0]
  8 ?        S      0:01 [rcu_sched]
1178 pts/0    Ss     0:00 bash
1206 ?        S      0:01 [kworker/0:2]
4280 tty1    Sl+   0:27 /usr/lib/firefox/firefox -new-window
4339 tty1    Sl+   0:53 /usr/lib/firefox/firefox -contentproc -childID 1 -isF
4454 ?        S      0:00 [kworker/0:0]
4513 ?        S      0:00 [kworker/u2:0]
4520 ?        S      0:00 [kworker/u2:3]
4617 ?        S      0:00 /sbin/dhclient -d -q -sf /usr/lib/NetworkManager/nm-d
4662 tty1    Sl+   0:00 /usr/lib/firefox/firefox -contentproc -childID 4 -isF
4720 ?        S      0:00 [kworker/u2:1]
4721 ?        S      0:00 [kworker/u2:2]
4743 pts/0    R+    0:00 ps -ax
osboxes@osboxes:~$
```

+: foreground process

R: runnable

ps (process status)

- UID: user ID number
- TTY: Terminal type
- PID:
 - process identifier
 - 16-bit numbers (2^{16} -> 32,768)
 - wrap around
- PPID:
 - ID number of the process's parent process
- Time:
 - The amount of CPU time used by the process
- Cmd:
 - Simple name of executable

Lệnh top

```
top - 07:33:54 up 11 min, 1 user, load average: 1.52, 0.91, 0.53
Tasks: 238 total, 3 running, 168 sleeping, 0 stopped, 0 zombie
%Cpu(s): 4.6 us, 6.0 sy, 89.1 ni, 0.0 id, 0.3 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 985080 total, 64084 free, 681300 used, 239696 buff/cache
KiB Swap: 998396 total, 661500 free, 336896 used. 99068 avail Mem
```

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|------|------|----|----|---------|--------|-------|---|------|------|---------|-------------|
| 5801 | root | 39 | 19 | 168184 | 95540 | 56716 | R | 88.4 | 9.7 | 0:35.76 | unattended+ |
| 949 | root | 20 | 0 | 451680 | 22288 | 4192 | S | 4.6 | 2.3 | 0:12.77 | Xorg |
| 2024 | duy | 20 | 0 | 1245892 | 158432 | 22704 | S | 2.6 | 16.1 | 0:17.42 | compiz |
| 5839 | duy | 20 | 0 | 631812 | 30252 | 25220 | S | 1.7 | 3.1 | 0:00.44 | gnome-scre |
| 89 | root | 20 | 0 | 0 | 0 | 0 | R | 1.3 | 0.0 | 0:00.40 | kworker/0:2 |
| 8 | root | 20 | 0 | 0 | 0 | 0 | I | 0.3 | 0.0 | 0:00.83 | rcu_sched |
| 416 | root | 20 | 0 | 201540 | 2552 | 2060 | S | 0.3 | 0.3 | 0:00.94 | vmtoolsd |
| 5823 | duy | 20 | 0 | 48968 | 3580 | 2876 | R | 0.3 | 0.4 | 0:00.06 | top |
| 1 | root | 20 | 0 | 185432 | 3084 | 1752 | S | 0.0 | 0.3 | 0:02.48 | systemd |
| 2 | root | 20 | 0 | 0 | 0 | 0 | S | 0.0 | 0.0 | 0:00.00 | kthreadd |
| 3 | root | 20 | 0 | 0 | 0 | 0 | I | 0.0 | 0.0 | 0:00.41 | kworker/0:0 |

3.1.2. Tạo lập tiến trình mới

- Tại sao cần tạo tiến trình mới ?
 - Chia chương trình thành nhiều chương trình (process), hoạt động song song, thực hiện các tác vụ khác nhau.
 - Tăng tốc thực thi công việc, tận dụng lợi thế CPU đa lõi.
 - Chuyển điều khiển, tận dụng các đoạn mã thực thi có sẵn, không cần phát triển lại từ đầu.
 - Một ứng dụng có nhu cầu gọi đến tiến trình thực thi khác

Tạo lập tiến trình mới

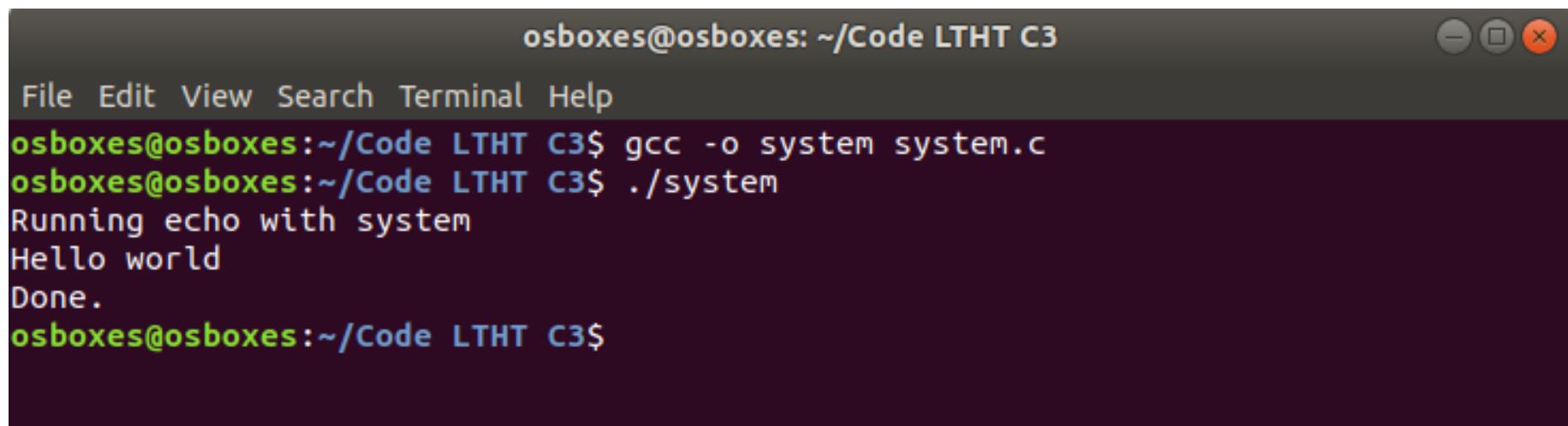
- Gọi tiến trình mới bằng hàm system()
- Thay thế tiến trình hiện hành bằng 1 tiến trình mới
- Nhân bản tiến trình với hàm fork()
- Kiểm soát và đợi tiến trình con
- Zombie process
- Chuyển hướng đầu vào đầu ra của tiến trình

Gọi tiến trình mới bằng hàm system()

- **Trường hợp 1:** Khởi tạo một tiến trình mới từ một tiến trình đang chạy bằng cách dùng hàm **system()**
- **#include <stdlib.h>**
int system (const char *string);

Ví dụ (lập trình C)

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    printf("Running echo with system\n");
    system("echo Hello world");
    printf("Done.\n");
    exit(0);
}
```



The screenshot shows a terminal window with the following content:

```
osboxes@osboxes: ~/Code LTHT C3
File Edit View Search Terminal Help
osboxes@osboxes:~/Code LTHT C3$ gcc -o system system.c
osboxes@osboxes:~/Code LTHT C3$ ./system
Running echo with system
Hello world
Done.
osboxes@osboxes:~/Code LTHT C3$
```

Ví dụ

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    printf("Running echo with system\n");
    system("echo Hello world&");
    printf("Done.\n");
    exit(0);
}
```

File Edit View Sea

```
osboxes@osboxes:~/Code LTHT C3$ gcc -o system system.c
osboxes@osboxes:~/Code LTHT C3$ ./system
Running echo with system
Done.
osboxes@osboxes:~/Code LTHT C3$ Hello world
```

osboxes@osboxes: ~/Code LTHT C3

```
File Edit View Search Terminal Help
osboxes@osboxes:~/Code LTHT C3$ gcc -o system system.c
osboxes@osboxes:~/Code LTHT C3$ ./system
Running echo with system
Done.
osboxes@osboxes:~/Code LTHT C3$ Hello world
echo Goodbye world
Goodbye world
osboxes@osboxes:~/Code LTHT C3$
```

Thay thế tiến trình hiện hành

- **Trường hợp 2: *Replacing a Process Image***
 - Thay thế tiến trình hiện hành bằng một tiến trình khác
- Trong trường hợp tiến trình A đang chạy mà muốn tiến trình B khởi chạy trong vùng nhớ của tiến trình A.
- Hàm exec() sẽ thay thế toàn bộ ảnh của tiến trình A (bao gồm mã lệnh, dữ liệu, bảng mô tả file) thành ảnh của tiến trình B. Chỉ có số định danh PID của tiến trình A là giữ lại.

Thay thế tiến trình hiện hành

- Hàm thay thế ảnh của tiến trình bao gồm tập các hàm execl, execlp, execle, execv, execvp, execve.
- Đa số hàm này sẽ yêu cầu phải có đối số path, file là đường dẫn đến tên chương trình cần thực thi trên đĩa.
- Các hàm exec sẽ không quay về nơi gọi trừ khi không nạp được tiến trình theo yêu cầu

Thay thế tiến trình hiện hành

- Các hàm exec() gồm một họ các hàm

```
#include <unistd.h>
```

```
char **environ;
```

```
int execl(const char *file, const char *args[i], ..., (char *)0);
```

```
int execlp(const char *file, const char *args[i], ..., (char *)0);
```

```
int execle(const char *file, const char *args[i], ..., (char *)0,  
          char *const envp[]);
```

```
int execv(const char *file, char *const args[]);
```

```
int execvp(const char *file, char *const args[]);
```

```
int execve(const char *file, char *const args[], char *const envp[]);
```

Ví dụ

File printf.c

```
#include<stdio.h>

int main()
{
    printf("Hello\n");
    return 0;
}
```

File execvp1.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
    printf("Beginning\n");
    char *args[]={"../printf", NULL};
    execvp(args[0], args);
    printf("Goodbye\n");
    return 0;
}
```

The terminal window shows the following session:

```
osboxes@osboxes: ~/Code LTHT C3/Exec family
File Edit View Search Terminal Help
osboxes@osboxes:~/Code LTHT C3/Exec family$ gcc -o printf printf.c
osboxes@osboxes:~/Code LTHT C3/Exec family$ gcc -o execvp1 execvp1.c
osboxes@osboxes:~/Code LTHT C3/Exec family$ ./execvp1
Beginning
Hello
osboxes@osboxes:~/Code LTHT C3/Exec family$
```

Ví dụ gọi một tiến trình trên Android (Java)

- Lập trình bằng Java

```
try
{
    Process p = Runtime.getRuntime().exec("/system/bin/pm
install -r smartlocker.apk");

} catch (Exception e)
{
    e.printStackTrace();
}
```

Nhược điểm của việc thay thế tiến trình

- Tiến trình mới chiếm toàn bộ không gian tiến trình cũ và chúng ta sẽ không có khả năng kiểm soát cũng như điều khiển tiến trình hiện hành của mình sau khi đã gọi hàm exec.
- Cách khắc phục là sử dụng hàm fork() để nhân bản hay tạo bản sao mới của tiến trình.

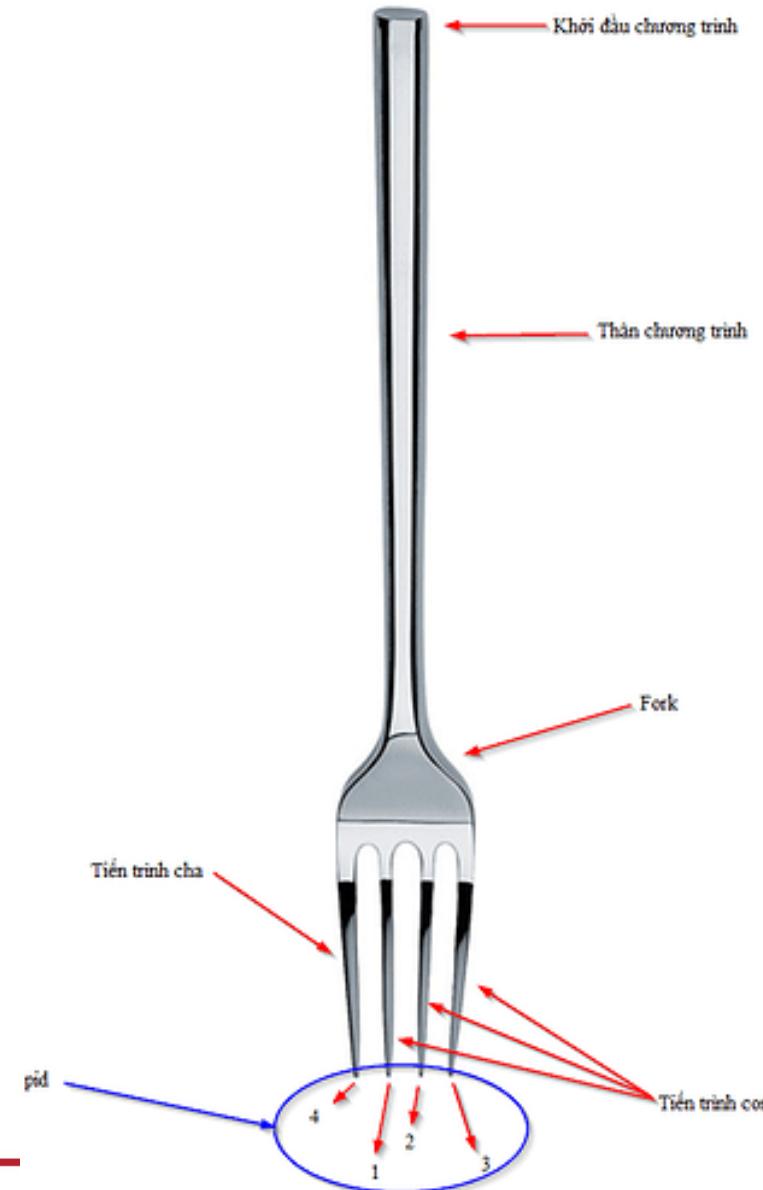
Nhân bản tiến trình bằng hàm fork()

Tiến trình mới (new process) được tạo ra là tiến trình con

Tiến trình gọi (calling process) là tiến trình cha

Tiến trình con gần như là 1 bản sao chính xác của tiến trình cha.

Tiến trình con và tiến trình cha có các số pid (process id) và ppid (parent process id) khác nhau.



Nhân bản tiến trình với hàm fork()

- Hàm fork được khai báo như sau:

```
#include <sys/types.h>
```

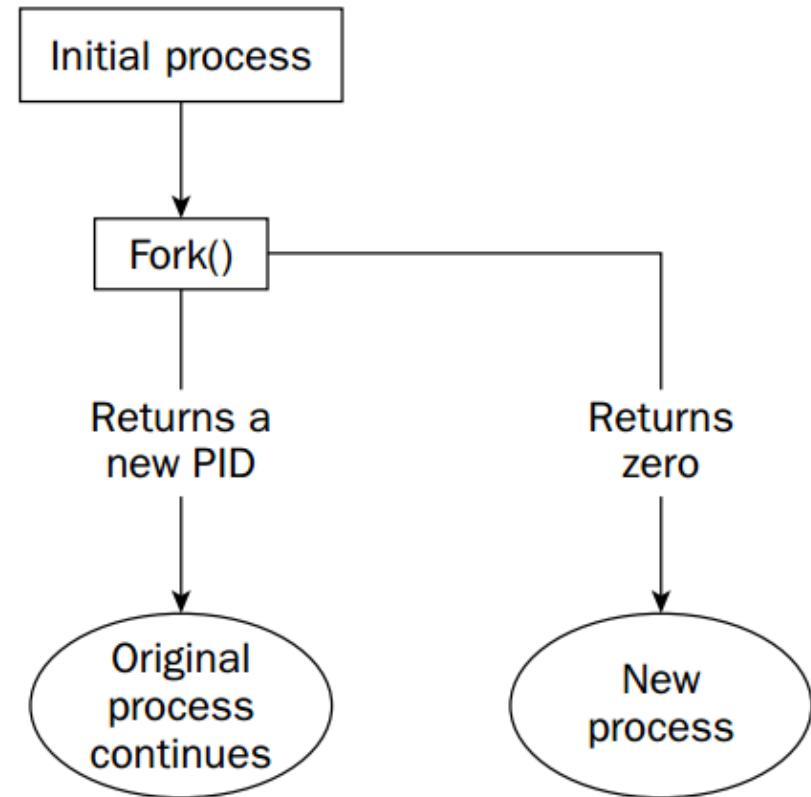
```
#include <unistd.h>
```

```
pid_t fork()
```

```
pid_t new_pid;
```

```
new_pid = fork();
```

```
switch(new_pid) {  
    case -1 : /* Error */  
        break;  
    case 0 : /* We are child */  
        break;  
    default : /* We are parent */  
        break;  
}
```



Ví dụ fork() tạo tiến trình con

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main() {
    pid_t pid;  char *message;  int n;
    printf("fork() program starting\n");
    pid = fork();
    switch (pid) {
        case -1:
            perror("fork failed");  exit(1);
        case 0:
            message = "This is the child";
            n = 20;  break;
        default:
            message = "This is the parent";
            n = 5;  break;
    }
    for ( ; n > 0; n--) {
        puts(message);
        sleep(1);
    }
}
```

Ví dụ socket server – multiple clients using fork()

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <signal.h>
#include <unistd.h>
int main()
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_in server_address, client_address;
    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
    server_address.sin_port = htons(9734);
    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
    /* Create a connection queue, ignore child exit details and wait for
     * clients. */
    listen(server_sockfd, 5);
    signal(SIGCHLD, SIG_IGN);
```

Server Program

Ví dụ socket server – multiple clients using fork()

```
while (1) {
    char ch;      printf("server waiting\n");
    /* Accept connection. */
    client_len = sizeof(client_address);
    client_sockfd = accept(server_sockfd, (struct sockaddr
*)&client_address, &client_len);
    /* Fork to create a process for this client and perform a test
to see whether we're the parent or the child. */
    if (fork() == 0) {
        /* If we're the child, we can now read/write to the client
on client_sockfd.*/
        read(client_sockfd, &ch, 1);
        ch++;
        write(client_sockfd, &ch, 1);
        close(client_sockfd);
        exit(0);
    }
    /* Otherwise, we must be the parent and our work for this
client is finished. */
    else { close(client_sockfd); }
}
```

Kiểm soát và đợi tiến trình con

- Khi fork() tách tiến trình ra làm hai tiến trình cha và con, một số trường hợp tiến trình cha cần phải đợi tiến trình con thực hiện xong mới tiếp tục thực thi.
- Hàm wait() giúp thực hiện công việc trên:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

- Hàm wait() được gọi trong tiến trình cha sẽ dừng lại chờ tiến trình con kết thúc trước khi thực hiện tiếp lệnh điều khiển trong tiến trình cha.

Kiểm soát và đợi tiến trình con

- Hàm wait(): Tiến trình cha đợi tiến trình con kết thúc

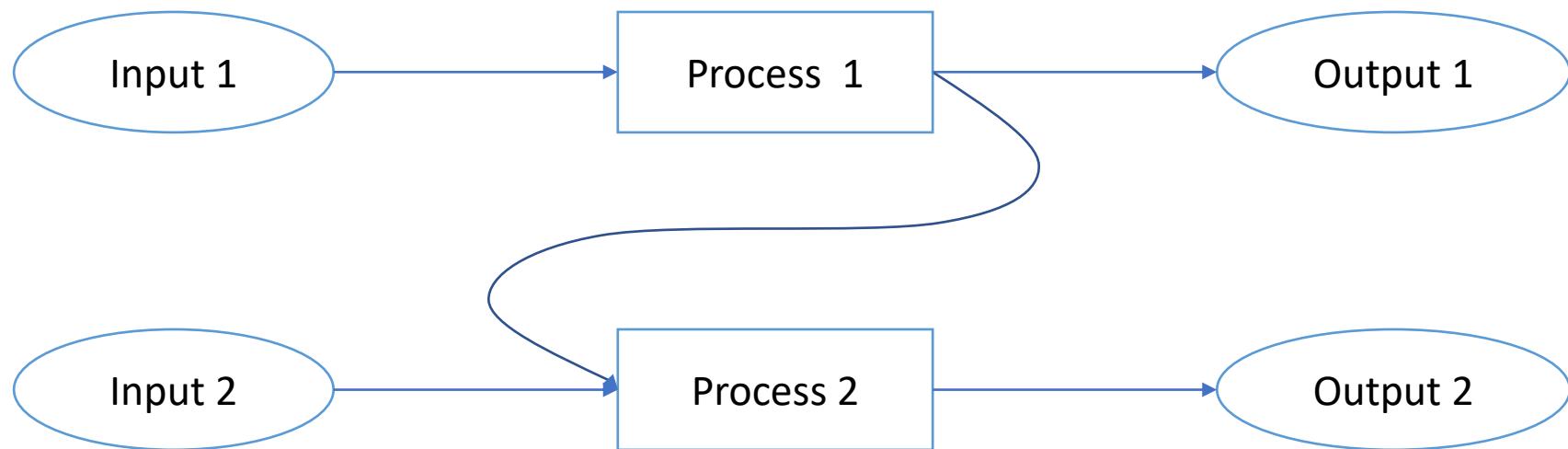
```
if (pid != 0) {  
    int stat_val;  
    pid_t child_pid;  
    child_pid = wait(&stat_val);  
    printf("Child has finished : PID = %d\n", child_pid);  
    if (WIFEXITED(stat_val))  
        printf("Child exited with code %d\n",  
               WEXITSTATUS(stat_val));  
    else  
        printf("Child terminated abnormally\n");  
}  
exit(exit_code);
```

Tiến trình Zombie

- Dùng fork() tạo tiến trình con là một cơ chế hiệu quả, tuy nhiên:
- Khi tiến trình con kết thúc, một liên kết với tiến trình cha được giữ cho đến khi tiến trình cha kết thúc hoặc gọi hàm wait(). (thông tin tiến trình con vẫn được lưu trong bảng thông tin tiến trình của hệ điều hành)
- Tiến trình con khi đó không còn ở trạng thái active, rơi vào trạng thái zombie (defunct)

Điều hướng input và output của tiến trình

- Mỗi tiến trình đều có input, output riêng biệt
- Tuy nhiên bằng cách điều hướng, ta có thể sử dụng output của tiến trình này làm input của tiến trình khác



Điều hướng input và output của tiến trình

- Cơ chế chuyển hướng input-output trong shell

```
dangnh@X:~$ cat myfile.txt  
  
dangnh@X:~$ echo 'Hello world!' > myfile.txt  
dangnh@X:~$ cat myfile.txt  
Hello world!
```

- Vận dụng các phương thức sinh tiến trình và chuyển hướng input-output, có thể tùy biến, tái sử dụng các đoạn mã có sẵn.

3.1.3. Signals – Liên lạc giữa các tiến trình

- Khái niệm
- Gửi tín hiệu đến tiến trình
- Đón bắt xử lý tín hiệu
- Cài đặt bộ xử lý tín hiệu
- Tránh tranh chấp xử lý tín hiệu

Cơ chế liên lạc giữa các tiến trình

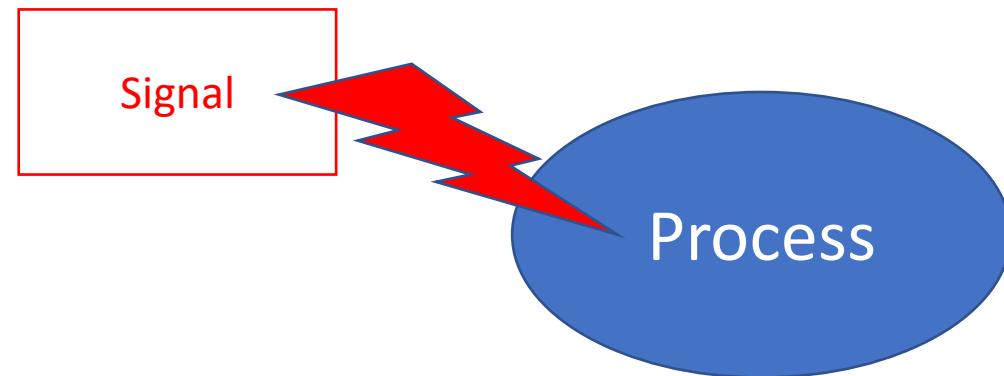
- Làm thế nào để các tiến trình có thể liên lạc (giao tiếp) với nhau ?
- Signal là cơ chế cho phép giao tiếp giữa các tiến trình
- Signal là cơ chế không đồng bộ
- Khi tiến trình nhận được signal, tiến trình phải xử lý signal ngay lập tức
- Linux hỗ trợ 32 SIGNAL

Signals

| Signal Name | |
|-------------|--------------------------------------|
| SIGABORT | Tiến trình bị bỏ dở |
| SIGALRM | Đồng hồ báo thức |
| SIGFPE | Lỗi dấu chấm động |
| SIGHUP | Hangup |
| SIGILL | Tập lệnh không hợp lệ |
| SIGINT | Gián đoạn terminal |
| SIGKILL | Kill (không thể bắt hay bỏ qua) |
| SIGPIPE | Lỗi ghi vào pipeline không có reader |
| SIGQUIT | Thoát terminal |
| SIGSEGV | Truy cập vùng bộ nhớ không hợp lệ |
| SIGTERM | Tiến trình bị hủy |
| SIGUSR1 | Người dùng định nghĩa |
| SIGUSR2 | Người dùng định nghĩa |

Gửi tín hiệu đến tiến trình

- Một tiến trình có thể gửi tín hiệu cho tiến trình khác, bao gồm cả chính nó
- Các trường hợp:
 - Gửi tín hiệu từ bàn phím.
 - Gửi tín hiệu từ dòng lệnh
 - Gửi tín hiệu bằng hàm hệ thống kill



Gửi tín hiệu từ bàn phím

- Ctrl-C: Hệ điều hành gửi tín hiệu INT (SIGINT) đến tiến trình đang chạy. Kết thúc tiến trình đang chạy (interrupt).
- Ctrl-Z: Hệ điều hành gửi tín hiệu TSTP (SIGTSTP) đến tiến trình đang chạy. Đưa tiến trình vào trạng thái dừng (suspend).
- Ctrl-\: Hệ điều hành gửi tín hiệu ABRT(SIGABRT) đến tiến trình đang chạy. Kết thúc ngay tức khắc (abort).

Gửi tín hiệu từ dòng lệnh

- Sử dụng lệnh: **kill [-SIGNAL] PID**
 - -SIGNAL: Tên tín hiệu
 - PID: Số hiệu tín trình cần gửi đến
- Ví dụ: kill -9 9616 (Gửi tín hiệu SIGKILL)
=> tiến trình có PID 9616 bị yêu cầu chấm dứt

Gửi tín hiệu bằng hàm hệ thống kill()

```
#include <sys/types.h>
```

```
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

- sig: tín hiệu xác định, pid: tiến trình.
- Trả về 0 nếu thành công
- Trả về -1 nếu lỗi
 - Tín hiệu không hợp lệ (errno: EINVAL)
 - Không có quyền (errno: EPERM)
 - Tiến trình không tồn tại (errno: ESRCH)

kill(PID, SIGNAL_TYPE)

Gửi tín hiệu bằng hàm hệ thống kill()

- Đoạn mã tạm dừng tiến trình bằng cách gửi tín hiệu STOP đến nó

```
#include <unistd.h>
```

```
#include <sys/types.h>
```

```
#include <signal.h>
```

```
pid_t my_pid = getpid();
```

```
/* Lấy về định danh của tiến trình hiện hành */
```

```
kill (my_pid, SIGSTOP);
```

```
/* gửi tín hiệu STOP đến tiến trình hiện hành */
```

Gửi tín hiệu

- Hàm alarm được sử dụng để cung cấp một tín hiệu SIGALRM trong tương lai (hẹn giờ).
- **#include <unistd.h>**
unsigned int alarm(unsigned int seconds);
- Hàm trả về số giây còn lại hoặc -1 nếu có lỗi.



Đón bắt xử lý tín hiệu

- Hầu hết các tiến hiệu có thể đón bắt được bởi tiến trình ngoại trừ một số tín hiệu hệ thống tiến trình không thể đón bắt được
- Với các tín hiệu hệ thống thì tiến trình buộc phải chấm dứt thực thi khi nhận được tín hiệu này.
- Hệ thống dành sẵn các hàm mặc định xử lý tín hiệu cho mỗi tiến trình

Ví dụ: Bộ xử lý tín hiệu TERM là gọi hàm exit(), chấm dứt tiến trình hiện hành. Bộ xử lý tín hiệu ABRT là gọi hàm hệ thống abort().

Cài đặt bộ xử lý tín hiệu

- Để sử dụng bộ xử lý tín hiệu thay cho bộ xử lý tín hiệu mặc định (của hệ điều hành) ta gọi hàm signal().

```
#include <signal.h>
```

```
void signal(int signum, void(*sighandler) (int));
```

- signum: Số hiệu của tín hiệu
- sighandler: Hàm xử lý tín hiệu
- Nếu tín hiệu được gửi đến tiến trình, hàm xử lý tín hiệu sẽ được gọi thực thi

Xử lý tín hiệu bằng hàm sigaction()

- Hàm này giải quyết vấn đề tranh chấp xảy ra khi nhiều tín hiệu gửi đến tiến trình cùng lúc.
- Khác biệt giữa sigaction() với signal():
 - sigaction() sau khi đón bắt và xử lý tín hiệu xong, hệ thống tự động khôi phục lại hàm xử lý tín hiệu mặc định

Xử lý tín hiệu bằng hàm sigaction()

- Khai báo:

```
#include <signal.h>
```

```
int sigaction(int sig, const struct sigaction  
*act, const struct sigaction *oldact)
```

- sig: là tín hiệu cần bắt
- *act: là con trỏ cấu trúc có kiểu sigaction chứa các thông tin về hàm xử lý tín hiệu

```
struct sigaction {  
    void(*) int sa_handler; /* ham dung xu ly tin hieu */  
    sigset_t sa_mask; /* tap cac tin hieu can khoa */  
    int sa_flags; /* flag cho biet trang thai khai phuc  
    sau khi xu ly du lieu*/  
}
```

3.2. Luồng và lập trình ứng dụng đa luồng

3.2.1. Luồng

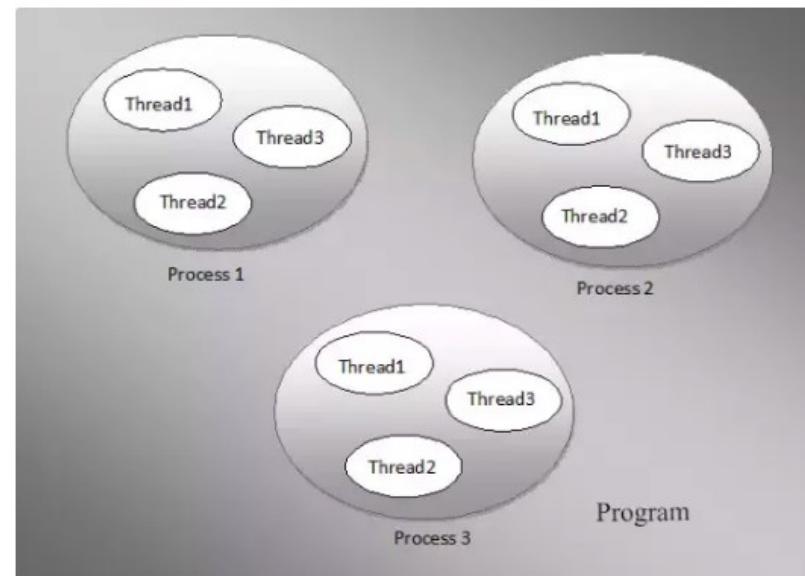
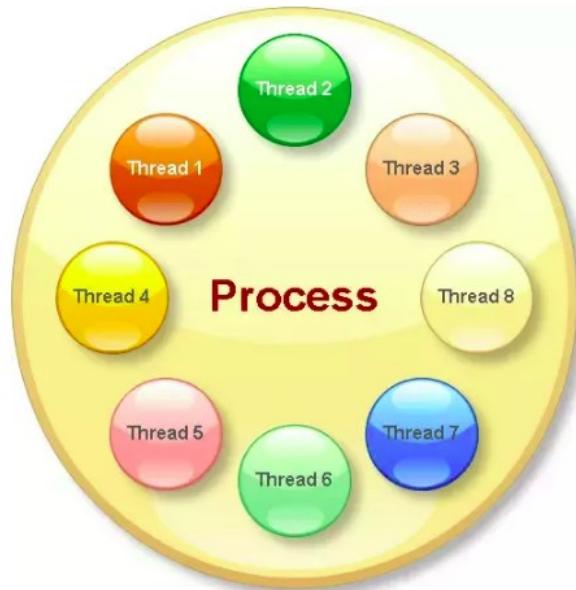
3.2.2. Lập trình luồng trên Linux

3.2.3. Mô hình đa luồng

3.2.4. Thiết lập thuộc tính cho luồng

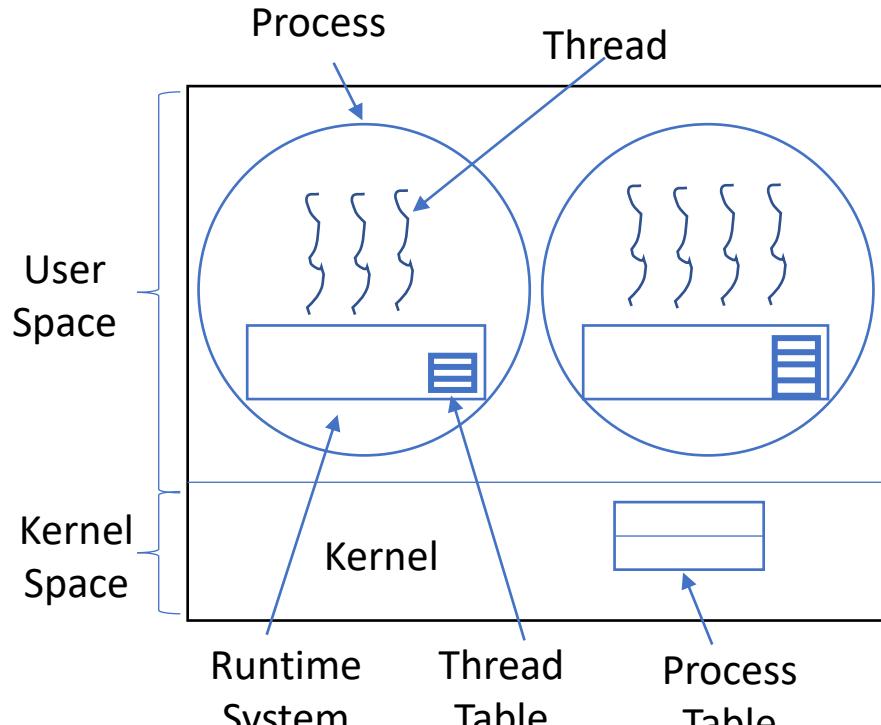
3.2.1. Luồng

- Luồng (Thread) được thực thi thông qua tiến trình (process)
- “Thread is a sequence of control within a process”

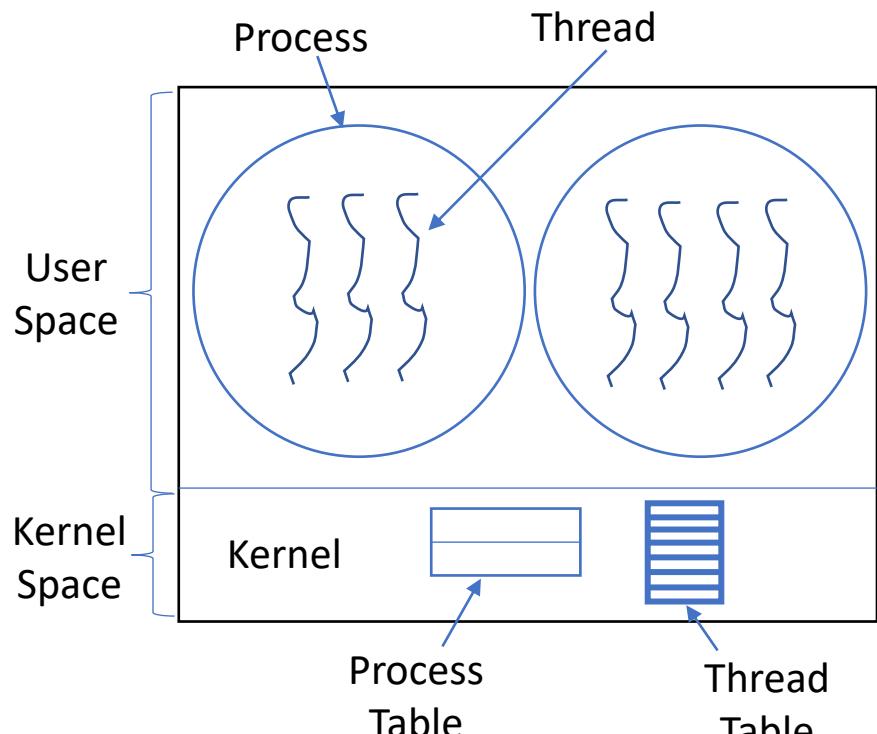


Program, Process, Thread

Phân loại luồng (Thread)



Cài đặt thread trong User Space



Cài đặt thread trong Kernel Space

Ưu điểm

- **Khả năng đáp ứng:** multi thread giúp các ứng dụng làm việc ngay cả khi một đoạn chương trình bị block hoặc cần thời gian để xử lý.
- **Khả năng chia sẻ tài nguyên :** giúp cho có nhiều threads hoạt động trong cùng một không gian địa chỉ chung mặc định bởi hệ thống.
- **Tiết kiệm kinh tế:** thread tự động chia sẻ data mà process chưa nó.
- **Sử dụng kiến trúc nhiều vi xử lý**

Nhược điểm

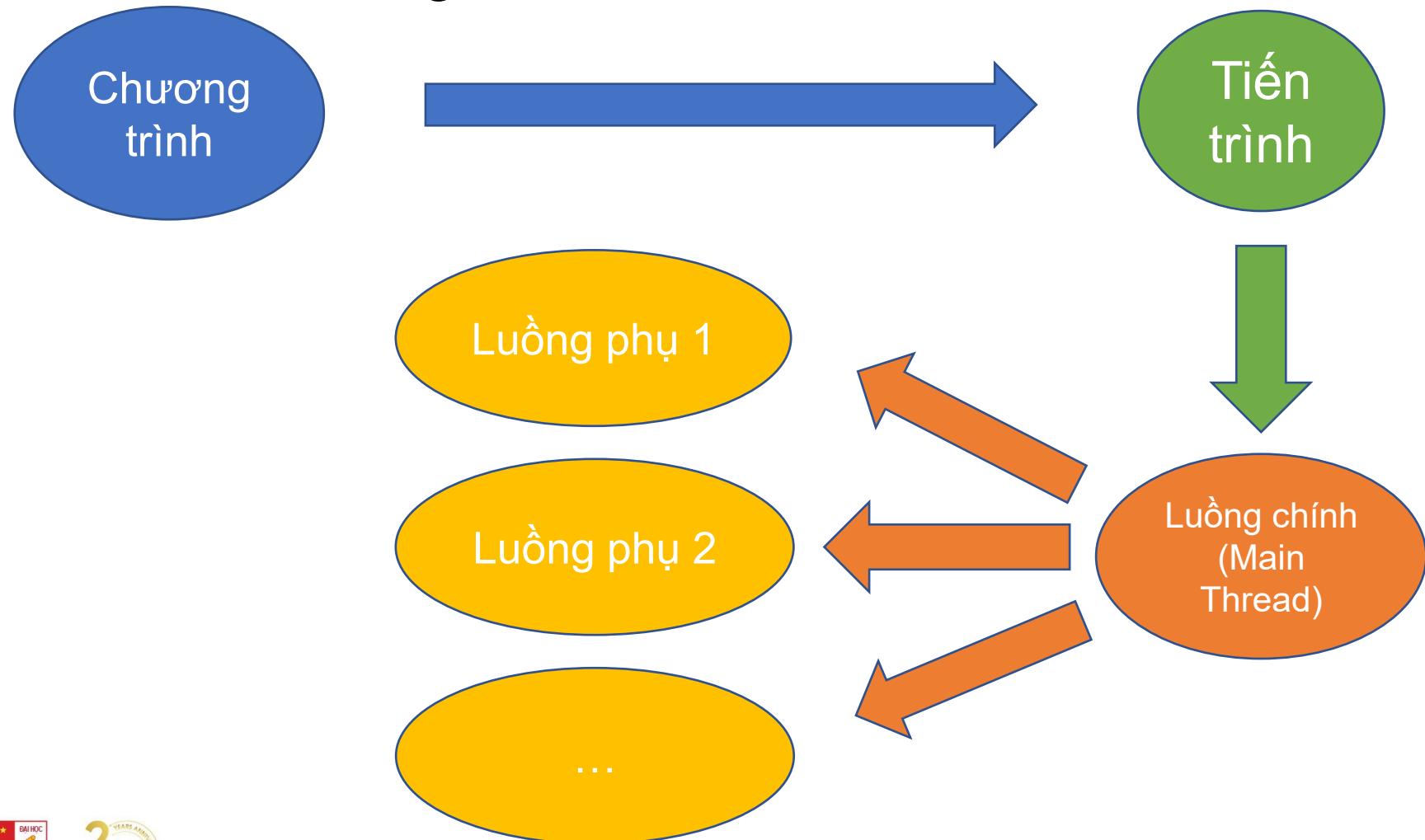
- Đòi hỏi thiết kế cẩn thận, do việc chia sẻ đồng bộ có thể dẫn đến các lỗi logic nghiêm trọng
- Gỡ lỗi 1 chương trình đa luồng khó hơn một chương trình đơn luồng bởi vì tương tác giữa các luồng là rất khó kiểm soát

3.2.2. Lập trình luồng trong Linux

- Giới thiệu chung
- Tạo luồng
- Hủy luồng

Giới thiệu chung

■ Mô hình chung



Thread trong Linux

- GNU/Linux cung cấp các hàm hệ thống (API) chuẩn **POSIX** (gọi là **pthreads**)
- Các hàm tương tác với luồng và kiểu dữ liệu được khai báo trong tệp tiêu đề **<pthread.h>**
- Các hàm **pthread** không được bao gồm trong thư viện C tiêu chuẩn mà ở dạng **libpthread**, vì vậy cần thêm **-lpthread** vào dòng lệnh khi **link** chương trình

```
$gcc -o demo demo.c -lpthread
```

Tạo luồng

```
pthread_create(pthread_t *thread,  
             const pthread_attr_t *attr,  
             void *(*start_routine) (void *),  
             void *arg);
```

- Con trả lưu trữ ID của luồng mới được tạo
- Con trả đến cấu trúc **pthread_attr_t** xác định các thuộc tính cho luồng mới
- Con trả tới hàm thực thi của luồng
- Đối số của hàm thực thi của luồng

Tạo luồng

Giá trị trả về của hàm `pthread_create()`

- Thành công: trả về 0
- Lỗi: trả về error number (EAGAIN, EINVAL, EPERM)

Ví dụ chương trình tạo thread

```
#include <pthread.h>
#include <stdio.h>

void *print_xs (void* unused)
{
    while (1)
        printf("0");
    return 0;
}

int main ()
{
    pthread_t thread_id;
    pthread_create (&thread_id, 0, &print_xs, 0);
    while (1)
        printf("1");
    return 0;
}
```

Tạo luồng: truyền dữ liệu cho luồng

- Đối số của luồng là kiểu **void** nên có thể truyền trực tiếp rất nhiều dữ liệu thông qua đối số
- Vấn đề: truyền dữ liệu cho nhiều luồng, nếu một luồng thực hiện xong và thoát trong khi các luồng còn lại vẫn đang truy cập bộ nhớ?

Tạo luồng: trạng thái của luồng

Một luồng có thể ở trạng thái joinable hoặc detached

- Nếu luồng ở trạng thái joinable

- Một luồng khác có thể gọi **pthread_join()** để chờ luồng kết thúc và lấy trạng thái thoát của nó
- Chỉ khi **một luồng joinable đã được joined** thì **tài nguyên của nó mới được giải phóng** trả lại hệ thống. Tức tài nguyên **không** tự động được dọn dẹp
- Trạng thái thoát của luồng treo xung quanh trong hệ thống (giống như tiến trình zombie)

- Nếu luồng ở trạng thái detached

- **Tài nguyên của nó sẽ tự động được giải phóng** trở lại hệ thống
- Hữu ích để đặt cho những luồng không cần quan tâm trạng thái kết thúc

pthread_join(pthread_t thread, void **retval)

Tạo luồng: luồng trả về giá trị

- Nếu đối số thứ hai truyền cho **pthread_join()** khác NULL, giá trị trả về của luồng sẽ được đặt ở vị trí được trả bởi đối số đó
- Giá trị trả về của luồng có kiểu **void***
- Nếu muốn trả về kiểu **int?**

Tạo luồng: định danh của luồng

Để xác định một luồng nào đó đang thực hiện, gọi **pthread_self()**

- Trả về ID của luồng được gọi
- ID này có thể được so sánh với ID của luồng khác bằng **pthread_equal()** để xác định xem một luồng cụ thể có tương ứng với luồng hiện tại hay không

```
if(!pthread_equal(pthread_self(), other_thread)
    pthread_join(other_thread, 0);
```

Tạo luồng: thuộc tính của luồng

Cấu trúc `pthread_attr_t` được tạo bằng `pthread_attr_init()` và các hàm liên quan. Nếu con trỏ trỏ đến cấu trúc này có giá trị `NULL`, luồng sẽ được tạo với thuộc tính mặc định.

```
typedef struct
{
    int __detachstate;
    int __schedpolicy;
    struct sched_param
    __schedparam;
    int __inheritsched;
    int __scope;
    size_t __guardsize;
    int __stackaddr_set;
    void *__stackaddr;
    unsigned long
    __stacksize;
}
pthread_attr_t;
```

| Thuộc tính | Ý Nghĩa | Giá trị mặc định |
|-----------------|--|-------------------------|
| __detachstate | Đặt trạng thái tách | PTHREAD_CREATE_JOINABLE |
| __schedpolicy | Đặt chính sách lập lịch | SCHED_OTHER |
| __schedparam | Thiết lập các tham số trong chính sách lập lịch | ¶m |
| __inheritsched | Đặt chính sách lập lịch kế thừa | PTHREAD_INHERIT_SCHED |
| __scope | Tạo luồng liên kết (bound thread) hoặc không liên kết (unbound thread) | PTHREAD_SCOPE_PROCESS |
| __guardsize | Cung cấp bảo vệ chống tràn của con trỏ ngăn xếp | |
| __stackaddr_set | Đặt địa chỉ bắt đầu ngăn xếp của luồng | |
| *__stackaddr | Luồng có địa chỉ ngăn xếp được phân bổ bởi hệ thống | NULL |
| __stacksize | Kích thước ngăn xếp | 1MB |

*Nếu không được lập lịch, không biết luồng gọi hay luồng mới được tạo sẽ thực hiện tiếp theo => luồng gọi và luồng mới mặc định không đồng bộ

Tạo luồng: thuộc tính của luồng

Các bước để tùy chỉnh thuộc tính của luồng

1. Tạo một đối tượng **pthread_attr_t**
2. Gọi **pthread_attr_init(pthread_attr_t *attr)**, truyền một con trỏ tới đối tượng này. Điều này khởi tạo giá trị mặc định cho các thuộc tính
3. Sửa đổi đối tượng thuộc tính với các giá trị mong muốn
4. Truyền đối tượng con trỏ đến đối tượng thuộc tính khi gọi **pthread_create**
5. Gọi **pthread_attr_destroy(pthread_attr_t *attr)** để giải phóng đối tượng thuộc tính. Bản thân **pthread_attr_t** không được giải phóng, nó có thể được khởi tạo lại với **pthread_attr_init**

*Không cần giữ đối tượng thuộc tính luồng sau khi luồng đã được tạo

Hủy luồng

Luồng chấm dứt theo một trong những cách sau

- Gọi **pthread_exit(void *retval)**
- Trả về giá trị từ **start_routine()**
- Bị hủy bởi một luồng khác, bằng cách luồng đó gọi **pthread_cancel(pthread_t thread)**
 - Nên joined luồng bị hủy để giải phóng tài nguyên, trừ khi luồng có trạng thái detached
 - Giá trị trả về của luồng bị hủy là giá trị đặc biệt được cung cấp bởi **PTHREAD_CANCELED**
- Gọi **exit()**, điều này **chấm dứt tất cả** các luồng trong tiến trình

Hủy luồng: đồng bộ & không đồng bộ

- Một luồng có thể ở một trong ba trạng thái liên quan đến hủy bỏ luồng
 - Bị hủy bỏ **không đồng bộ**: luồng có thể bị hủy tại bất kỳ điểm nào trong quá trình thực thi
 - Bị hủy **đồng bộ**: luồng chỉ bị hủy khi nó đạt đến các điểm cụ thể trong quá trình thực thi, các vị trí này gọi là các **điểm hủy**. Luồng sẽ xếp hàng một yêu cầu hủy cho đến khi đến điểm hủy tiếp theo. Để tạo điểm hủy, gọi **pthread_cancel()**
 - **Không thể hủy được**: nỗ lực hủy bỏ luồng bị bỏ qua

*khi mới được tạo, luồng có thể hủy **đồng bộ**

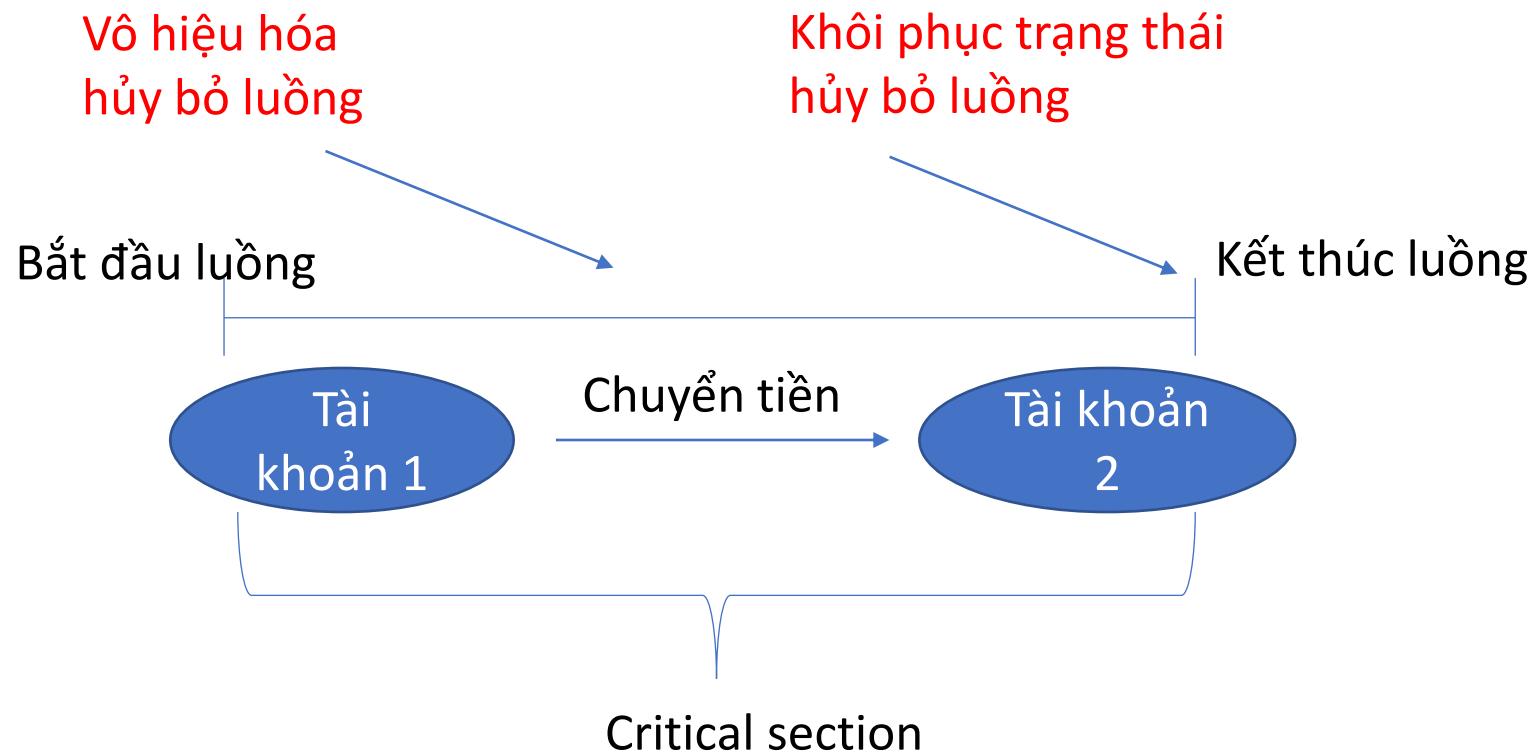
Để tạo một luồng có thể hủy **không đồng bộ**, sử dụng **pthread_setcanceltype(int type, int *oldtype)**

Hủy luồng: uncancelable critical sections

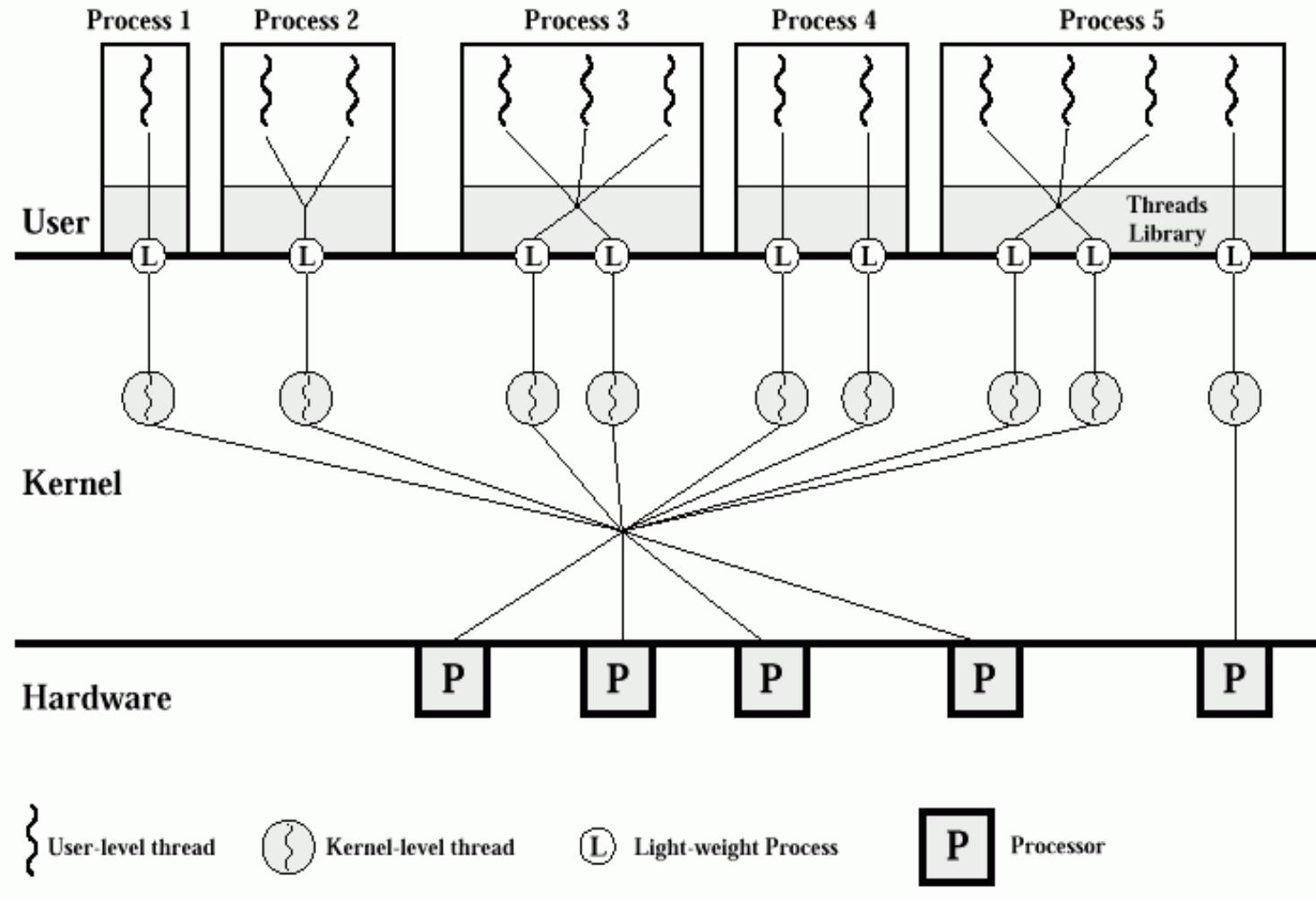
- Một luồng có thể vô hiệu hóa hủy bỏ chính nó hoàn toàn với hàm `pthread_setcancelstate(int state, int *oldstate)`
- Sử dụng `pthread_setcancelstate()` cho phép thực hiện các phần critical section
 - Một critical section (đoạn mã quan trọng), là một đoạn mã mà phải được thực thi toàn bộ hoặc không thực thi
 - Hay, nếu một luồng bắt đầu thực thi critical section, nó phải tiếp tục cho đến khi kết thúc critical section mà không bị hủy

Hủy luồng: uncancelable critical sections

Ví dụ: luồng chuyển tiền của ngân hàng bằng tài khoản



3.2.3. Mô hình đa luồng



Vấn đề đa luồng

- Kiểm soát thời điểm truy cập biến dùng chung

```
int x = 0, y = 1;
void T1() {
    while (1) {x = y + 1; printf("%4d",
x);}
}
void T2() {
    while (1) {y = 2; y = y * 2;}
}

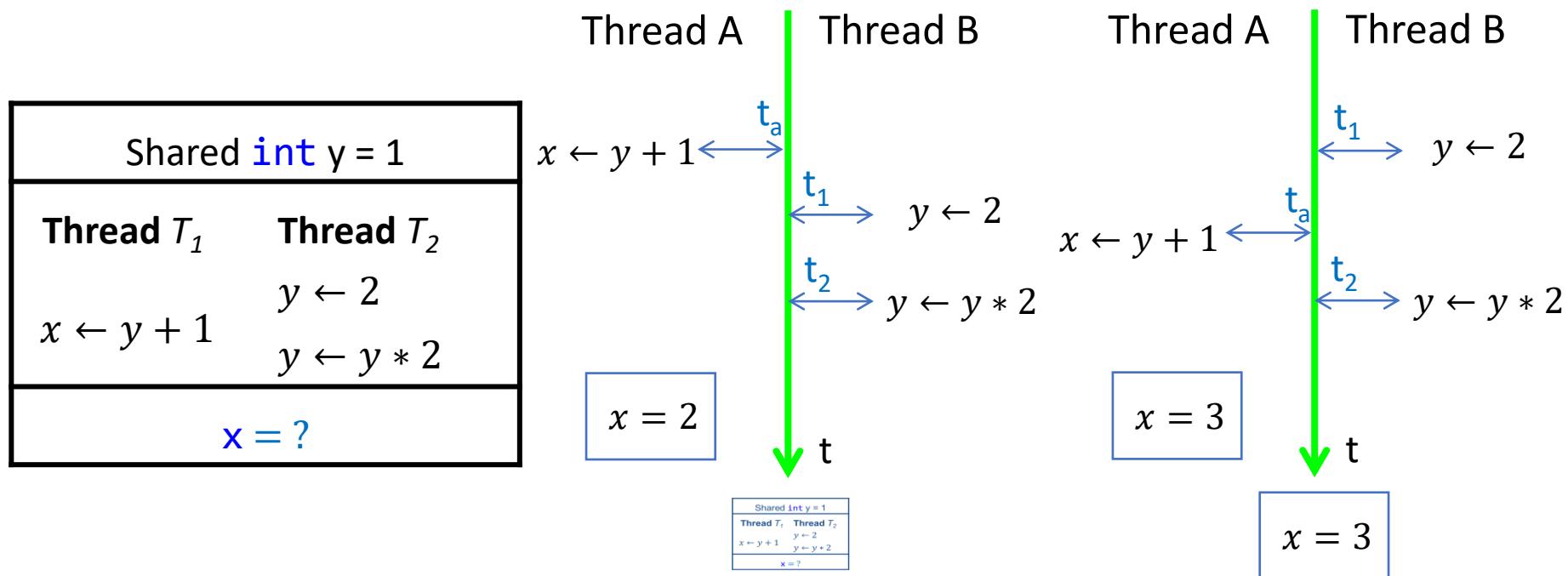
int main() {
    CreateThread(T1);
    CreateThread(T2);
}
```

Thread A

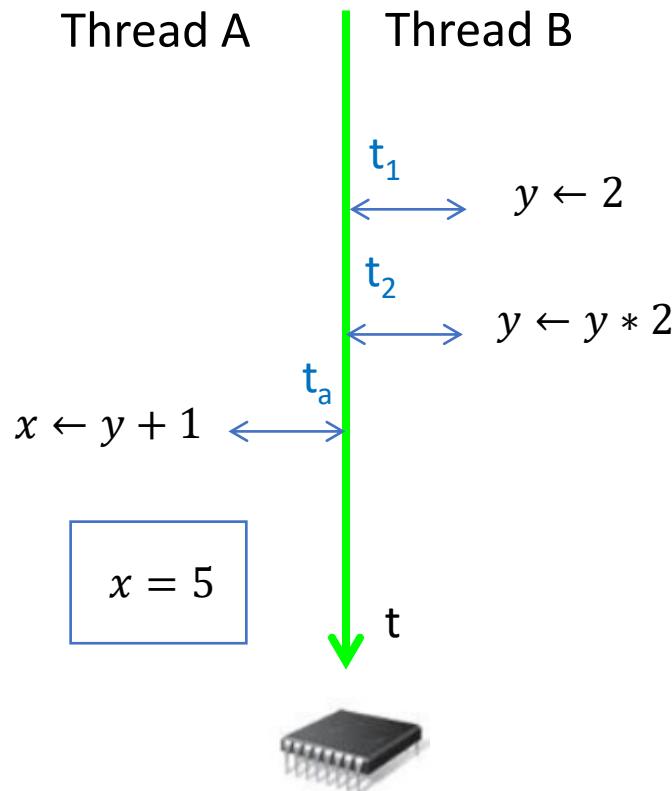
Thread B



Vấn đề đa luồng



Vấn đề đa luồng



Kết quả thực hiện các luồng song
song phụ thuộc **trật tự truy nhập**
các biến dùng chung

Vấn đề đa luồng

- **Critical Resources (tài nguyên gang):**
 - Tài nguyên hạn chế về khả năng sử dụng chung
 - Cần sử dụng đồng thời bởi nhiều luồng, tiến trình
 - Có thể là thiết bị vật lý (device) hoặc dữ liệu (data) dùng chung
- **Critical Section (Đoạn găng)** là đoạn code sử dụng tài nguyên găng
- **“Race Condition” (Điều kiện tương tranh):**
 - Kết quả chương trình **không xác định**, phụ thuộc vào thời điểm luồng truy cập tài nguyên gang
 - Có thể xuất hiện hoặc không --> Khó mô phỏng

Vấn đề đa luồng

- Để ngăn chặn các điều kiện tương tranh, cần đồng bộ (**Synchronize**) giữa các tiến trình/luồng đang thực hiện đồng thời
 - Đảm bảo chỉ một tiến trình/luồng truy cập tài nguyên chia sẻ tại một thời điểm
 - Các lệnh truy nhập tới tài nguyên chia sẻ trong các tiến trình/luồng phải thực hiện theo thứ tự xác định

Vấn đề đa luồng

- Deadlock (Bế tắc)



- Xảy ra khi hai hoặc nhiều tiến trình/luồng cùng chờ đợi một sự kiện nào đó xảy ra
- Nếu không có tác động của bên ngoài, thì sự **chờ đợi là vô hạn**

Đồng bộ đa luồng

- Đồng bộ luồng bằng biến toàn cục
 - Ví dụ: sử dụng biến toàn cục `run_now` để giao tiếp giữa các luồng
 - Kiểm tra giá trị `run_now` để in ra luồng tương ứng đang hoạt động.

□ Kết luận

- Việc lập lịch giữa các luồng là tự động
- **Kiểm soát luồng** bằng biến sẽ phức tạp khi số luồng và số tài nguyên tăng lên
- Phải chờ đợi tích cực (busy waiting) trước khi truy nhập tài nguyên

```
int print_count1 = 0;
while(print_count1++ < 20){
    if(run_now == 1){
        printf("1");
        run_now = 2;
    }
    else{
        sleep(1);
    }
}
```

Đồng bộ luồng

■ Hai kỹ thuật chính

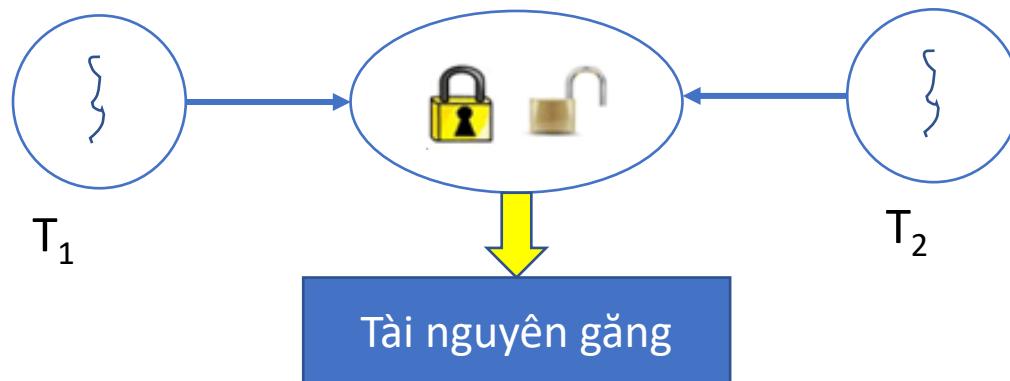
Mutex (Mutual Exclusion) – Loại trừ lẫn nhau

- Đảm bảo tài nguyên găng không phải phục vụ số lượng luồng hoặc tiến trình vượt quá khả năng của nó
- **Ví dụ:** Tại một thời điểm, khi một luồng đang sử dụng tài nguyên găng thì không một luồng nào khác được sử dụng nó

Semaphore (Đèn báo)

- Hoạt động như người “gác cổng” quanh đoạn mã nguồn
- **Ví dụ:** Sử dụng bộ đếm, kiểm soát truy nhập

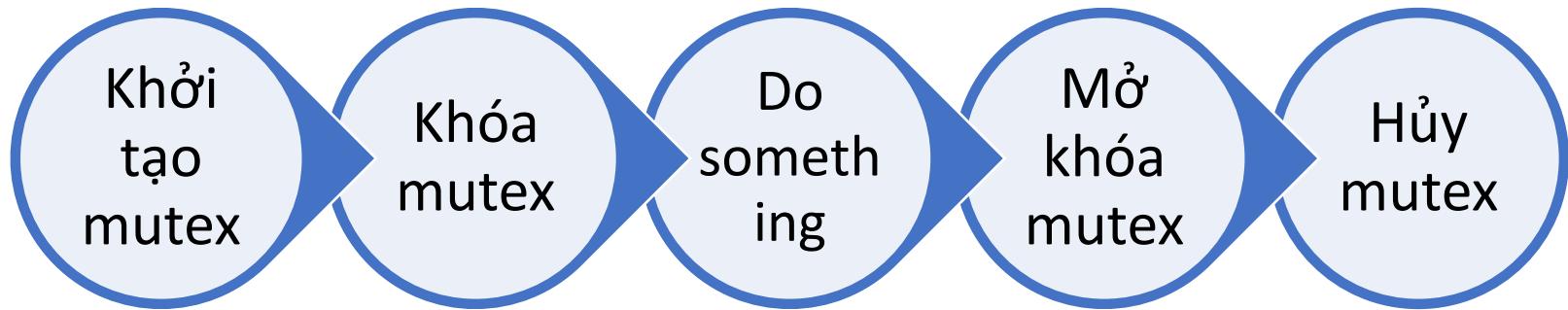
Đồng bộ luồng – Khóa Mutex



- Luồng vào đoạn găng, đóng khóa lại (lock)
- Luồng ra khỏi đoạn găng, mở khóa (unlock)
- Luồng khác muốn truy nhập tài nguyên găng: Kiểm tra khóa của các luồng khác
 - Đang khóa => Đợi
 - Đang mở khóa => Được phép truy nhập tài nguyên găng

Đồng bộ luồng – Khóa Mutex

■ Mô hình cài đặt



- Thư viện: #include <pthread.h>
- Một số hàm:
 - `int pthread_mutex_init(pthread_mutex_t *mutex,
const pthread_mutexattr_t *mutexattr)`
 - `int pthread_mutex_lock(pthread_mutex_t *mutex)`
 - `Int pthread_mutex_unlock(pthread_mutex_t *mutex)`
 - `int pthread_mutex_destroy(pthread_mutex_t *mutex)`

Đồng bộ luồng – Khóa Mutex

❖ Khai báo và khởi tạo biến mutex

- `pthread_mutex_t mutex;`
`pthread_mutex_init(&mutex, NULL);`
- `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`

- Khởi tạo biến mutex với các **giá trị mặc định**
- Thường áp dụng với **biến toàn cục**

Đồng bộ luồng – Khóa Mutex

- Khóa mutex cho phép một luồng chặn việc thực thi của một luồng khác
 - Luồng nào khóa (**lock**) thì sẽ chịu trách nhiệm mở khóa (**unlock**)
 - Tại một thời điểm, có thể có nhiều luồng đang bị chặn (**block**) chờ cho đến khi mutex được mở khóa
- ❖ Ba loại khóa mutex

Fast Mutex (Mutex nhanh)

- Mặc định
- **Deadlock**: Khóa một mutex 2 lần liên tiếp

Recursive Mutex (Mutex đệ quy)

- Tùy chọn
- Ghi **số lần lock** trên luồng khóa mutex. Số lần unlock phải bằng số lần lock

Error-Checking Mutex (Mutex kiểm tra lỗi)

- Tùy chọn
- Khóa một mutex 2 lần liên tiếp trả về **mã lỗi EDEADLK**

Đồng bộ luồng – Khóa Mutex

❖ Cài đặt các loại mutex khác

Khai báo

- `pthread_mutex_attr_t attr;`
- `pthread_mutex_t mutex;`

PTHREAD_MUTEX
_RECURSIVE_NP

Khởi tạo

- `pthread_mutexattr_init(&attr);`
- `pthread_mutexattr_setkind_np(&attr, PTHREAD_MUTEX_ERRORCHECK_NP);`
- `pthread_mutex_init(&mutex, &attr);`

Hủy thuộc tính

- `pthread_mutexattr_destroy(&attr);`

Đồng bộ luồng – Khóa Mutex

❖ Chế độ không chặn dừng (non-blocking)

- `int pthread_mutex_trylock(pthread_mutex_t *mutex)`

| | <code>pthread_mutex_trylock</code> | <code>pthread_mutex_lock</code> |
|--------------------|--|--|
| Mutex đang mở khóa | <ul style="list-style-type: none">▪ Trả về 0, và mutex bị khóa lại sau đó | |
| Mutex đang bị khóa | <ul style="list-style-type: none">▪ Trả về mã lỗi EBUSY▪ Các luồng gọi hàm <code>try_lock</code> sẽ không bị chặn dừng | <ul style="list-style-type: none">▪ Các luồng gọi hàm <code>lock</code> sẽ bị chặn. Chỉ trả về khi mutex được mở khóa. |

Đồng bộ luồng – Semaphore

- ❖ Dijkstra đề xuất 1972
- ❖ Điều phối nhiều luồng cùng truy cập tài nguyên găng mà không gây ra xung đột
- ❖ Cài đặt
 - Mức luồng (**thread**) gọi là POSIX Realtime Extension
 - Mức tiến trình (**process**) gọi là System V Semaphore
- ❖ Đặc điểm
 - Là biến đặc biệt có thể **tăng/giảm**
 - Giao tác truy nhập, thay đổi giá trị biến semaphore được hệ thống đảm bảo tính nguyên tử (atomic) – được sắp xếp theo thứ tự
- ❖ Có hai loại Semaphore
 - Nhị phân (**binary**): Điều phối 2 luồng thực thi trên tài nguyên găng
 - Bộ đếm (**counter**): Giới hạn số luồng thực thi trên tài nguyên găng

Đồng bộ luồng – Semaphore

■ Thư viện

- #include <semaphore.h>

■ Một số hàm

- int sem_init(sem_t *sem, int pshared, unsigned int value)
- int sem_wait(sem_t *sem)
- int sem_post(sem_t *sem)
- int sem_destroy(sem_t *sem)

➤ Trả về 0 khi thành công, và mã lỗi (Error Code) khi thất bại

Đồng bộ luồng – Semaphore

- Khai báo
 - `sem_t sem;`
- Khởi tạo
 - `int sem_init(sem_t *sem, int pshared, unsigned int value)`
 - Loại semaphore
 - `pshared = 0` thì sem do cho tiến trình hiện tại
 - `pshared ≠ 0` thì sem chia sẻ giữa các tiến trình. Chưa được hỗ trợ bởi Linux
 - Khởi tạo giá trị `value` cho semaphore

Đồng bộ luồng – Semaphore

- ❖ Kiểm soát giá trị của semaphore

| | Hàm sem_wait | Hàm sem_post |
|----------|---|---|
| Giao tác | <ul style="list-style-type: none">▪ Giảm giá trị semaphore đi 1 | <ul style="list-style-type: none">▪ Tăng giá trị semaphore lên 1 |
| sem = 0 | <ul style="list-style-type: none">▪ Chặn luồng gọi cho đến khi giá trị sem dương trở lại bởi một luồng khác | <ul style="list-style-type: none">▪ Tăng sem lên 1, kích hoạt một trong số các luồng bị chặn.▪ Gán lại sem = 0 |
| sem ≠ 0 | <ul style="list-style-type: none">▪ Giảm sem đi một đơn vị và thực thi luồng | <ul style="list-style-type: none">▪ Tăng sem lên một đơn vị |

- ❖ Hàm sem_trywait(sem_t sem)

- Nếu sem = 0 trả về ngay lập tức mã lỗi **EAGAIN** thay vì chặn luồng lại
- Nếu sem ≠ 0 giảm đi một đơn vị và thực thi luồng

Đồng bộ luồng – Semaphore

❖ Hàm hủy biến semaphore

- `int sem_destroy(sem_t *sem)`
- Giải phóng biến semaphore không cần thiết
- ✖ Có thể lỗi nếu đang có một số luồng chờ trên semaphore đó

❖ Hàm lấy giá trị hiện tại của semaphore

- `int sem_getvalue(sem_t *sem, int *sval)`
- Giá trị `sval` trả về giá trị hiện tại của semaphore
- ✖ Không nên quyết định gọi `post` hay `wait` dựa trên giá trị này vì có thể gây ra các điều kiện tương tranh.

Ví dụ đồng bộ đa luồng trên Android (Java)

- Đồng bộ giữa 2 luồng cùng làm việc với 1 cổng serial dùng Semaphore

```
public Semaphore domothreadSync = new Semaphore(1);
```

```
//Luồng 1 đọc dữ liệu cổng serial 100 ms một gói tin 5 bytes
```

```
public void run() {  
    if (domothreadSync.tryAcquire()) {  
        try {  
            //Đọc dữ liệu cổng serial 100ms một gói tin  
        } catch (Exception e) {}  
        finally {  
            domothreadSync.release();  
        }  
    }  
}
```

```
//Luồng 2 ghi 1 lệnh ra cổng serial và nhận 1 gói tin ack 5 bytes
```

```
public void run() {  
    if(domothreadSync.tryAcquire()) {  
        try {  
            //Ghi ra cổng serial 1 gói command và nhận 1 gói phản hồi  
        }  
        catch (Exception e){}  
        finally {  
            domothreadSync.release();  
        }  
    }  
}
```

3.3. Lập trình Socket trên Linux

- Giới thiệu lập trình socket
- Mô hình lập trình
- Minh họa

Giới thiệu lập trình socket

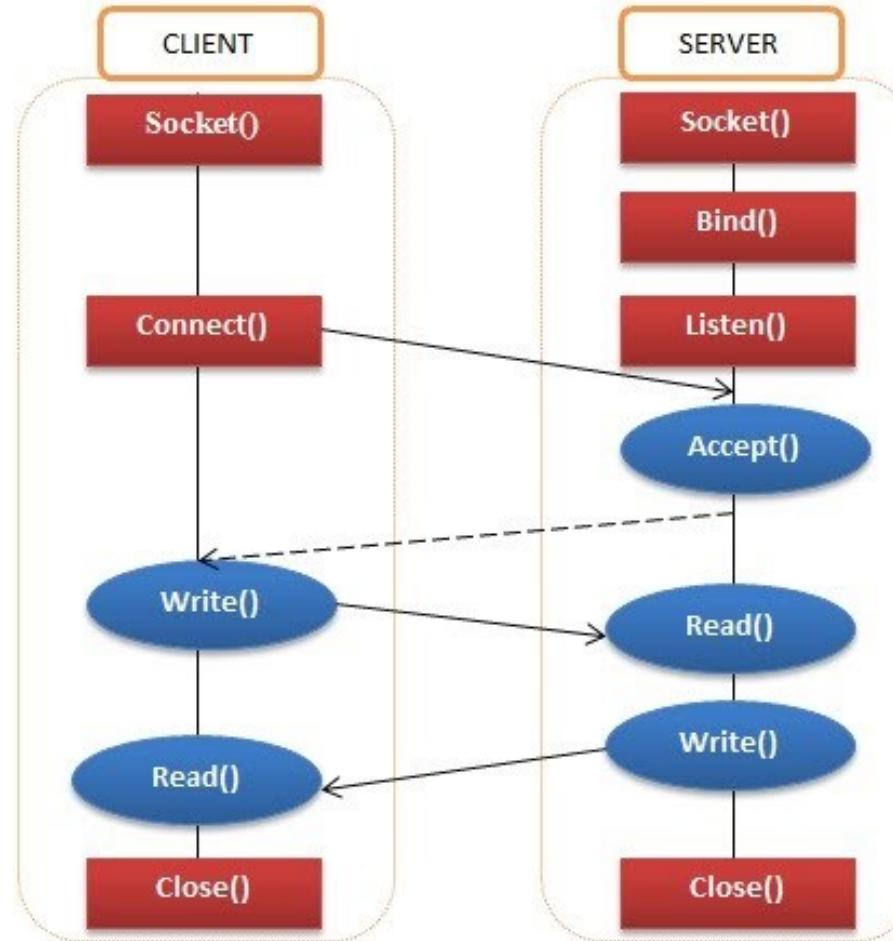
- Socket: Kết nối đầu cuối giữa 2 tiến trình/2 máy qua mạng (mô hình client/server)
- Tiến trình client kết nối đến tiến trình server yêu cầu trao đổi dữ liệu
- Client cần biết về địa chỉ và sự tồn tại của server, trong khi server không cần biết về client cho đến khi nó được kết nối đến.
- Mỗi khi thiết lập kết nối, cả 2 bên có thể gửi và nhận dữ liệu
- *Liên hệ như kết nối trong một cuộc gọi điện thoại*

Giới thiệu lập trình socket

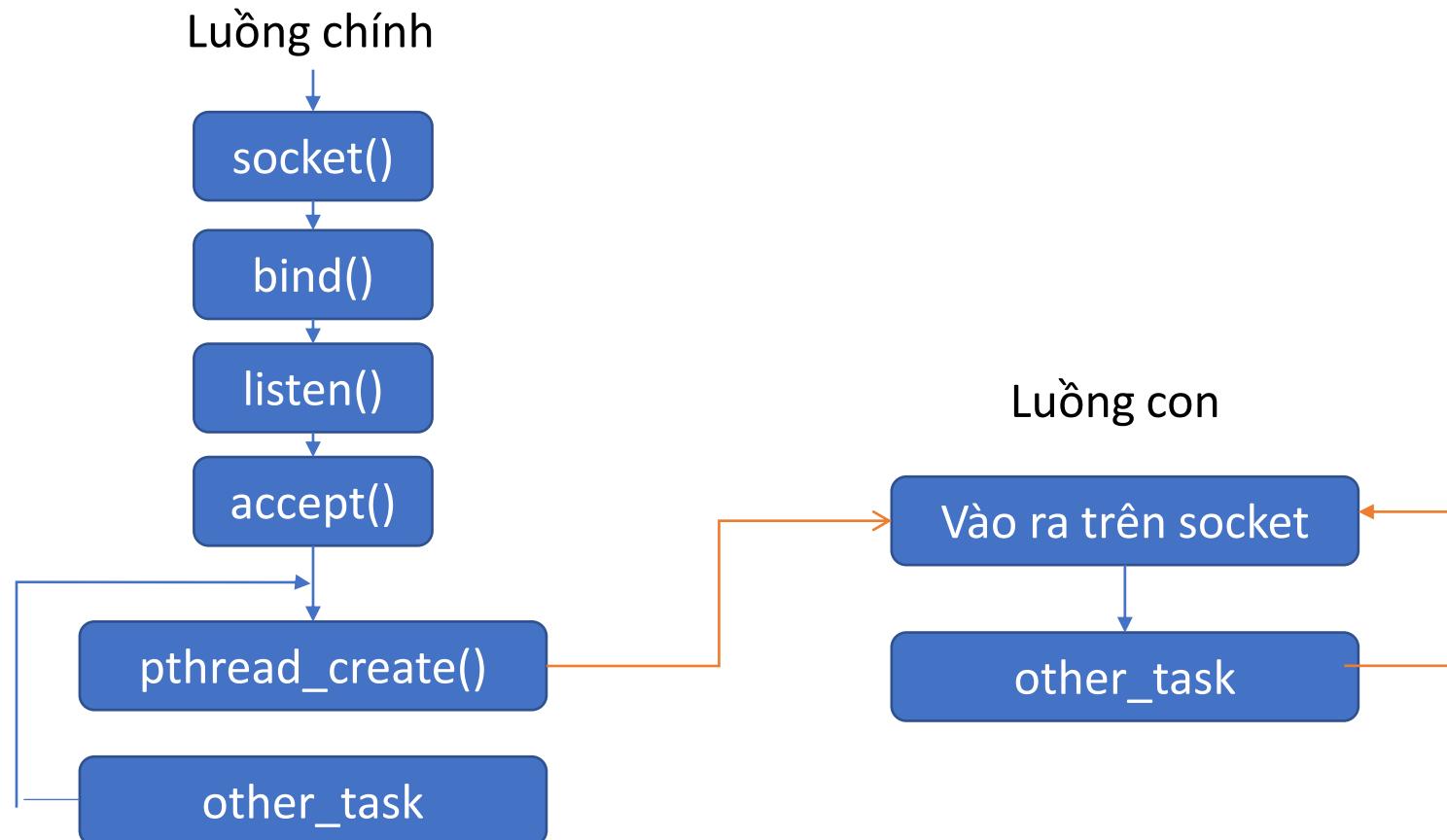
- Các hệ thống (Windows, Linux, ...) đều cung cấp các hàm hệ thống lập trình socket
- Có 2 loại socket sử dụng rộng rãi:
 - Stream socket
 - Datagram socket
- Stream sockets: Dựa trên giao thức TCP (Transmission Control Protocol), là giao thức hướng luồng (stream oriented).
- Datagram sockets: Dựa trên giao thức UDP (User Datagram Protocol), là giao thức hướng thông điệp (message oriented)

Mô hình lập trình socket

- Mô hình lập trình socket TCP giữa 2 tiến trình client/server



Ví dụ - Lập trình đa luồng với socket



Chương trình minh họa

- 2 tiến trình (Mã nguồn tham khảo):
 - server.c
 - client.c
- Biên dịch và chạy 2 chương trình này (trên cùng một máy local host, hoặc 2 máy riêng biệt kết nối mạng)

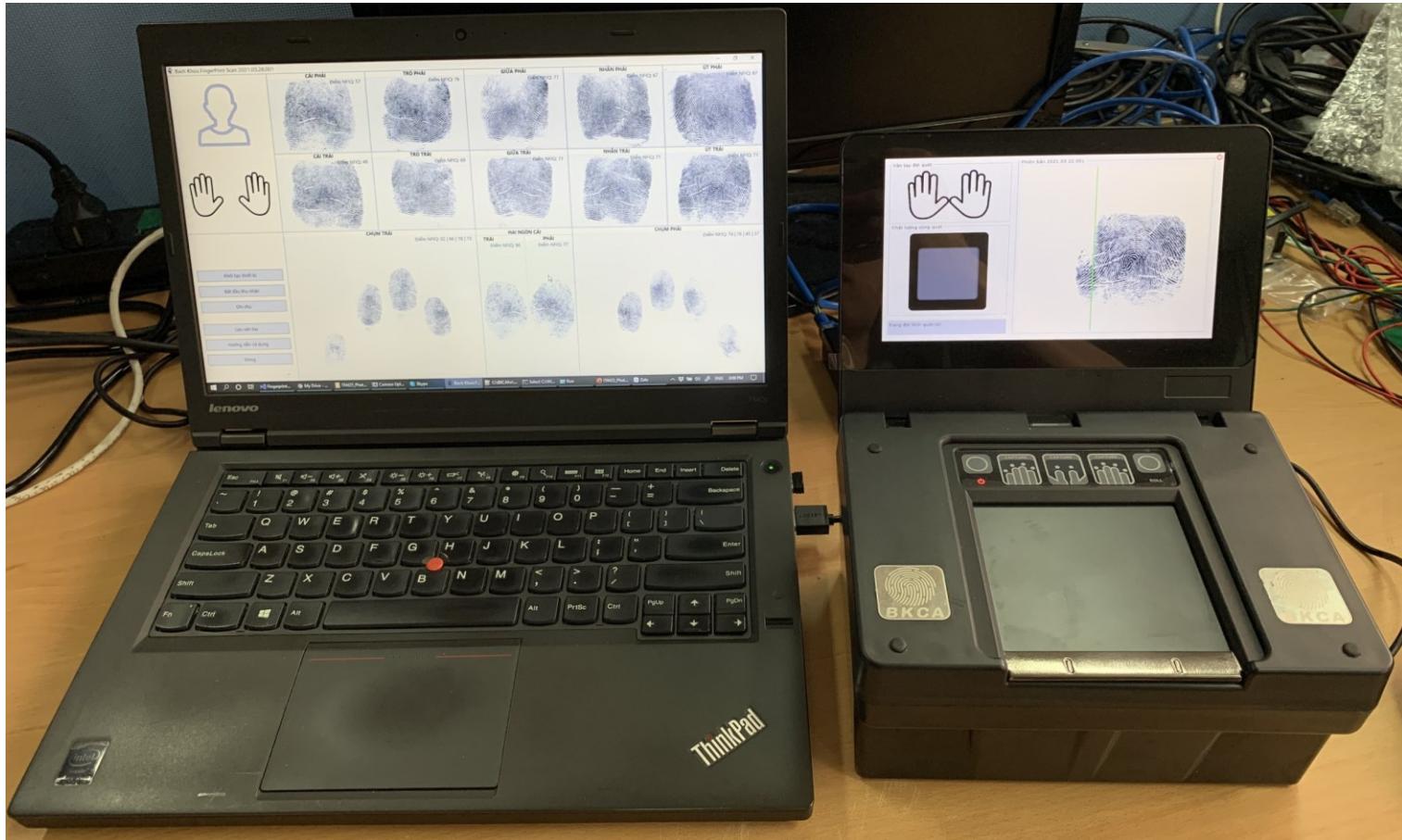
Demo

- Lập trình giao tiếp socket giữa KIT micro 2440 và PC



Ví dụ ứng dụng lập trình socket

- Giao tiếp sử dụng socket



Chương 4

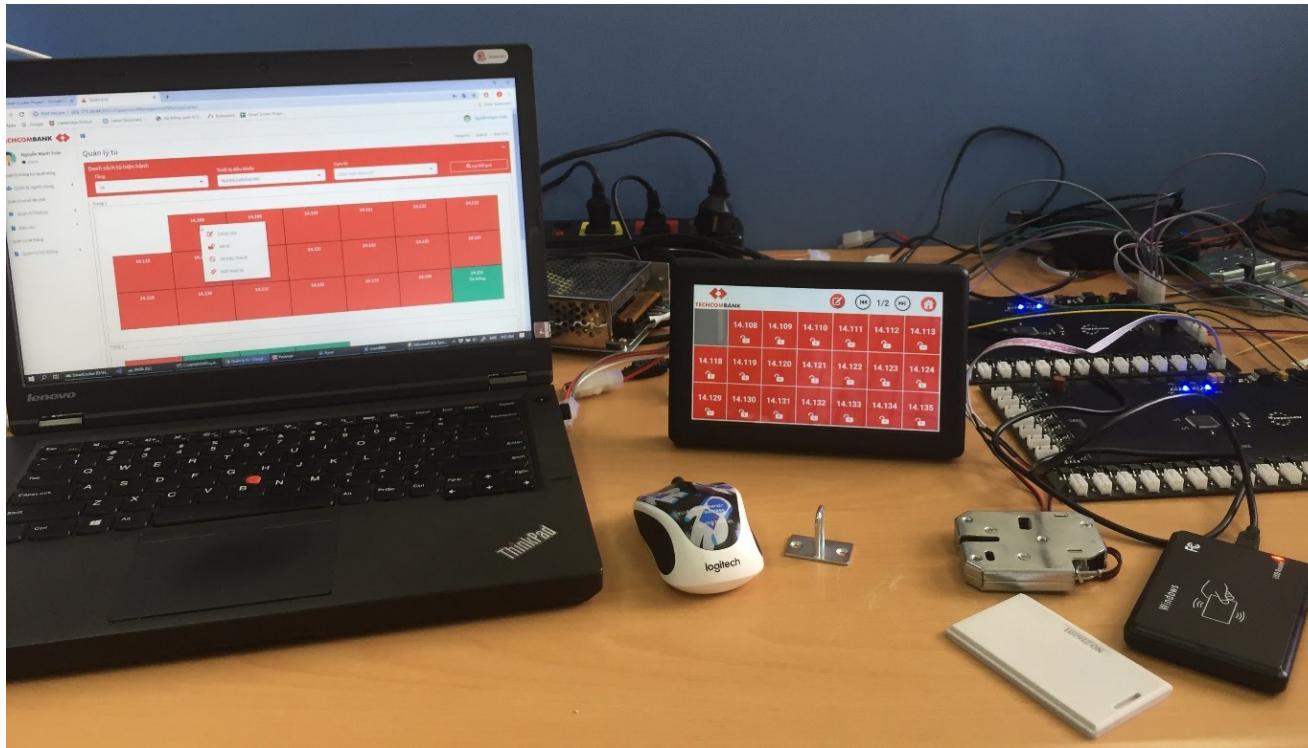
Cơ chế xây dựng device driver

Nội dung

- 4.1. Tổng quan Device Drivers
- 4.2. Xây dựng và thực thi Kernel modules
- 4.3. Cơ chế xây dựng character device driver
- 4.4. Tìm hiểu USB Device Driver

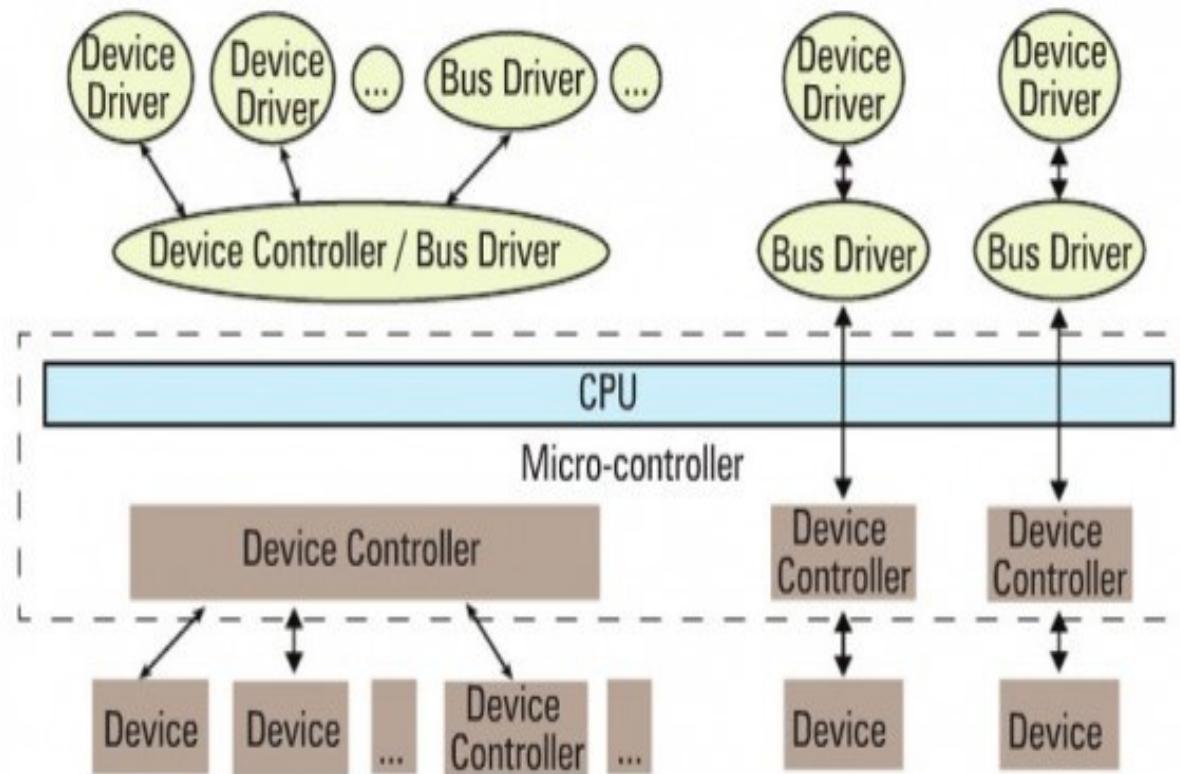
Xây dựng device driver ?

- Thiết bị phần cứng + hệ điều hành tùy biến
- Nhu cầu tùy biến hệ điều hành (Linux, Android, ...) với các tùy chỉnh về device drivers



4.1. Tổng quan về Device Drivers

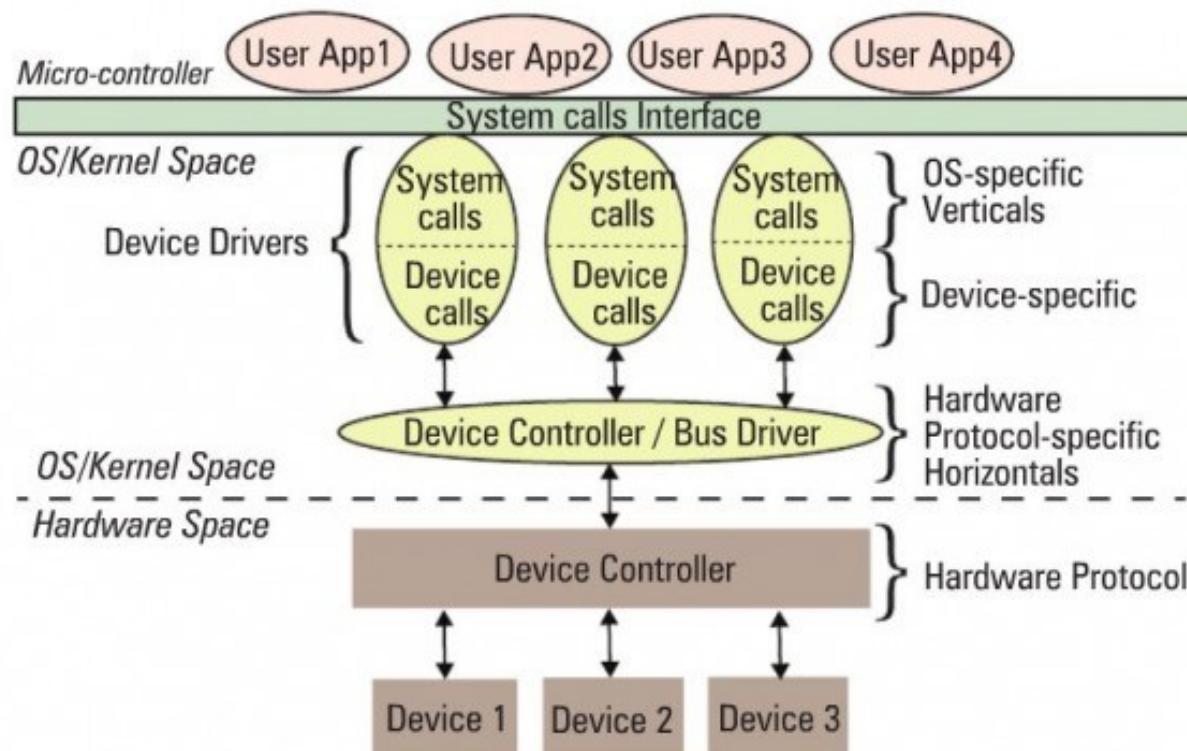
- **Drivers:** Trình điều khiển, có vai trò điều khiển, quản lý, giám sát một thực thể nào đó dưới quyền của nó
- Bus driver làm việc với một đường bus (giao thức phần cứng)
- Device driver làm việc với một thiết bị (đặc trưng theo thiết bị)



Device Drivers

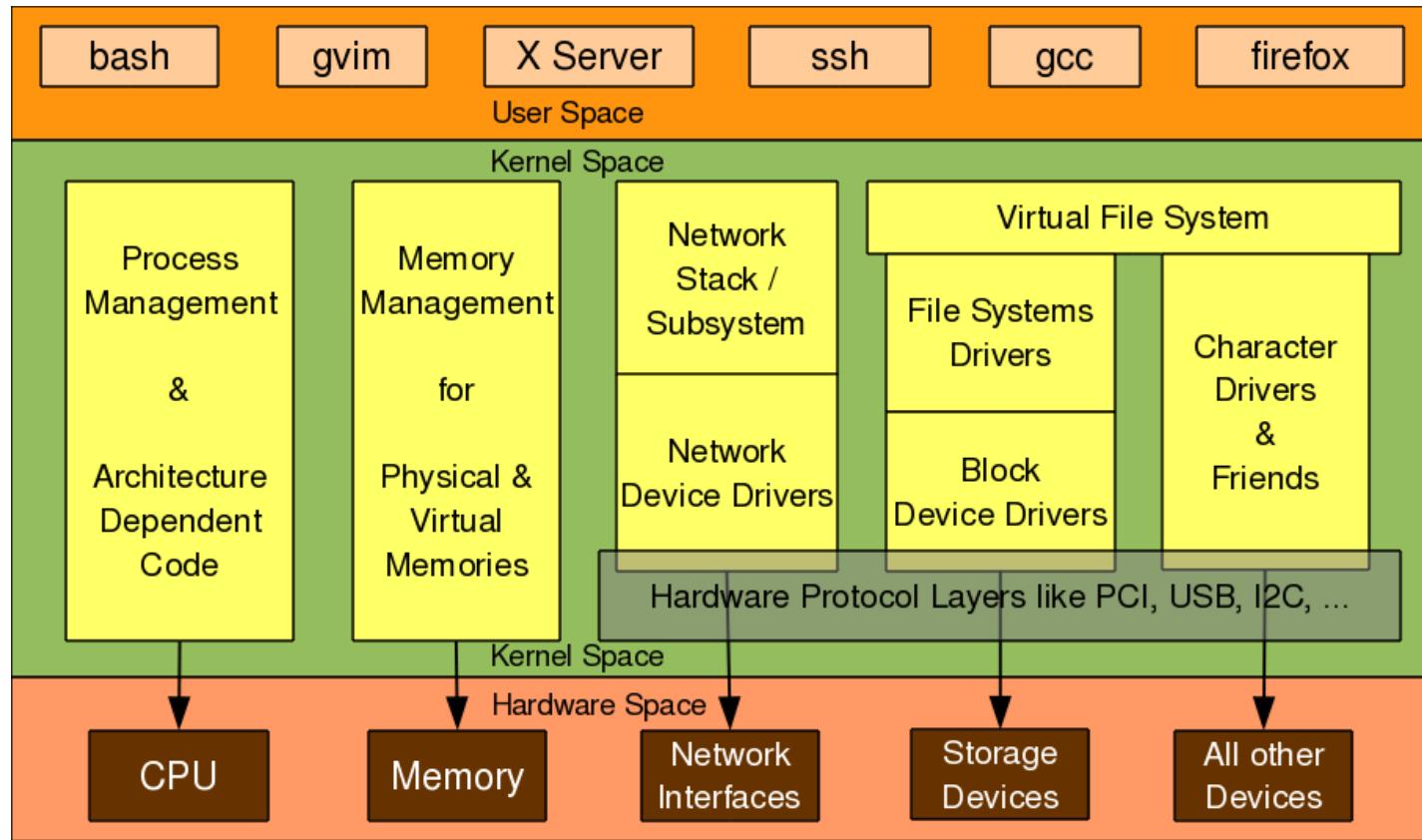
Một device driver bao gồm 2 phần

- Giao tiếp với thiết bị (Device-specific): giống nhau với các hệ điều hành
- Giao tiếp với hệ điều hành (OS-specific): gắn kết chặt chẽ với các cơ chế của hệ điều hành



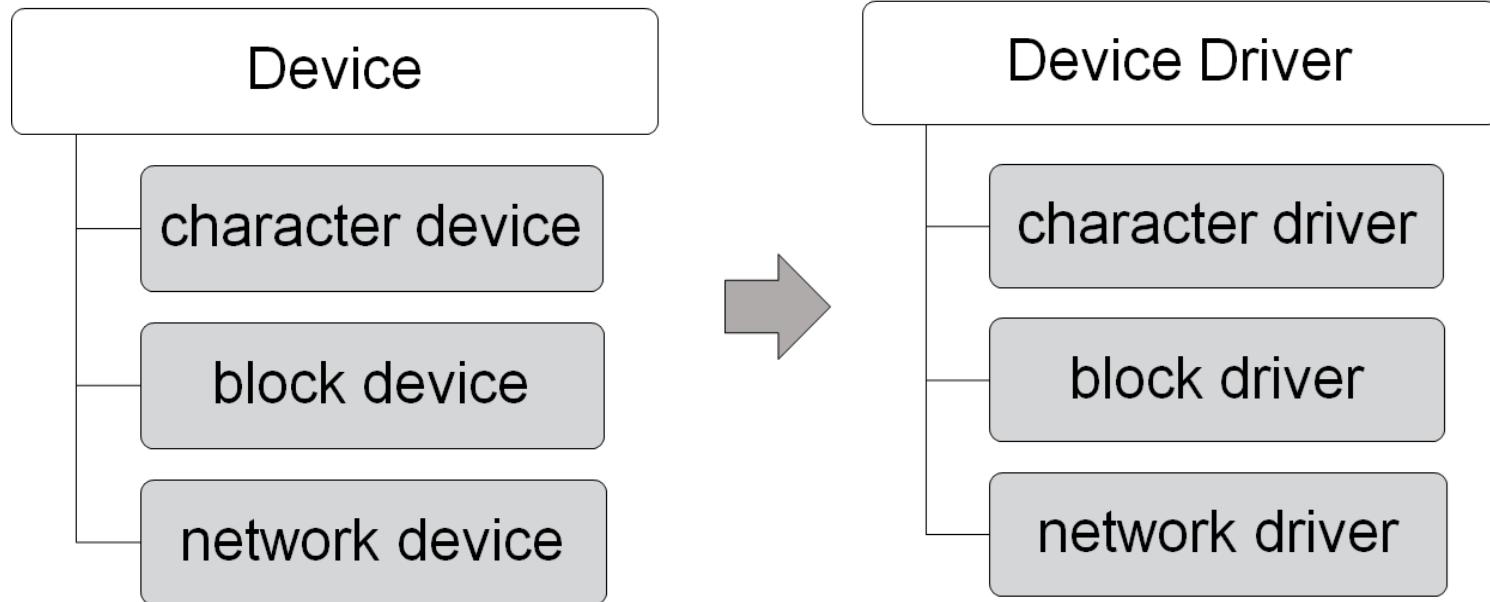
Device Drivers

- Device driver cung cấp giao diện “system call” (lời gọi hàm hệ thống cho các ứng dụng)



Device Drivers

- Các thiết bị có thể chia thành 3 loại dựa trên lượng dữ liệu trao đổi với CPU
- Device drivers được phân loại tương ứng



Character drivers

- Character devices: Thiết bị có thể được truy cập dưới dạng một luồng byte, sử dụng các lời gọi hệ thống open, close, read & write
- Vd: console, serial port
- Character driver: chia thành các lớp con như tty driver, input driver, console driver, frame-buffer drivers, sound driver, .. tương ứng với các giao tiếp như RS232, PS/2, VGA, I2C, SPI, ..

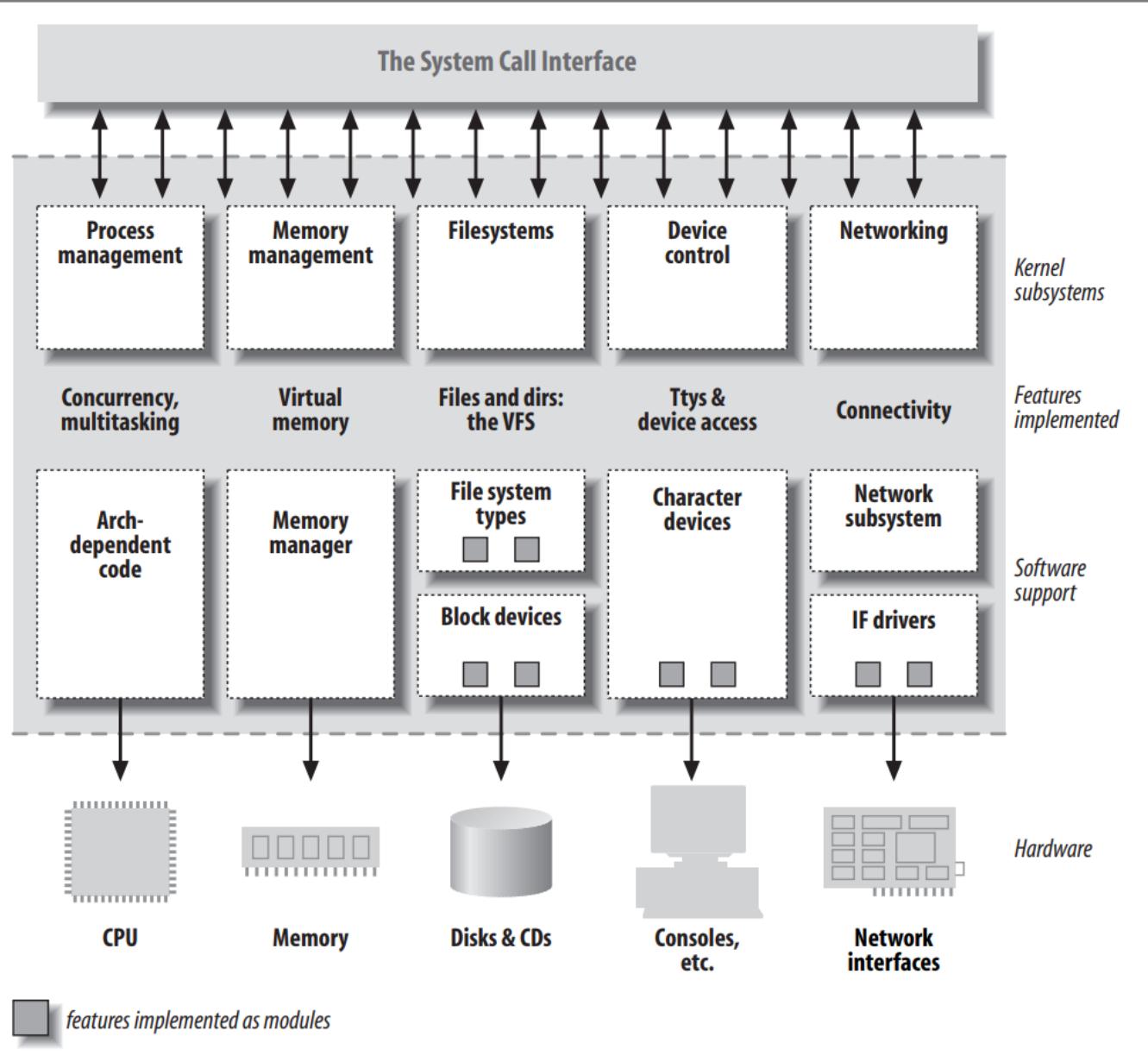
Block drivers

- Block devices: Giống với Character devices, trừ cách dữ liệu được quản lý bên trong bởi Kernel => interface khác nhau
- Vd: disk
- Filesystem driver: giải mã các định dạng khác nhau trên các phân vùng lưu trữ khác nhau (FAT, ext, ..)
- Block device driver: cho các giao thức phần cứng sứng với các thiết bị lưu trữ khác nhau (IDE, SCSI, MTD, ..)

Network Drivers

- Network interfaces: Thay vì đọc ghi như Char hay Block, gọi các hàm liên quan đến truyền gói
- Network driver:
 - Network protocol stack
 - Network interface card (NIC)

Device drivers



4.2. Kernel Modules

- Modules ở tầng nhân (kernel) của OS
- Các drivers được xây dựng sử dụng cơ chế Kernel modules

Kernel Module

- Hoạt động trên Kernel Space, có thể truy xuất tới các tài nguyên của hệ thống
- Kernel Module cho phép thêm mới các module một cách linh hoạt, tránh việc phải biên dịch lại nhân hệ điều hành
- Kernel Module là cơ chế hữu hiệu để phát triển các device driver
- Xem danh sách các module đang chạy: **lsmod**

Kernel Module

- Các bước để thêm một kernel module vào hệ thống
 - Viết mã nguồn: chỉ sử dụng các thư viện được cung cấp bởi kernel, không sử dụng được các thư viện bên ngoài
 - Biên dịch mã nguồn module
 - Cài đặt module: dùng lệnh
insmod Tên_Module.ko
 - Gỡ module: dùng lệnh **rmmod Tên_Module**
 - Xem các thông tin log: sử dụng System Log Viewer

Xây dựng Hello Kernel module

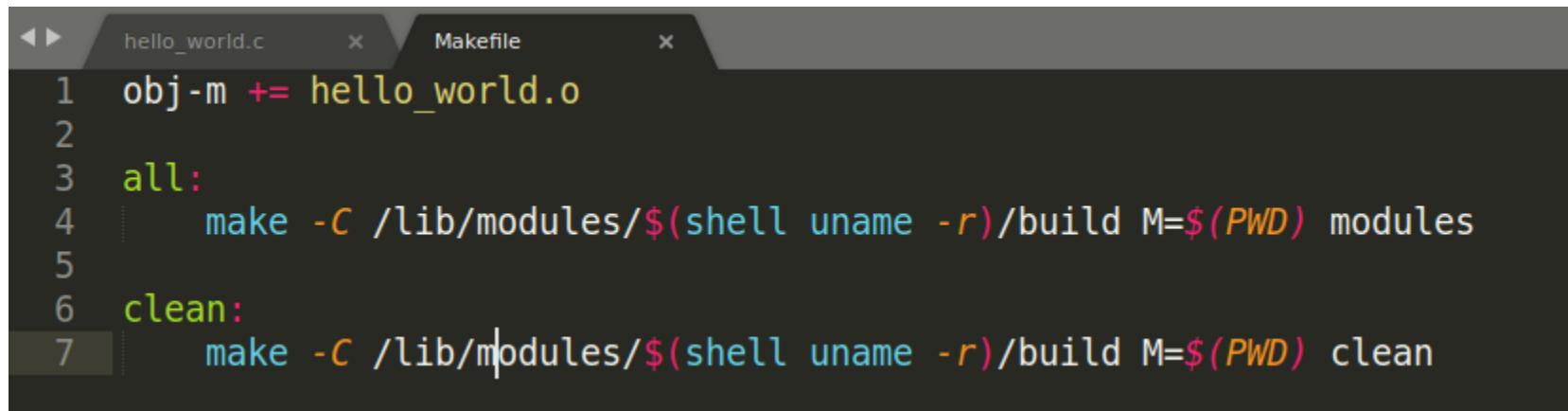
```
//File: hello_kernel_module.c
#include<linux/kernel.h>
#include<linux/init.h>
#include<linux/module.h>
static int hello_init(void)
{
    printk(KERN_ALERT "Khoi tao thanh cong\n");
return 0;
}
static void hello_exit(void)
{
    printk(KERN_ALERT "Ket thuc thanh cong\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

Xây dựng Hello kernel module

- Hàm khởi tạo cần có chỉ thị `_init()` và đăng kí sử dụng macro `module _init()` để giúp xác định hàm sẽ được thực thi ngay sau khi lắp module vào kernel.
- Hàm hủy cần có chỉ thị `_exit()` và đăng kí sử dụng macro `module _ exit()` để giúp xác định hàm sẽ được thực thi ngay trước khi gỡ module vào kernel.

Xây dựng Hello kernel module

- Biên dịch dùng Makefile
- Thực hiện việc đăng ký thư viện biên dịch ở tầng nhân, và đăng ký driver đang cần biên dịch.



```
hello_world.c      Makefile
1 obj-m += hello_world.o
2
3 all:
4     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
5
6 clean:
7     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Kernel Module Makefile

- Viết Makefile biên dịch kernel module

```
obj-m += hello_kernel_module.o
```

all:

```
make -C /lib/modules/$(shell uname -r)/build M=$(PWD)  
modules
```

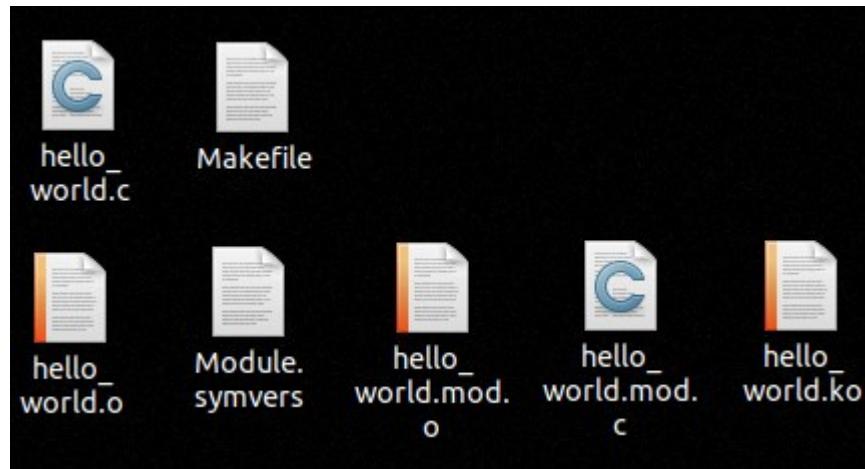
clean:

```
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

- Biên dịch bằng lệnh make được file
hello_kernel_module.ko

Biên dịch Hello kernel module

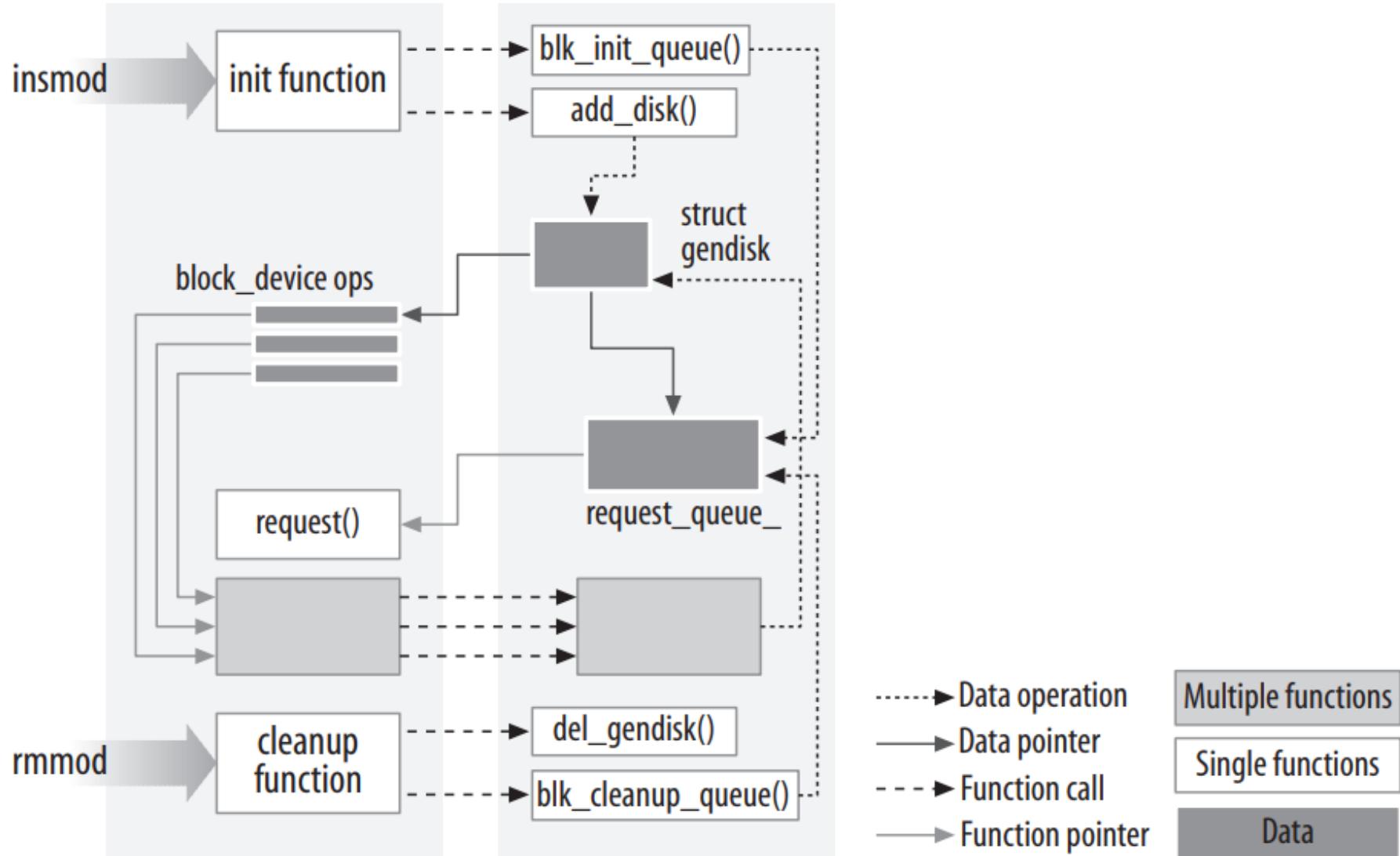
```
f4k3r@pc:~/Desktop$ make
make -C /lib/modules/4.18.0-20-generic/build M=/home/f4k3r/Desktop modules
make[1]: Entering directory '/usr/src/linux-headers-4.18.0-20-generic'
  CC [M]  /home/f4k3r/Desktop/hello_world.o
Building modules, stage 2.
MODPOST 1 modules
WARNING: modpost: missing MODULE_LICENSE() in /home/f4k3r/Desktop/hello_world.o
see include/linux/module.h for more information
  LD [M]  /home/f4k3r/Desktop/hello_world.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.18.0-20-generic'
f4k3r@pc:~/Desktop$ modinfo hello_world.ko
filename:      /home/f4k3r/Desktop/hello_world.ko
srcversion:     2D94C20842ADC6DB0D42D0D
depends:
retpoline:      Y
name:          hello_world
vermagic:      4.18.0-20-generic SMP mod_unload
abi_version:   1
signer:        (none)
sig_key:       (none)
```



Cài đặt/gỡ bỏ kernel module

| insmod/rmmod | modprobe/modprobe -r |
|---------------------|---|
| Nạp/gỡ từng module | Kiểm tra xem module có các module phụ thuộc không và kích hoạt insmod lắp/xóa các module phụ thuộc vào trước rồi đến các module cần thiết |

Cài đặt/gỡ bỏ kernel module



Ví dụ với Hello kernel module

```
f4k3r@pc:~/Desktop$ lsmod | grep hello
f4k3r@pc:~/Desktop$ sudo insmod ./hello_world.ko
[sudo] password for f4k3r:
f4k3r@pc:~/Desktop$ lsmod | grep hello
hello_world               16384  0
```

```
f4k3r@pc:~/Desktop$ cat /var/log/syslog
May 24 00:08:41 pc rsyslogd: [origin software="rsyslogd" swVersion="8.32.0" x-pi
d="1159" x-info="http://www.rsyslog.com"] rsyslogd was HUPed
May 24 00:09:12 pc anacron[2565]: Job `cron.daily' terminated
May 24 00:09:12 pc anacron[2565]: Normal exit (1 job run)
May 24 00:15:12 pc org.gnome.Shell.desktop[2252]: [3778:3778:0524/001512.409597:E
PPNP+buffer manager cc(488)] [DisplayCompositor]GL ERROR - GL TNVAL TD OPERATTION -
May 24 09:00:20 pc kernel: [ 3264.131720] Disabling lock debugging due to kernel
taint
May 24 09:00:20 pc kernel: [ 3264.132215] Hello, World!
May 24 09:00:53 pc gnome-shell[2333]: Some code accessed the property 'discreteGp
uAvailable' on the module 'appDisplay'. That property was defined with 'let' or '
```

```
f4k3r@pc:~/Desktop$ dmesg
[    0.000000] microcode: microcode updated early to revision 0x27, date = 2019-0
2-26
[    0.000000] Linux version 4.18.0-20-generic (buildd@lcy01-amd64-020) (gcc vers
ion 7.3.0 (Ubuntu 7.3.0-16ubuntu3)) #21~18.04.1-Ubuntu SMP Wed May 8 08:43:37 UTC
2019 (Ubuntu 4.18.0-20.21~18.04.1-generic 4.18.20)
[    0.000000] Command line: BOOT_IMAGE=/vmlinuz-4.18.0-20-generic root=UUID=babe
198.800047] /dev/vmmon[3325]: HV      IPI vector: f2
[ 3264.131717] hello_world: module license 'unspecified' taints kernel.
[ 3264.131720] Disabling lock debugging due to kernel taint
[ 3264.132215] Hello, World!
```

Ví dụ với Hello kernel module

- Lệnh insmod: Cài đặt module vào hệ thống (runtime)

```
insmod hello_kernel_module .ko
```

- Gỡ kernel module dùng lệnh rmmod
- Trên Ubuntu để kiểm tra quá trình cài đặt xem module đã cài đặt thành công hay chưa mở System/Log Viewer xem mục kernel log)

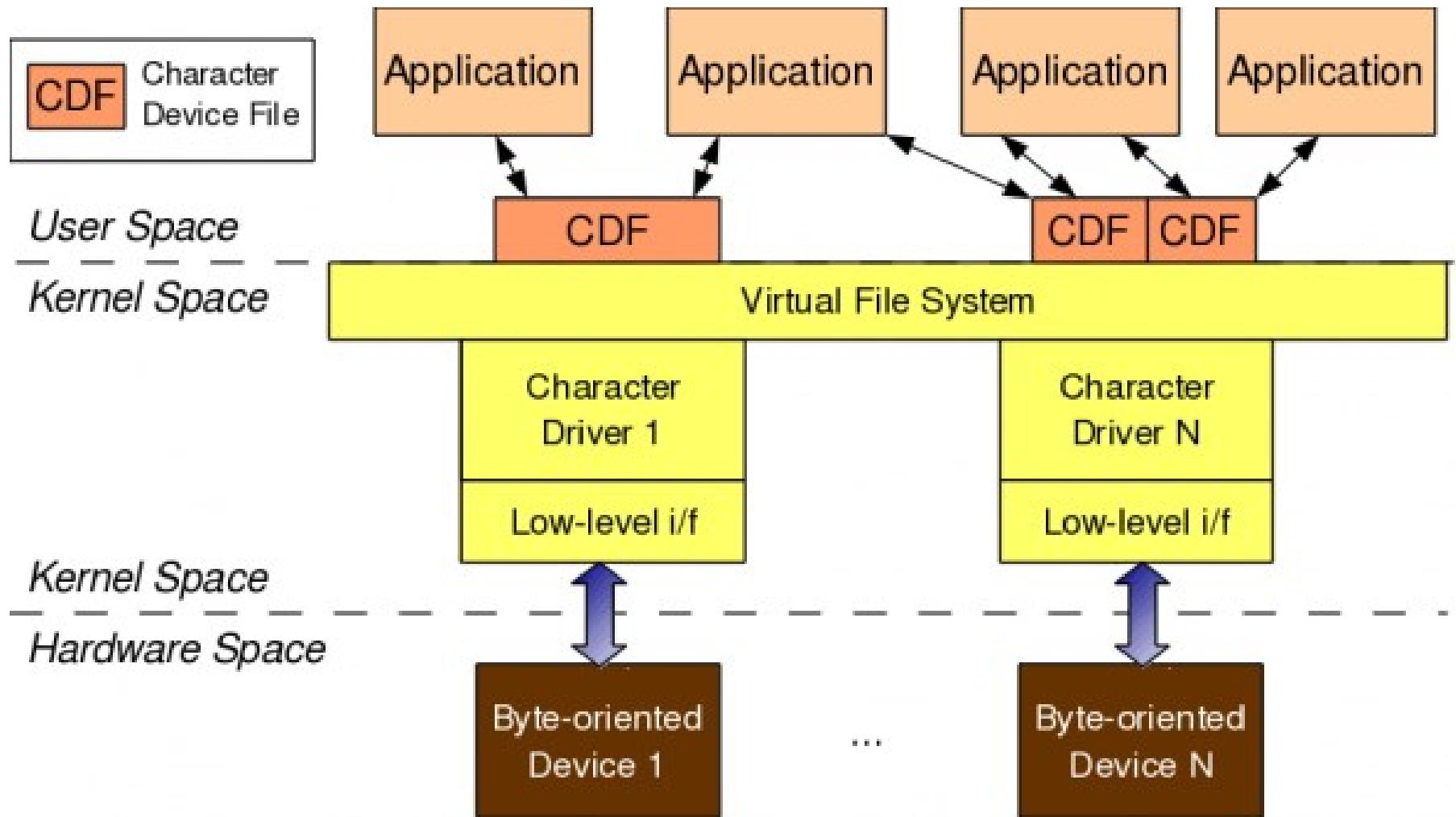
4.3. Linux Character Drivers

- 4.3.1. Cơ chế hoạt động
- 4.3.2. Các thao tác với driver (driver operations)

4.3.1. Cơ chế hoạt động

- Hầu hết các thiết bị (device) đều thuộc kiểu thiết bị hướng byte (byte-oriented). Do vậy, hầu hết các device drivers là các character device drivers.
- Thực tế các storage drivers và network drivers cũng là các dạng của một character driver.
- Ví dụ: serial driver, audio driver, video driver, camera driver, các IO drivers.

Cơ chế hoạt động



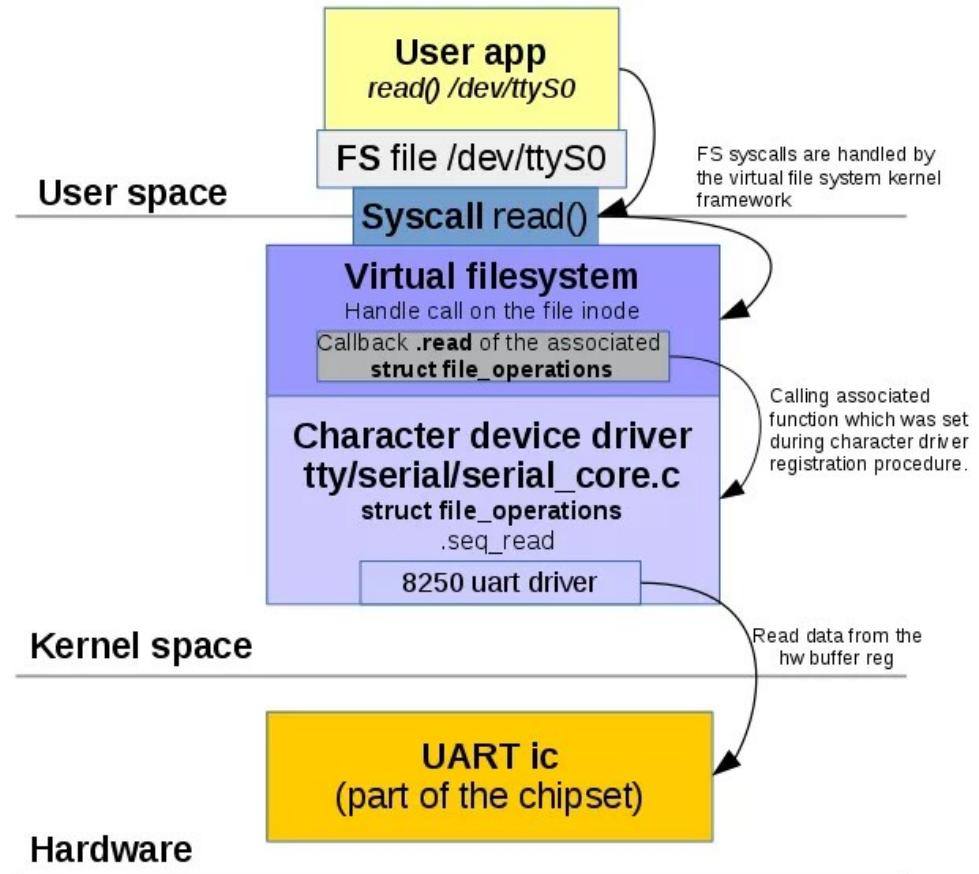
Cơ chế hoạt động

- Việc sử dụng các character driver được thực hiện thông qua các file thiết bị (device files) tương ứng, được liên kết với driver thông qua hệ thống file ảo (virtual file system – VFS).
- Các ứng dụng có thể thực hiện các thao tác file thông thường (mở, đọc, ghi, đóng) trên các file thiết bị.
- Các thao tác file này sẽ được VFS diễn giải ra các hàm tương ứng trong driver liên kết với nó. Các hàm này chứa các chỉ dẫn thực hiện các truy cập ở mức thấp đến các thiết bị (device) thật sự.

Cơ chế hoạt động

Kết nối từ ứng dụng đến thiết bị thông qua 4 bước

1. Application (ứng dụng)
2. Character device file (file thiết bị)
3. Character device driver (driver thiết bị)
4. Character device (thiết bị)



Major và Minor numbers

Số hiệu file thiết bị

- Ngầm định các character device file (CDF) được đặt trong thư mục /dev
- Ứng dụng nhìn nhận và kết nối với CDF thông qua tên file thông thường
- Driver nhìn nhận và kết nối với các CDF thông qua một cặp số hiệu <major, minor>, gọi là số hiệu file thiết bị, chính là định danh cho file thiết bị đó.

```
f4k3r@pc:~/Desktop$ ls -l /dev/ | grep "^\c"
crw-r--r-- 1 root root          10, 235 Thg 5 24 08:06 autofs
crw----- 1 root root          10, 234 Thg 5 24 08:06 btrfs-control
crw----- 1 root root           5,   1 Thg 5 24 08:07 console
crw----- 1 root root          10,  59 Thg 5 24 08:06 cpu_dma_latency
crw----- 1 root root          10, 203 Thg 5 24 08:06 cuse
crw----- 1 root root         241,   0 Thg 5 24 08:06 drm_dp_aux0
crw----- 1 root root          10,  61 Thg 5 24 08:06 ecryptfs
crw-rw--- 1 root video          29,   0 Thg 5 24 08:06 fb0
crw----- 1 root root          10,  55 Thg 5 24 08:06 freefall
crw-rw-rw- 1 root root           1,   7 Thg 5 24 08:06 full
crw-rw-rw- 1 root root          10, 229 Thg 5 24 08:06 fuse
crw----- 1 root root         254,   0 Thg 5 24 08:06 gpiochip0
crw----- 1 root root          243,   0 Thg 5 24 08:06 hidraw0
crw----- 1 root root          243,   1 Thg 5 24 08:06 hidraw1
```

Major và Minor numbers

Kết nối giữa file thiết bị và device driver thông qua 2 bước

- 1. Đăng ký số hiệu <major, minor> cho file thiết bị**
- 2. Kết nối các thao tác file thiết bị với các hàm tương ứng trong driver**

Major và Minor numbers

linux/types.h

- `dev_t` chứa tất cả số hiệu Major và Minor

linux/kdev_t.h, macro:

- `MAJOR(dev_t dev)` lấy số hiệu major từ tham số dev
- `MINOR(dev_t dev)` lấy số hiệu minor từ tham số dev
- `MKDEV(int major, int minor)` tạo ra dữ liệu dev từ cặp số hiệu Major và Minor

Major và Minor numbers

- Các hàm định nghĩa trong <linux/fs.h>
- Đăng ký driver:
 - **int register_chrdev_region(dev_t first, unsigned int count, char *name);**
 - Tham số:
 - first: device number mong muốn
 - count: số lượng device numbers (liên tục) mong muốn
 - name: tên của device liên kết với device number trên
- Cấp phát động:
 - **int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);**
- Giải phóng:
 - **void unregister_chrdev_region(dev_t first, unsigned int count);**

Tạo file thiết bị (device file)

- Khi cấp phát tự động sử dụng `alloc_chrdev_region()`, các device file chưa được tạo ra trong `/dev`.
- Người dùng cần thao tác tạo các device file trong `/dev` bằng lệnh.
- Giải pháp: viết script thao tác tạo các device file, có thể tự động gọi thực thi khi hệ thống khởi động.

Tạo file thiết bị (device file)

load.sh

```
#!/bin/sh
module = "test" #module name
device = "test_device" #device file name
/ sbin / insmod . / $module.ko
rm - f / dev / ${ device }[0 - 2]
major = $(awk - v mod = "$device" '$2==mod{print $1}' /
proc / devices)
mknod / dev / ${ device }0 c $major 0
mknod / dev / ${ device }1 c $major 1
mknod / dev / ${ device }2 c $major 2
```

Tạo file thiết bị (device file)

- Có thể tạo file thiết bị tự động (trong code của driver)
- Các API liên quan tới mô hình thiết bị được định nghĩa trong <linux/device.h>

```
struct class * class_create(struct module * owner,  
const char * name);  
struct device * device_create(struct class * class,  
struct device * parent, dev_t devt, const char * fmt,  
...);  
void device_destroy(struct class * class, dev_t devt);  
void class_destroy(struct class * cls);
```

4.3.2. Các thao tác với file thiết bị

- (File Operations)
- Nhắc lại: Các ứng dụng có thể thực hiện các thao tác file thông thường (mở, đọc, ghi, đóng) trên các file thiết bị.
- Các thao tác file này sẽ được VFS diễn giải ra các hàm tương ứng trong driver liên kết với nó. Các hàm này chứa các chỉ dẫn thực hiện các truy cập ở mức thấp đến các thiết bị (device) thật sự.
- Thông thường qua 2 bước:

Các thao tác với file thiết bị

- **Bước 1:** Tạo biến cấu trúc struct file_operations, gán các thao tác tùy chỉnh vào cấu trúc này.
- Các thao tác cơ bản như:
 - open
 - release
 - read
 - write
 - seek
 - ioctl

```
struct file_operations  
mydev_fops =  
{  
    .owner = THIS_MODULE,  
    .open = my_open,  
    .release = my_close,  
    .read = my_read,  
    .write = my_write  
};
```

Các thao tác với file thiết bị

■ Bước 2:

- 1. Khởi tạo biến cấu trúc cdev:

```
void cdev_init(struct cdev *cdev, struct  
file_operations *fops);
```

- 2. Thông báo cho kernel sự hiện diện của character device:

```
int cdev_add(struct cdev *dev, dev_t num,  
unsigned int count);
```

File Operations

- Phương thức open()

```
int (*open)(struct inode *inode, struct file *filp);
```

- Thực hiện các thao tác khởi tạo, chuẩn bị cho những thao tác tiếp theo:

- Kiểm tra lỗi thiết bị (device-not-ready,...).
- Khởi tạo thiết bị nếu được mở lần đầu.
- Cấp phát và khởi tạo các thông tin cần thiết.

File Operations

- Phương thức release():

```
int (*release) (struct inode *, struct file *);
```

- Thực hiện các thao tác dọn dẹp khi kết thúc
 - Giải phóng các vùng nhớ đã xin cấp phát
 - Tắt thiết bị sau lần đóng cuối cùng

File Operations

- Phương thức read() và write()

Trao đổi thông tin giữa ứng dụng (qua bộ đệm buff) với device file.

File Operations

▪ Phương thức read()

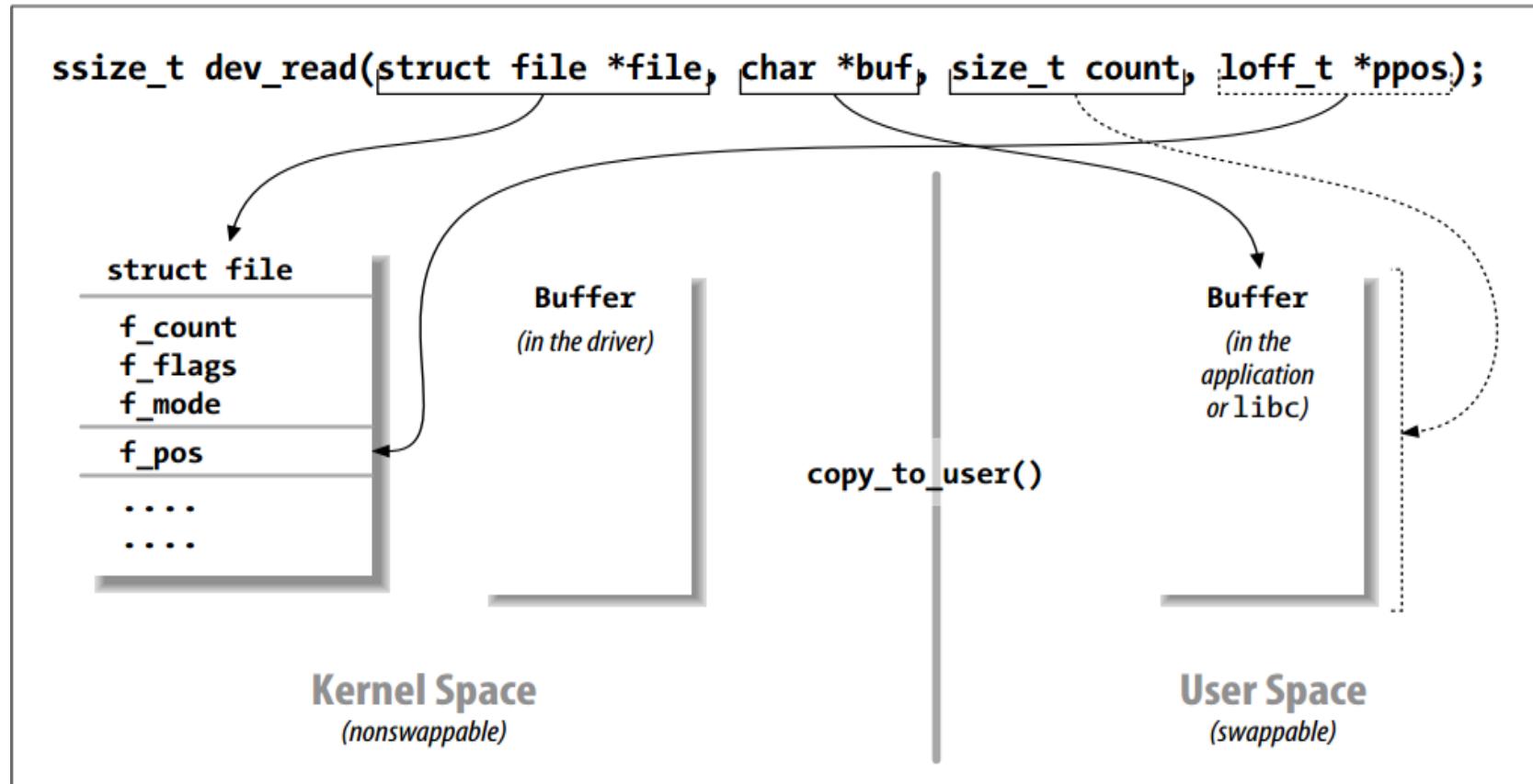
```
ssize_t read(struct file *filp, char __user  
            *buff, size_t count, loff_t *offp);
```

Trả về:

- Thông thường, trả về số byte đọc được (có thể < số byte yêu cầu)
- 0 nếu gặp kết thúc file (EOF)
- <0 nếu xảy ra lỗi

File Operations

▪ Phương thức read()



File Operations

▪ Phương thức write():

```
ssize_t write( struct file *filp, const char __user *buff, size_t  
count, loff_t *offp);
```

Trả về:

- Thông thường, trả về số byte đã ghi thành công (có thể < số byte yêu cầu)
- 0 nếu chưa có dữ liệu được ghi, thao tác được xem xét thực hiện sau
- <0 nếu xảy ra lỗi

File Operations

- Giao tiếp với user space trong phương thức read() và write():
 - Tham số buff trong 2 phương thức trên chứa địa chỉ ở user-space, vì vậy không nên trực tiếp truy cập thông qua toán hạng dereferenced (*).
 - Để an toàn hơn, sử dụng các API sau:

```
unsigned long copy_to_user(void __user *to, const void *from, unsigned long count);
```

```
unsigned long copy_from_user(void *to, const void __user *from, unsigned long count);
```

File Operations

■ Các phương thức khác:

- Driver có thể triển khai các phương thức khác, cung cấp các tính năng khác nhau cho tầng application.
- Ngoài các thao tác chung nói trên, driver có thể cung cấp thêm các thao tác giống với các thao tác với file thông thường: ioctl, llseek, poll (nonblocking IO),..
- int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
- loff_t (*llseek) (struct file *, loff_t, int);
- int (*flush) (struct file *);
- unsigned int (*poll) (struct file *, struct poll_table_struct *);

Tóm tắt lập trình linux device driver

- Sử dụng cơ chế lập trình kernel module để xây dựng các device driver
- Qui trình xây dựng device driver:
 - Viết mã nguồn (cấu trúc tương tự kernel module).
 - Biên dịch mã nguồn (dùng Makefile) (được file .ko)
 - Cài đặt sử dụng lệnh **insmod**
 - Đăng ký tên file thiết bị (device file) với hệ thống
 - Đăng ký tự động trong mã nguồn
 - Hoặc sử dụng lệnh **mknod** để tạo device file (trong thư mục /dev)
mknod [options] NAME Type [Major Minor]

Mã nguồn hello_driver.c

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
static ssize_t hello_read(struct file *filp, char *buffer, size_t length, loff_t * offset);
static int Major;
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .read = hello_read,
};
static int hello_init(void)
{
    Major = register_chrdev(0, "KTMT_Device", &fops);
    if (Major < 0) {
        printk(KERN_ALERT "Dang ky that bai, Major id=%d\n", Major);
        return Major;
    }
    else{
        printk(KERN_ALERT "Dang ky thiet bi thanh cong, Major id=%d\n", Major);
    }
    return 0;
}
static void hello_exit(void)
{
    unregister_chrdev(Major, "KTMT_Device" );
}
static ssize_t hello_read(struct file *filp, char *buffer, size_t length, loff_t * offset)
{
    return 100;
}
module_init(hello_init);
module_exit(hello_exit);
```

Tham khảo + Demo

<https://sites.google.com/site/embedded247/ddcourse/kernelmoduleprogramming>



Demo

```
/* Necessary includes for device drivers */
#include <linux/module.h>
#include <linux/kernel.h>      /* printk() */
#include <linux/fs.h>          /* everything ...*/
#include <linux/init.h>
#include <linux/slab.h> /* kmalloc() */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h> /* O_ACCMODE */
#include <linux/cdev.h>
#include <linux/miscdevice.h>
#include <asm/uaccess.h> /*copy_from/to_user() */

#define DEVICE_NAME "hello_device"
//static int Major;
static volatile char buff_values[] = { 0,0,0,0,0,0 };
static struct file_operations dev_fops = {
    .owner = THIS_MODULE,
    .read = hello_read,
};
static struct miscdevice misc = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DEVICE_NAME,
    .fops = &dev_fops,
};
```

```
static ssize_t hello_read(struct file *filp, char *buffer, size_t length, loff_t *offset);

static int hello_init(void)
{
    //Major=register_chrdev(0,"KTMT-Device",&fops);
    int ret;
    ret = misc_register(&misc);
    if (ret < 0)
    {
        printk(KERN_ALERT "Dang ky that bai\n");
    }
    else {
        printk(KERN_ALERT "Dang ky thiet bi thanh cong\n");
    }
    return ret;
}
static void hello_exit(void)
{
    //unregister_chrdev(Major,"KTMT-Device");
    misc_deregister(&misc);
    printk(KERN_ALERT "Thiet bi da bi ngat khoi he thong\n");
}
```

```
static ssize_t hello_read(struct file *filp, char *buffer,
size_t length, loff_t *offset)
{
    int i;
    unsigned long err;
    for (i = 0; i < 6; i++)
        buff_values[i] = 'A' + i;
    err = copy_to_user(buffer, (const void*)buff_values,
min(sizeof(buff_values), length));
    return 100;
}
module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("PNH");
```

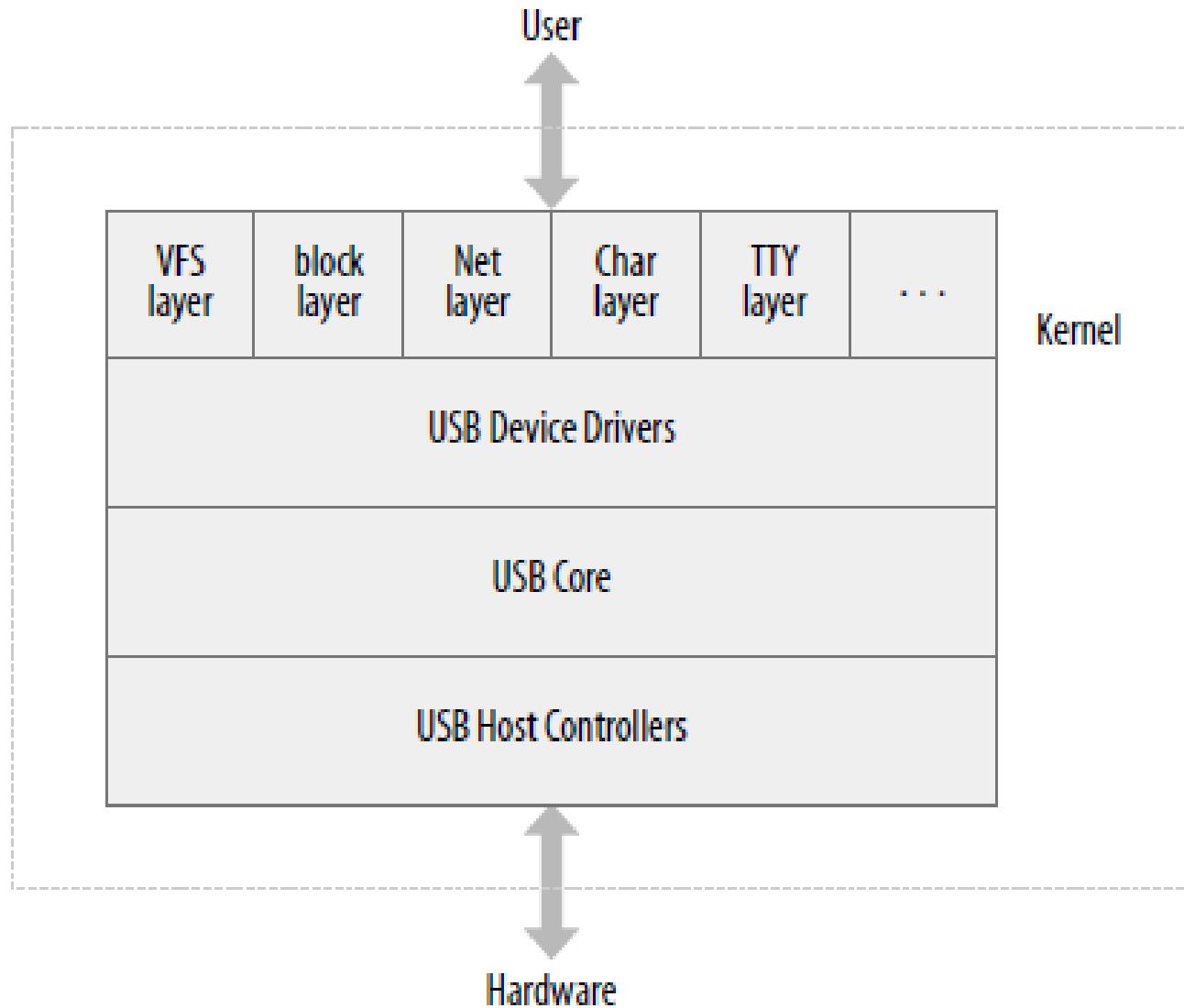
4.4. Tìm hiểu USB Device Drivers

- Tổng quan
- Mô hình thiết bị USB
- Viết USB Device Driver đầu tiên

Tổng quan USB drivers

- USB host controller (là thiết bị chủ động đường bus của giao thức usb) thực hiện phát hiện thiết bị usb
- USB host controller driver này sẽ thu thập và diễn giải các thông tin ở tầng vật lý (low-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) (tương ứng VFS char driver) trong tầng nhân (được điều khiển bởi usbcore driver). Lúc này tầng nhân đã nhận diện được thiết bị usb
- Usb Device driver sẽ có đóng vai trò trung gian tiếp tục nhận dạng ra thiết bị để phục vụ cho ứng dụng thiết bị ở tầng người dùng (user space).

Tổng quan USB drivers



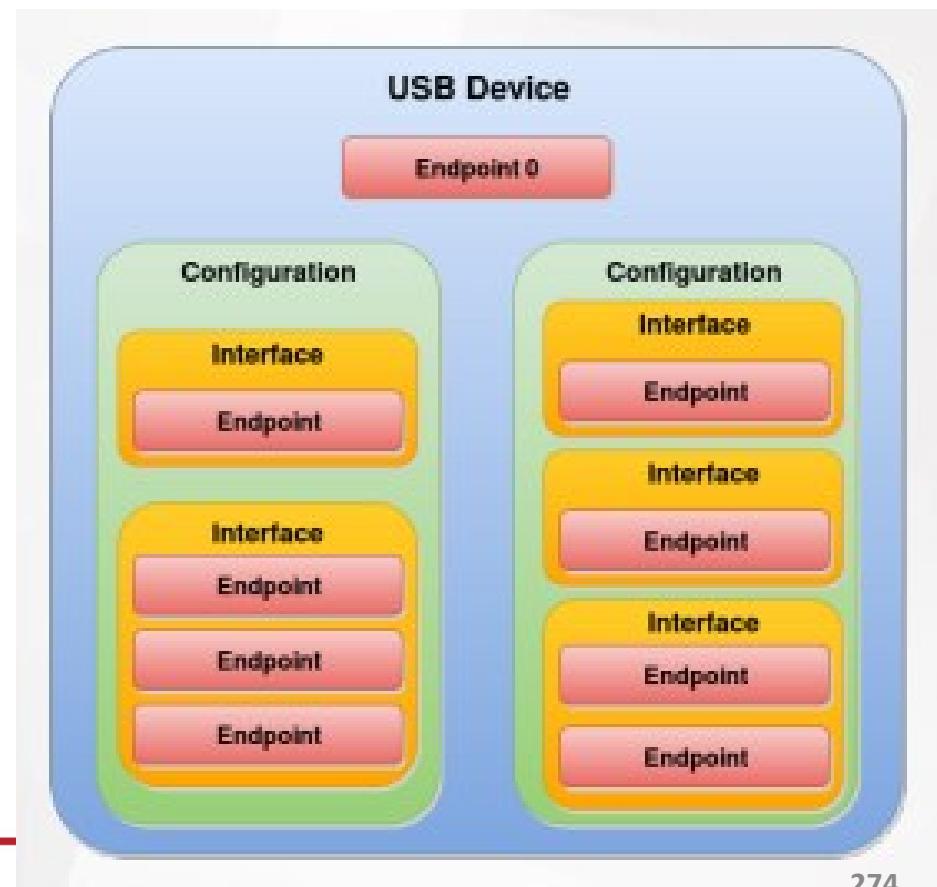
Mô hình thiết bị USB

- Tất cả các USB device đều có 1 profile gồm configuration.
- Configuration bao gồm các interface trong các device.
- Interfaces cung cấp các chức năng của device.

Ví dụ: device có thể là một printer, fax, scanner.

Mỗi chức năng như thế được đại diện bởi 1 interface.

Và mỗi interface được cấu thành từ một hay nhiều endpoint.



Mô hình thiết bị USB

ENDPOINT

- Số lượng có thể lên tới tối đa 32 endpoint (mặc định endpoint 0)
- Mỗi một endpoint có một address khác nhau
- Endpoint 0 có thể truyền dữ liệu theo cả 2 chiều
- Tất cả các endpoint có thể truyền dữ liệu 1 trong 2 chiều:
 - IN: Truyền dữ liệu từ device tới host
 - OUT: Truyền dữ liệu từ host tới device

Mô hình thiết bị USB

- Endpoint thiết bị có thể thuộc 1 trong 4 loại sau:
 - *Control*: dùng để truyền các thông tin điều khiển.
 - *Interrupt*: dùng để truyền một lượng thông tin nhỏ và nhanh
 - *Bulk*: dùng để truyền một lượng lớn dữ liệu, đảm bảo tính toàn vẹn dữ liệu.
 - *Isochronous*: dùng để truyền một lượng lớn dữ liệu, tốc độ cao, không cần đảm bảo tính toàn vẹn dữ liệu.

Viết một usb-pen driver

- Bắt đầu với một chương trình cơ bản

```
#include <linux/module.h>
#include <linux/kernel.h>
static int __init pen_init(void)
{
int ret = -1;
return ret;
}
static void __exit pen_exit(void)
{
}
module_init(pen_init);
module_exit(pen_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("USB Pen Registration Driver");
```

Viết một usb-pen driver

■ #include <linux/usb.h>

- Thư viện chứa các struct, attribute, function cần thiết cho một usb driver

Viết một usb-pen driver

■ Cấu trúc usb driver

```
static struct usb_driver pen_driver =  
{  
    .name = "Usb Stick-driver"  
    .id_table = pen_table,  
    .probe = pen_probe,  
    .disconnect = pen_disconnect,  
}
```

Viết một usb-pen driver

■ .id_table

```
const struct usb_device_id *id_table
static struct usb_device_id pen_table[] =
{
    { USB_DEVICE(0x0781, 0x5591) },
    {} /* Terminating module*/
}
```

Viết một usb-pen driver

- Hàm probe() thăm dò thiết bị

```
static int pen_probe(struct usb_interface *interface, const
                     struct usb_device_id *id)
{
    printk(KERN_INFO "Driver for device (%04X:%04X)
plugged\n", id->idVendor, id->idProduct);
    return 0;
}
```

Viết một usb-pen driver

- Hàm probe():
 - Được gọi khi một thiết bị được kết nối (cắm vào cổng usb) và usb_core nghĩ rằng có thể handle thiết bị bằng driver này.
 - probe() kiểm tra thông tin của thiết bị và quyết định thiết bị có thực sự được sử dụng bởi driver này hay không
 - Trường hợp một driver khác đã được sử dụng cho thiết bị này rồi thì hàm probe() sẽ không được gọi.

Viết một usb-pen driver

■ Hàm disconnect

```
static void pen_disconnect(struct usb_interface *  
interface)  
{  
    printk(KERN_INFO "Pen driver removed\n");  
    //do something clean and fragrant here!  
}
```

Viết một usb-pen driver

- Register driver to usb_core

```
static int __init pen_init(void) {  
    int ret = -1;  
    printk(KERN_INFO "Constructor of driver");  
    printk(KERN_INFO "\t Registering driver with Kernel");  
    ret = usb_register(&pen_driver);  
    printk(KERN_INFO "\tRegistration is complete");  
    return ret;  
}
```

Viết một usb-pen driver

- Deregister driver

```
static void __exit pen_exit(void)
{
    printk(KERN_INFO "Destructor of driver");
    usb_deregister(&pen_driver);
    printk(KERN_INFO "unregistration is complete!");
}
```

Viết một usb-pen driver

- Demo



Demo



25
YEARS ANNIVERSARY
SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

