

设备管理

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

<https://ipads.se.sjtu.edu.cn>

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
 - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
 - 完整文本：<https://creativecommons.org/licenses/by/4.0/legalcode>

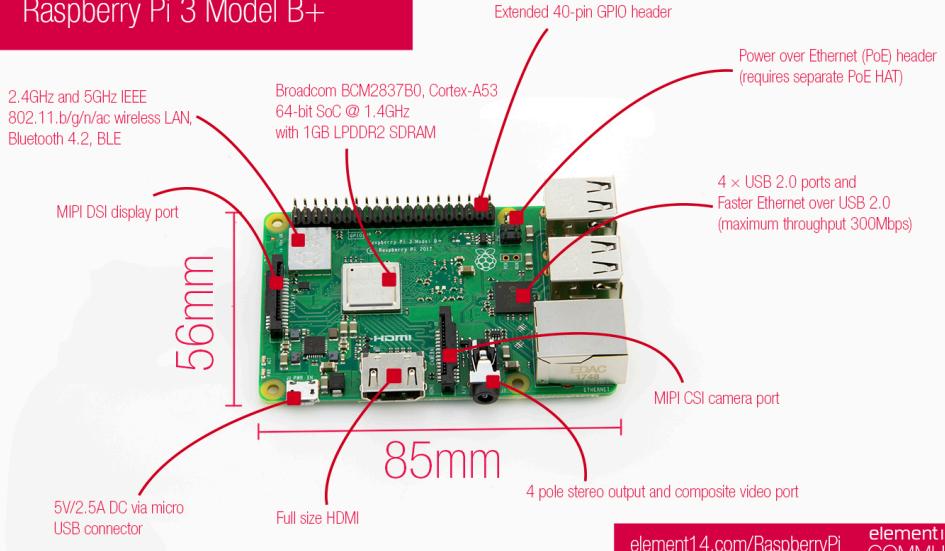
回顾：操作系统的管理/服务

- **根据不同需求和场景，人们发明了大量专用设备**
 - 通信、存储、智能计算、安全协处理器等
- **每种设备有自己的协议、规范**
 - 如何标准化外设接口？
- **设备也可能产生错误**
 - 如何得知外设的状态并修复错误？



案例：树莓派上的外设与接口

Raspberry Pi 3 Model B+



- 种类多 ☺
- 功能不同
- 接口、协议、规范不同 ☹



操作系统的担当：
把复杂留给自己，把简单留给他

如何解决这个问题呢？

- 理解设备的用途→找出共同点（分类）→总结抽象
- 理解设备是第一步（：）
- **What I cannot create, I do not understand!**



从认知外设开始

认知外设

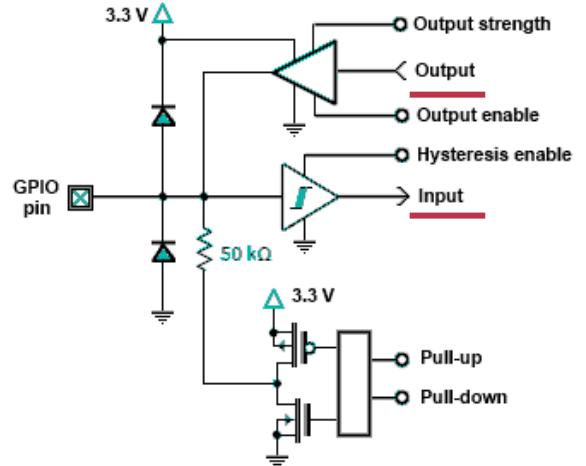
- 设备的基本用途（到底是干什么的）
- 是否具备某些共同点（能否分门别类）
- 如何驱使它们进行工作（设备的可编程接口长什么样）

GPIO LED

- 有专门的 INPUT/OUTPUT 管脚
- 通过管脚进行控制
- 每个01组合只显示一种状态



Equivalent Circuit for Raspberry Pi GPIO pins



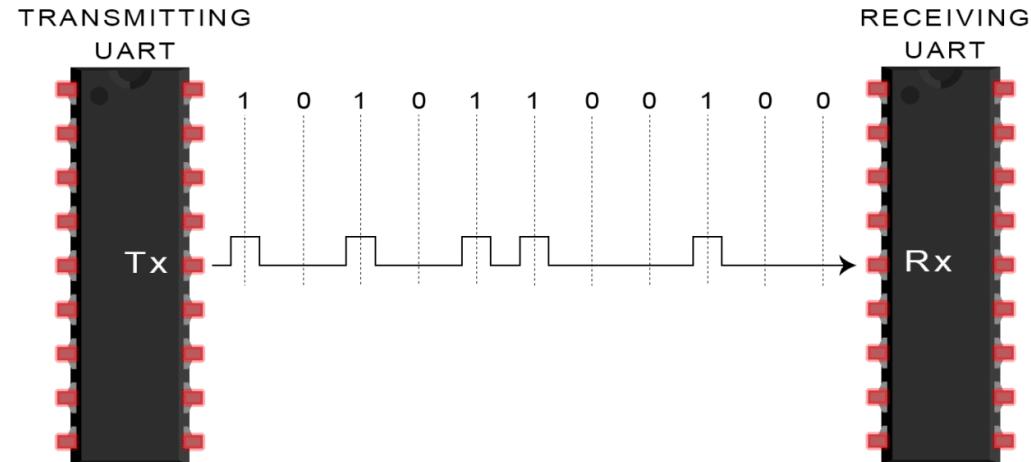
Intel 8042 (PS/2 键盘控制器)

- 电信号→数字信号→编码 (Scan Code)
 - 每次只能键入一个字符

思考：如果长摁呢？

UART (串口)

- 通用异步收发传输器 (Universal Asynchronous Receiver/Transmitter)
- 半双工
- 每次只能传输一个字符



总结

- **GPIO LED**
 - 每个01组合只显示一种状态
- **i8042 PS/2**
 - 每次只能键入一个字符
- **UART**
 - 每次只能传输一个字符
- **特点：时间串行性**

Flash闪存

- 按照页/块的粒度进行读写/擦除
- 支持页/块随机访问
- 特点：**空间随机性**



Ethernet网卡

- 每次传输一块数据（以太网帧）
- 特点：时间上串行、空间上随机
- Wifi、蓝牙与此类似



其实设备还有很多.....

- 传感器
- 陀螺仪
- 磁力计
- 协处理器 (GPU、TPU)



解决问题的有力武器 → 抽象

设备管理：复用文件系统抽象

- **抽象的作用**
 - 操作系统将外设细节和协议封装在接口的内部
- **如下示例代码可以运行在不同设备上**
 - 为应用程序提供的相同的抽象接口（文件接口）

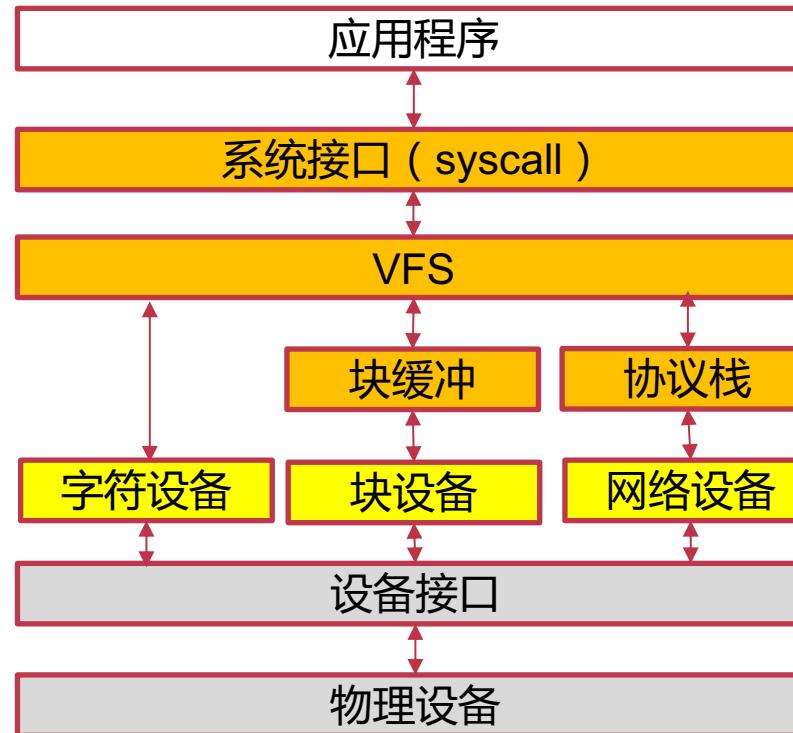
```
char buffer[256];
int read_num = -1;
int fd = open("/dev/something", O_RDWR);
write(fd, "something to device", 19);
while (read_num == -1)
    read(fd, buffer, 256);
close(fd);
```

设备抽象（以Linux为例）

- 对设备进行分类
 - 字符设备 (char) : LED、键盘、串口
 - 块设备 (block) : 闪存
 - 网络设备 (network) : Ethernet网卡
- 对设备进行管理
 - 字符抽象 : fs (read/write)
 - 块抽象 : fs (read/write) , mmap
 - 网络抽象 : socket , fs-compatible (用read/write使用socket)

案例：Linux常见设备分类

- **字符设备**
- **块设备**
- **网络设备**



字符设备

- **例子：**
 - 键盘、鼠标、串口、打印机等
- **访问模式：**
 - 串行访问，每次读取一个字符
 - 调用驱动程序和设备直接交互
- **通常使用文件抽象：**
 - `open()`, `read()`, `write()`, `close()`

块设备

- **例子：**
 - 磁盘、U盘、闪存等（以存储设备为主）
- **访问模式：**
 - 以块粒度进行读写，随机访问
 - 在系统层增加一层缓冲，避免和慢设备频繁交互
- **通常使用内存抽象：**
 - 内存映射文件(Memory-Mapped File)：直接访问数据
 - 同样可以使用文件抽象，但内存抽象更受欢迎（灵活性更好）

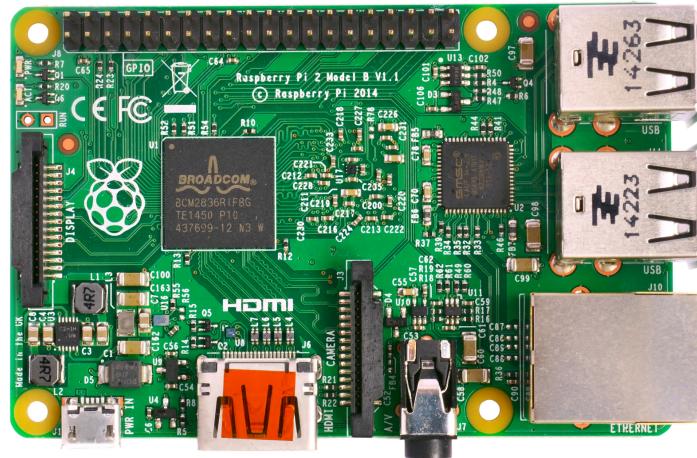
网络设备

- **例子：**
 - 以太网、WiFi、蓝牙等（以通信设备为主）
- **访问模式：**
 - 面向格式化报文的收发
 - 在驱动层以上维护多种协议，支持不同策略
- **通常使用套接字抽象：**
 - `socket()`, `send()`, `recv()`, `close()`, etc

思考题

- 树莓派上的这些外设属于什么设备？

- UART串口
- SD卡
- RTC实时时钟
- DS18B20温度传感器
- USB
- 以太网
- HDMI
- GPIO
- CSI摄像头
- 板载无线蓝牙

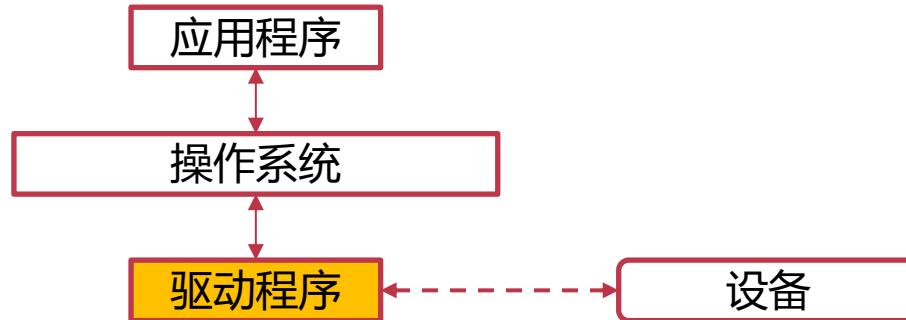


思考题

- 新型设备属于什么类型的设备？
 - GPU
 - Smart NIC
 - NVRAM
- Linux的设备分类是否过时？
 - 或过于简单？

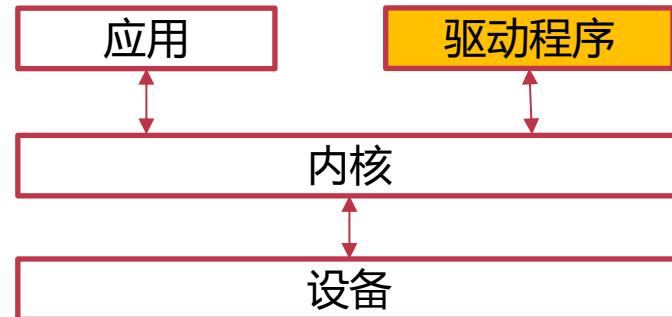
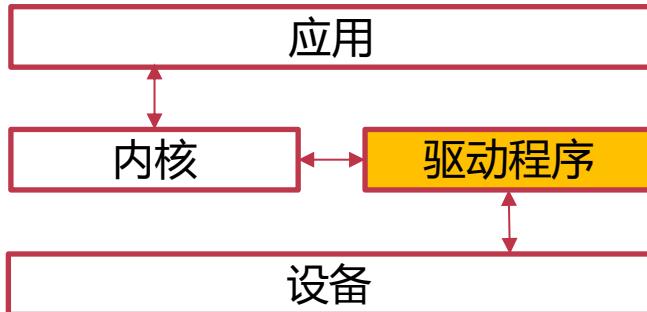
设备的代码——驱动

- 驱动
 - 使操作系统和设备间能相互通信的特殊程序
- 例子：操作系统——CPU的“驱动”



宏内核vs微内核的驱动

- 宏内核
 - 驱动在内核态
 - 优势：性能更好
 - 劣势：容错性差
- 微内核
 - 驱动在用户态
 - 优势：可靠性好
 - 劣势：性能开销（IPC）



驱动程序



- **系统接口 (ioctl) :**
 - 让用户空间的应用通过驱动间接和设备进行交互
- **驱动模型 :**
 - 让驱动程序可以在系统统一安排下和设备交互

问题：为什么需要设备驱动模型？

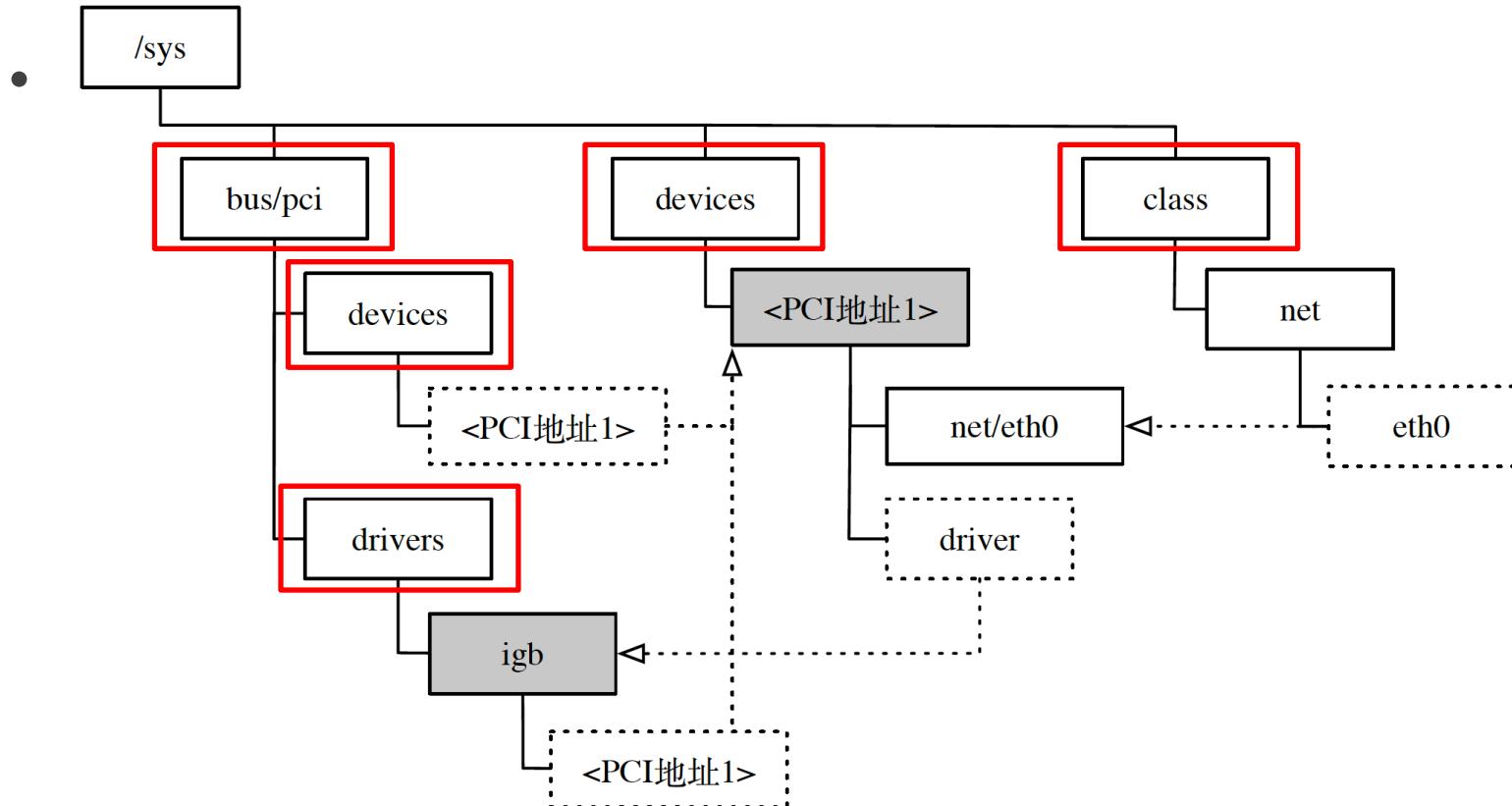
为什么需要驱动模型

- Linux在2.4以前没有驱动模型（一样可以用）
- 设备的整体趋势：
 - 数量和规模越来越大
 - 更新速度越来越快：驱动代码量在快速增长
- 驱动开发者的要求：
 - 标准化的数据结构和接口
 - 将驱动开发简化为对数据结构的填充和实现

Linux Device Driver Model (LDDM)

- 支持电源管理与设备的热拔插
- 利用sysfs向用户空间提供系统信息
- 维护驱动对象的依赖关系与生命周期，简化开发工作
 - 驱动人员只需告诉内核对象间的依赖关系
 - LDDM启动设备会将依赖对象自动初始化，直到启动条件满足为止

sysfs



Linux设备驱动抽象

- **Device (设备)** : 用于抽象系统中所有的硬件
 - 包括CPU和内存
- **Bus (总线)** : CPU连接Device的通道
 - 所有的Device都通过bus相连
- **Class (分类)** : 具有相似功能或属性的设备集合
 - 类似面向对象程序设计中的Class
 - 抽象出一套可以在多个设备之间共享的数据结构和接口
 - 从属于相同Class的设备驱动程序，直接继承

Linux设备驱动抽象数据结构

- 对设备连接关系进行抽象
 - struct bus_type
 - 表示怎么连
- 对硬件设备进行抽象
 - struct device
 - 表示有什么
- 对设备驱动进行抽象
 - struct device_driver
 - 表示怎么用

struct bus_type

```
struct bus_type {  
    const char *name; /* 总线名称：PCI、USB、I2C、SPI */  
    const struct attribute_group **bus_groups; /* 总线属性 */  
  
    int (*match)(struct device *dev, struct device_driver *drv); /* 添加设备  
或驱动到该总线时被调用 */  
    int (*uevent)(struct device *dev, struct kobj_uevent_env *env); /* 任何属  
于该bus的device发生变化时被调用 */  
    int (*probe)(struct device *dev); /* 初始化，保证device所属的bus已被初始化  
*/  
    int (*remove)(struct device *dev); /* 移除函数 */  
  
    /* IOMMU相关操作 */  
    const struct iommu_ops *iommu_ops;  
};
```

总线的注册：int **bus_register**(struct bus_type *bus)
==> 若注册成功，新的总线将被添加进系统，可在 /sys/bus 下看到

总线的注销：void **bus_unregister**(struct bus_type *bus)

struct device

```
struct device {  
    struct device *parent; /* 父设备，通常是总线或总控制器 */  
    const char *init_name; /* 设备名 */  
    const struct device_type *type; /* 设备类型：char, block, network */  
    struct bus_type *bus; /* 所属的总线 */  
    struct device_driver *driver; /* 对应的driver */  
    dev_t devt; /* 设备号，对应 sysfs "dev" */  
    void *driver_data; /* 驱动私有数据 */  
    struct dev_pm_info power; /* 电源管理 */  
  
    /* DMA操作函数 */  
    const struct dma_map_ops *dma_ops;  
  
    struct class *class; /* 所属的设备类型集合 */  
    const struct attribute_group **groups; /* 默认attribute集合 */  
};
```

设备的注册：int `device_register(struct device * dev)`

设备的注销：void `device_unregister(struct device * dev)`

struct device_driver

```
struct device_driver {  
    const char          *name; /* 驱动名称 */  
    struct bus_type     *bus;  /* 驱动所属的总线 */  
  
    struct module       *owner;  
    const char          *mod_name; /* Linux内核模块的名称 */  
  
    int (*probe) (struct device *dev); /* 热插：检测到设备 */  
    int (*remove) (struct device *dev); /* 热拔：移除设备 */  
  
    /* 电源管理接口 */  
    void (*shutdown) (struct device *dev);  
    int (*suspend) (struct device *dev, pm_message_t state);  
    int (*resume) (struct device *dev);  
};
```

驱动的注册：int **driver_register**(struct device_driver *drv)

驱动的注销：void **driver_unregister**(struct device_driver *drv)

LDDM特点

- **device_driver和device都注册到bus上**
- **bus_type的match()**
 - 如果设备与驱动相匹配，将调用device_driver的probe()，交给device_driver来完成余下工作
- **device_driver的probe()**
 - 驱动程序的入口，probe成功后内核生成设备实例，驱动注册的file_operations可以被应用程序所访问
- **device为bus_type、device_driver父类的多继承**
 - 要实例化一个device，先实例化父类driver和bus

案例：Linux的内核驱动程序开发

1. 注册总线

```
// 1. 自定义总线类型
struct bus_type my_bus_type = {
    .name = "pci",
};

// 2. 注册该总线类型
int my_bus_init(void)
{
    return bus_register(&my_bus_type);
}

void my_bus_exit(void)
{
    bus_unregister(&my_bus_type);
}

// 3. 将总线类型暴露给其他设备使用
EXPORT_SYMBOL(my_bus_type);

module_init(my_bus_init);
module_exit(my_bus_exit);
```

```
// 4. 查看是否注册成功
# tree -d /sys/bus/pci/
/sys/bus/pci/
|-- devices
`-- drivers
```

2. 注册设备

```
// 1. 引入已注册好的总线类型
extern struct bus_type my_bus_type;

// 2. 自定义设备
struct device my_dev = {
    .init_name = "my_dev",
    .bus = &my_bus_type,
};

// 2. 注册设备
int my_device_init(void)
{
    return device_register(&my_dev);
}

void my_device_exit(void)
{
    device_unregister(&my_dev);
}

module_init(my_device_init);
module_exit(my_device_exit);
```

```
// 3. 查看是否注册成功(样例)
# tree -d /sys/devices/pci0/
/sys/devices/pci0/
|-- 00:00.0
|-- 00:01.0
|   '-- 01:00.0
|-- 00:02.0
|   '-- 02:1f.0
|       '-- 03:00.0
|-- 00:1e.0
|   '-- 04:04.0
|-- 00:1f.0
|-- 00:1f.1
|   |-- ide0
|   |   |-- 0.0
|   |   '-- 0.1
|   '-- ide1
|       '-- 1.0
|-- 00:1f.2
|-- 00:1f.3
`-- 00:1f.5
```

3. 注册驱动

```
// 1. 引入已注册好的总线类型
extern struct bus_type my_bus_type;

// 2. 自定义初始化函数
int my_probe(struct device *dev)
{
    return 0;
}

// 3. 声明驱动
struct device_driver my_driver = {
    .name = "my_dev",
    .bus = &my_bus_type,
    .probe = my_probe,
};

// 4. 注册驱动
int my_driver_init(void)
{
    return driver_register(&my_driver);
}
module_init(my_driver_init);
```

```
// 5. 查看是否注册成功(样例)
# tree -d /sys/bus/pci/drivers/
/sys/bus/pci/drivers/
|-- 3c59x
|-- Ensoniq AudioPCI
|-- agpgart-amdk7
|-- e100
`-- serial
```

ioctl —— 用户空间接口

```
#define LED_ALL_ON      _IO('L',0x1234)
#define LED_ALL_OFF     _IO('L',0x5678)

int main(void)
{
    int fd;

    fd = open("/dev/led", O_RDWR);
    if (fd < 0)
        exit(1);

    while(1) {
        ioctl(fd, LED_ALL_ON);
        sleep(1);
        ioctl(fd, LED_ALL_OFF);
        sleep(1);
    }

    close(fd);
    return 0;
}
```

```
long led_ioctl(..., unsigned int
cmd, ...)
{
    switch (cmd) {
        case LED_ALL_ON:
            *gpc0_data |= 0x3<<3;
            break;
        case LED_ALL_OFF:
            *gpc0_data &= ~0x3<<3;
            break;
        default: break;
    }
    return 0;
}

static struct file_operations fops = {
    .open = led_open,
    .write = led_write,
    .unlocked_ioctl = led_ioctl,
    .release = led_close,
};
```

总结

- LDDM将设备组织成树状结构，用sysfs暴露给用户空间
- 采用面向对象（Object）思想，用继承避免代码冗余

目录	内容
/sys/devices	Linux内核对系统中所有设备的分层次表达模型
/sys/dev	维护字符设备和块设备的主次号码(major:minor) 到真实的设备(/sys/devices下)的符号链接
/sys/bus	Linux内核设备按总线类型分层放置的目录结构
/sys/class	按照设备功能分类的设备模型



设备树

问题：如何刻画一个设备？

- 以串口为例：
 - 名称：uart0
 - 波特率：115200
- 用C语言的结构体进行表示：
- 用JSON结构表示：

```
struct uart_struct {  
    const char *name;  
    unsigned baudrate;  
}  
uart = {  
    .name = "uart0",  
    .baudrate = 115200,  
};
```

```
{  
    "uart": {  
        "name": "uart0",  
        "baudrate": 115200  
    }  
}
```

Linux使用设备树来表示

- Device Tree
 - 描述硬件的数据结构
- 硬件信息可通过 Device Tree Source (DTS) 传递给内核
 - 避免在内核中编码大量硬件细节
 - 冗余的硬件信息可以包含和引用

```
/ {  
    uart:uart0@0 {  
        baudrate = <115200>;  
    };    属性名    属性值  
};
```

DTS的声明

根节点

```
/ {  
    节点名 地址  
    node@40000000 {  
        a-string-property = "a string";  
        a-string-list-property = "first string", "second string"; 数组属性  
        a-byte-data-property = [0x01 0x02 0x03];  
        child-node@0 {  
            first-child-property;  
            second-child-property;  
            a-string-property = "child string";  
            a-reference-to-some = <&node1>; 节点引用  
        };  
        child-node@1 {  
        };  
    };  
    标签 node1: node@1 {  
        an-empty-property;  
        a-cell-property = <1 2 3>; 多值属性  
        child-node1 {  
        };  
    };  
};
```

真实的DTS

- arch/arm/boot/dts/imx6sx.dtsi

```
/ {
    uart5: serial@021f4000 {
        compatible = "fsl,imx6sx-uart", "fsl,imx6q-uart", "fsl,imx21-uart";
        reg = <0x021f4000 0x4000>;
        interrupts = <GIC_SPI 30 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&clks IMX6SX_CLK_UART_IPG>, <&clks IMX6SX_CLK_UART_SERIAL>;
        clock-names = "ipg", "per";
        dmas = <&sdma 33 4 0>, <&sdma 34 4 0>;
        dma-names = "rx", "tx";
        status = "disabled";
    };
};
```

驱动对DTS的使用

- 通过compatible属性进行匹配

```
/ {
    uart5: serial@021f4000 {
        compatible = "fsl,imx6sx-uart", "fsl,imx6q-uart", "fsl,imx21-uart";
    };
};
```

```
static const struct of_device_id stm32_match[] = {
    { .compatible = "fsl,imx6sx-uart", .data = &stm32f4_info},
    { .compatible = "fsl,imx6q-uart", .data = &stm32f4_info},
    { .compatible = "fsl,imx21-uart", .data = &stm32f7_info},
    {},
};

static struct platform_driver stm32_serial_driver = {
    .driver      = {
        .of_match_table = of_match_ptr(stm32_match),
    },
};
```



操作系统与设备的交互

CPU与外设的数据交互

- **可编程 I/O (Programmable I/O)**
 - 通过CPU in/out 或 load/store 指令
 - 消耗CPU时钟周期和数据量成正比
 - 适合于简单小型的设备
- **直接内存访问 (DMA)**
 - 外设可直接访问总线
 - DMA与内存互相传输数据，传输不需要CPU参与
 - 适合于高吞吐量I/O

可编程 I/O

- **PIO (Port IO)**
 - IO设备具有独立的地址空间
 - 使用特殊的指令（如x86中的in/out指令）
- **MMIO (Memory-mapped IO)**
 - 将设备映射到连续物理内存中
 - 使用内存访问指令
 - 行为与内存不完全一样，读写会有副作用

ChCore的UART MMIO

```
u32 pl011_nb_uart_recv(void)
{
    if (!(get32((u64) UART_PPTR + UART_FR)
        & UART_FR_RXFF))
        return (get32((u64) UART_PPTR + UART_DR) & 0xFF);
    else
        return NB_UART_NRET;
}

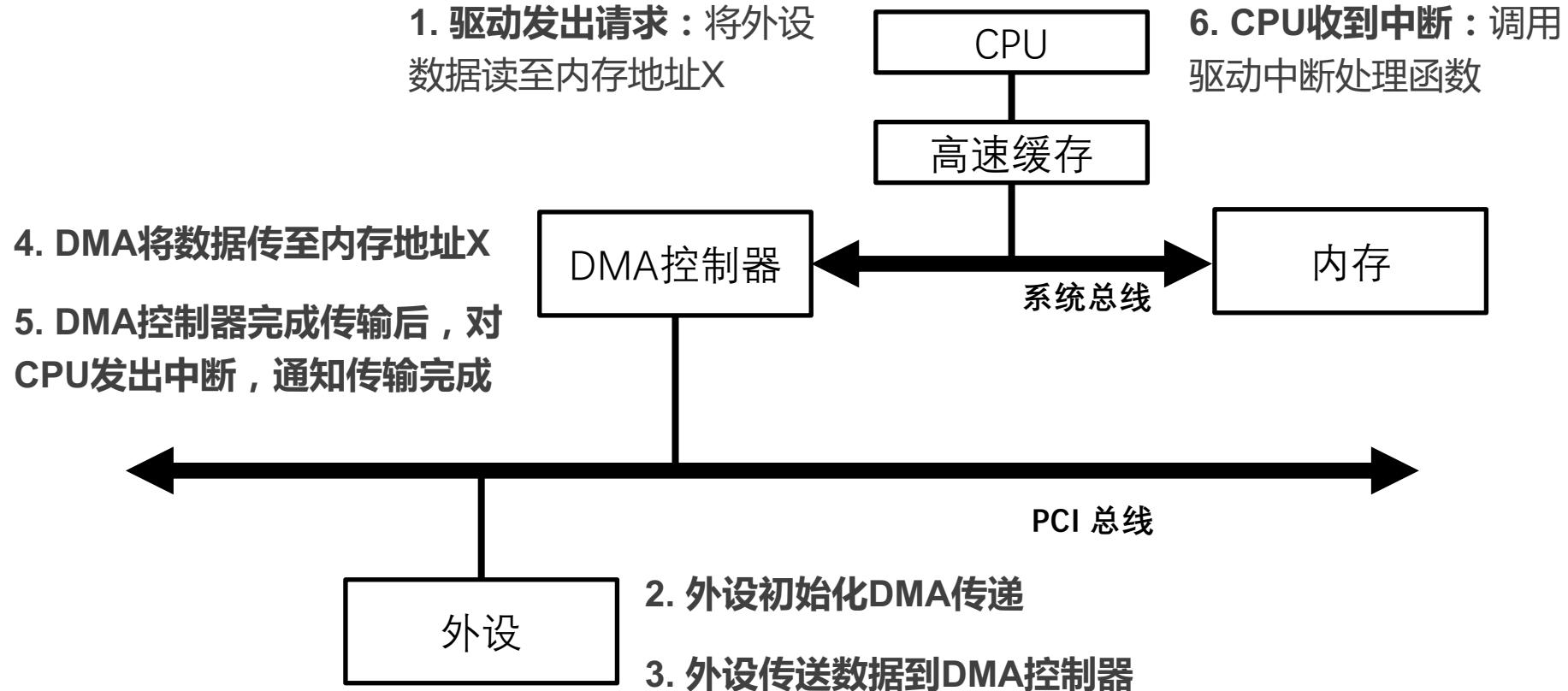
void pl011_uart_send(u32 ch)
{
    /* Wait until there is space in the FIFO or device is disabled */
    while (get32((u64) UART_PPTR + UART_FR)
        & UART_FR_TXFF) {
    }
    /* Send the character */
    put32((u64) UART_PPTR + UART_DR, (unsigned int)ch);
}
```

```
BEGIN_FUNC(get32)
    ldr w0, [x0]
    ret
END_FUNC(get32)
```

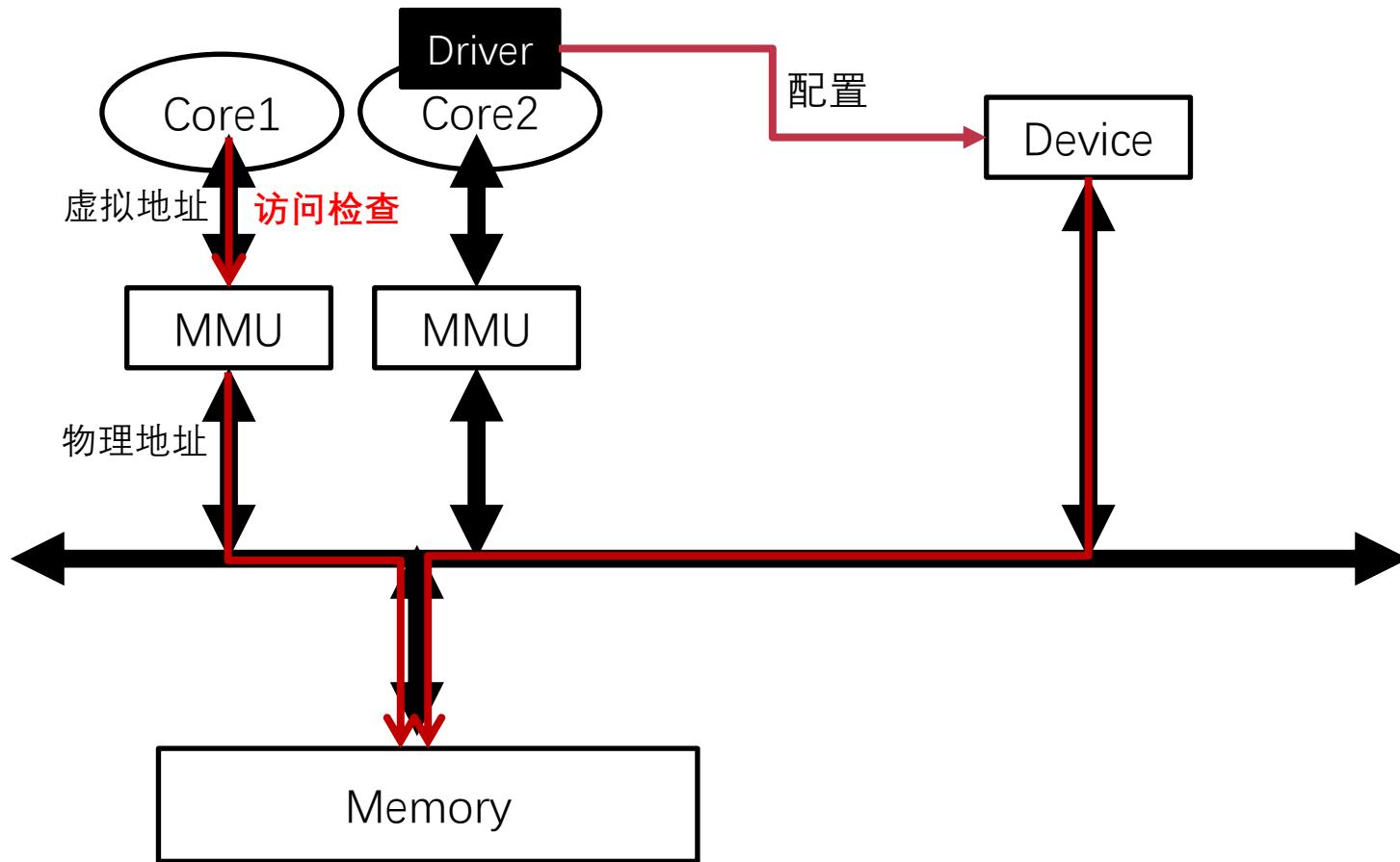
- **MMIO: 复用ldr和str指令**
 - 映射到物理内存的特殊地址段

```
BEGIN_FUNC(put32)
    str w1, [x0]
    ret
END_FUNC(put32)
```

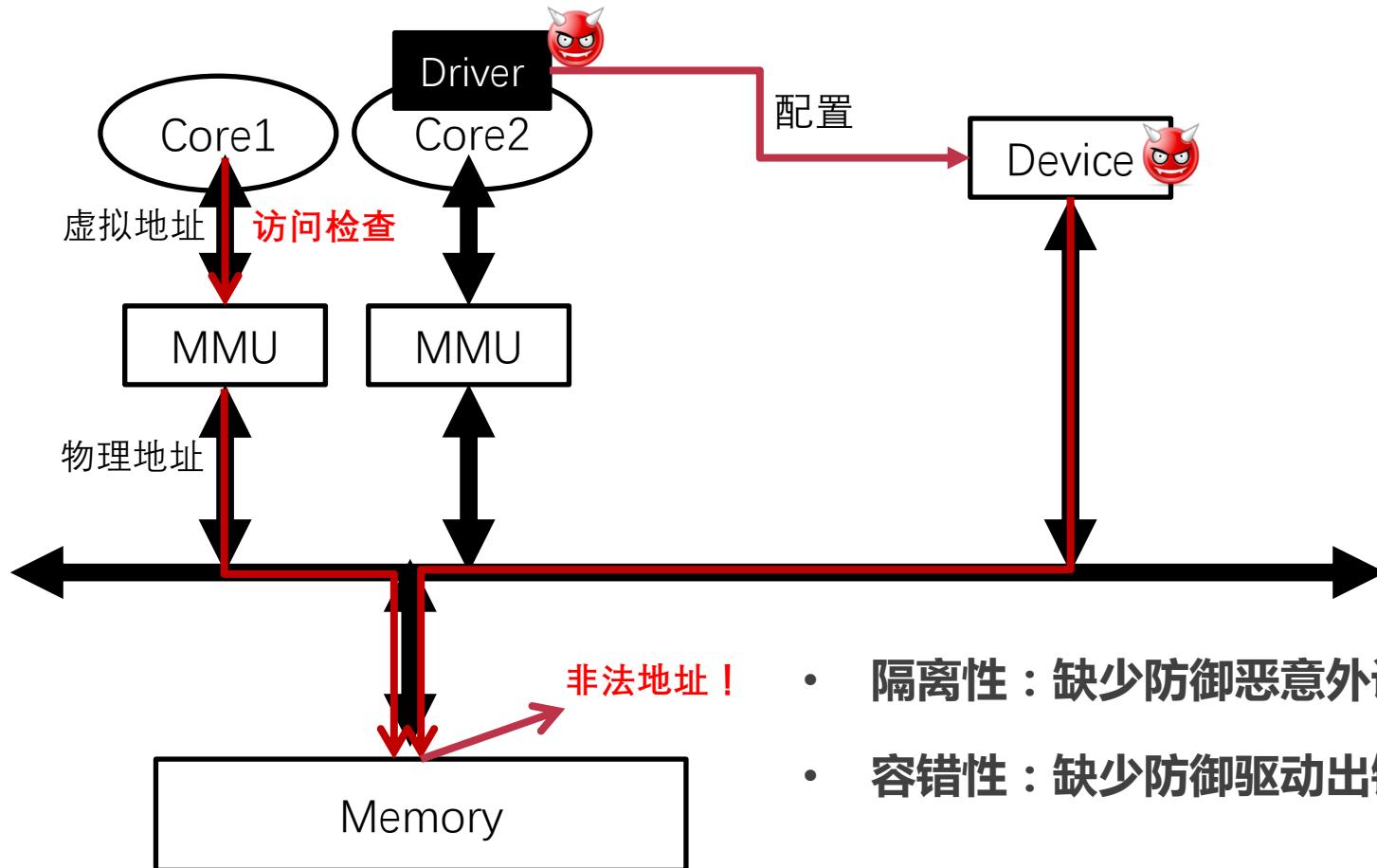
DMA



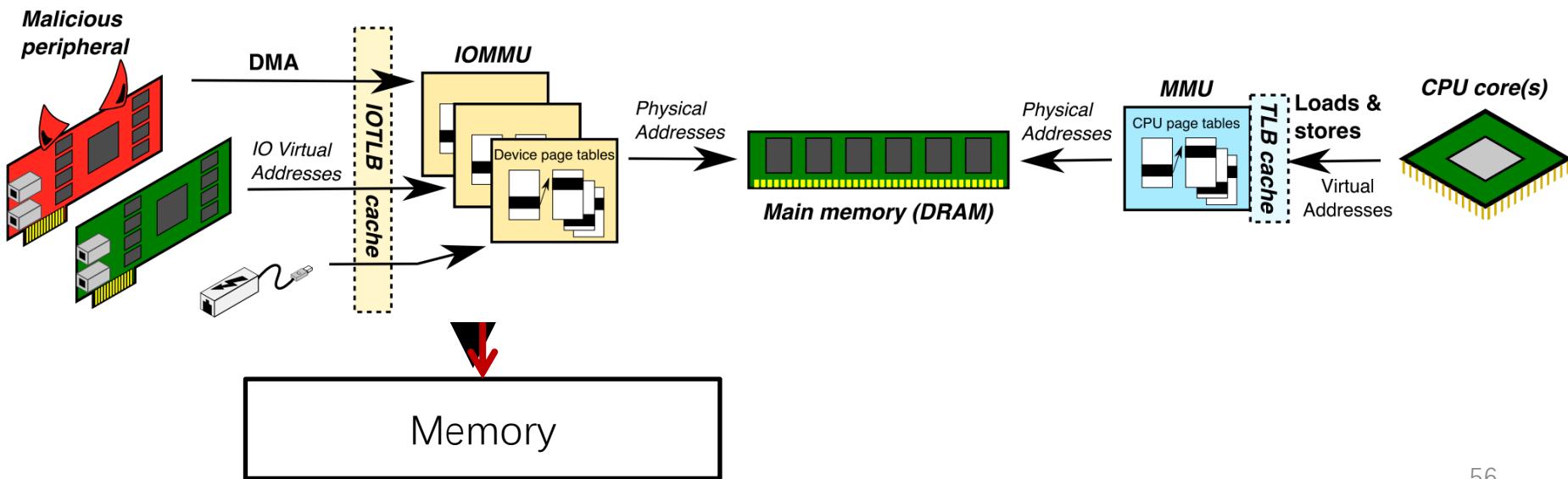
为什么需要IOMMU ?



为什么需要IOMMU ?



引入IOMMU

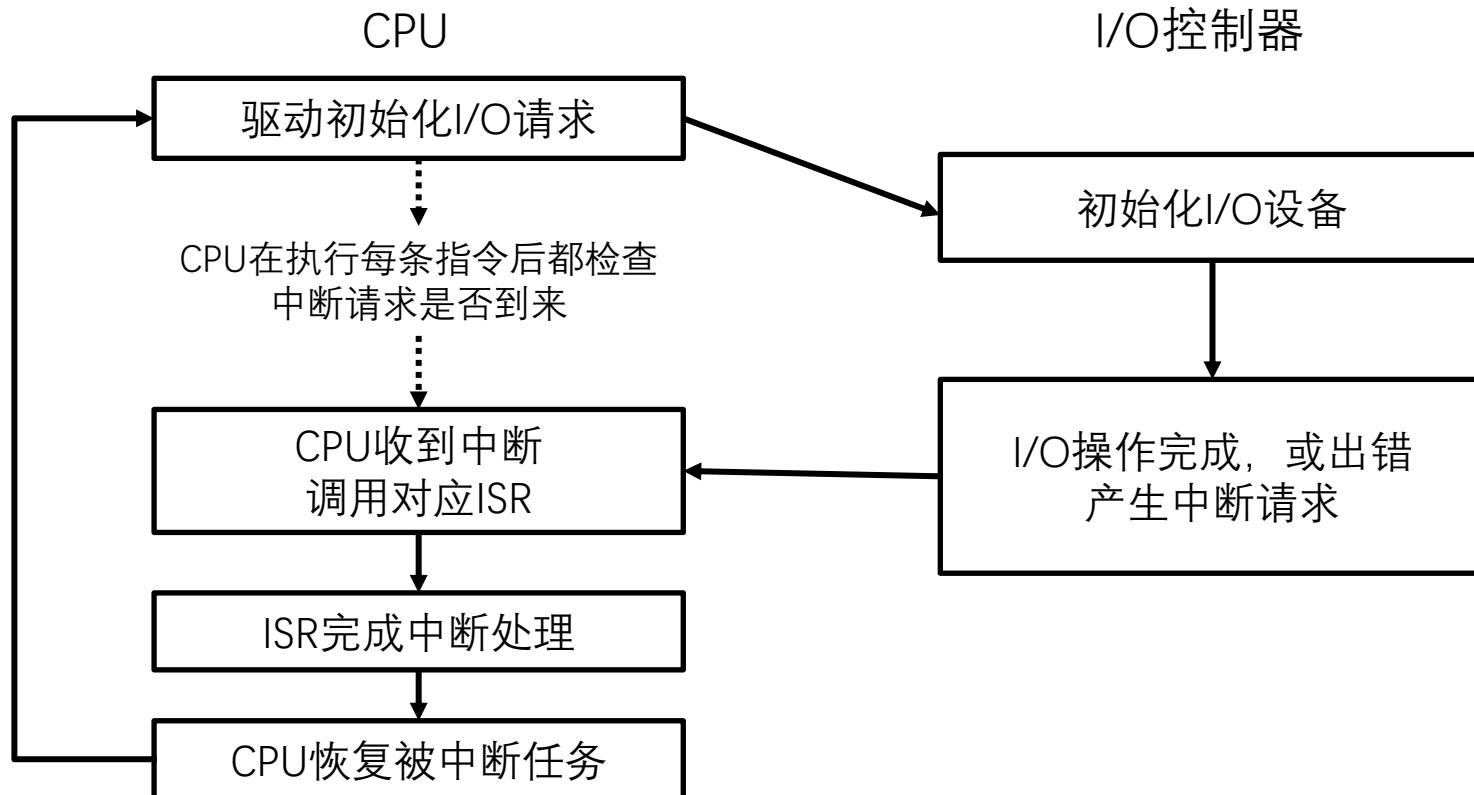




设备通知驱动的方式

中断与中断管理

CPU中断处理流程



中断

```
$ cat /proc/interrupts
```

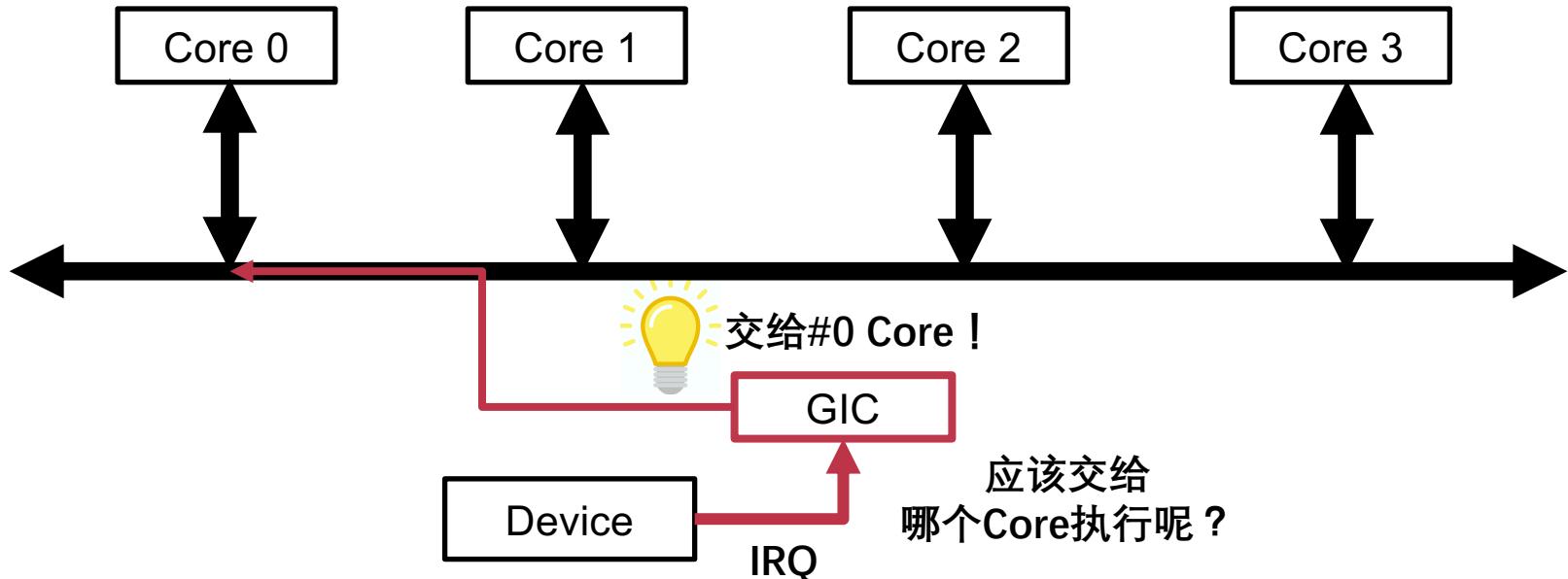
CPU0			
0:	865119901	IO-APIC-edge	timer
1:	4	IO-APIC-edge	keyboard
2:	0	XT-PIC	cascade
8:	1	IO-APIC-edge	rtc
12:	20	IO-APIC-edge	PS/2 Mouse
14:	6532494	IO-APIC-edge	ide0
15:	34	IO-APIC-edge	ide1
16:	0	IO-APIC-level	usb-uhci
19:	0	IO-APIC-level	usb-uhci
23:	0	IO-APIC-level	ehci-hcd
32:	40	IO-APIC-level	ioc0
33:	40	IO-APIC-level	ioc1
48:	273306628	IO-APIC-level	eth0

AArch64中断分类

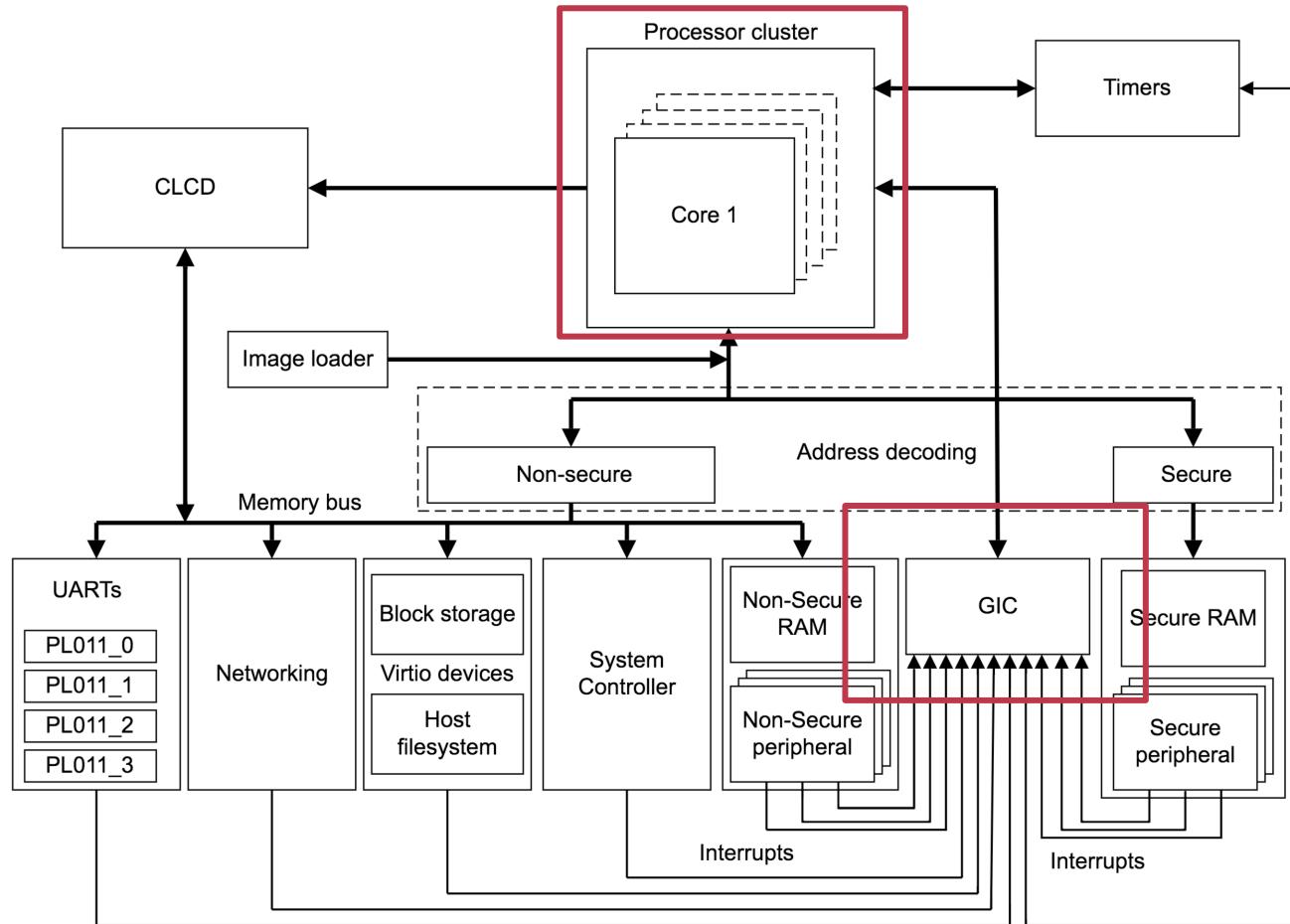
- **IRQ (Interrupt Request)**
 - 普通中断，优先级低，处理慢
 - **FIQ (Fast Interrupt Request)**
 - 一次只能有一个FIQ
 - 快速中断，优先级高，处理快
 - 常为可信任的中断源预留
 - **SError (System Error)**
 - 原因难以定位、较难处理的异常，多由异步中止（ Abort ）导致
 - 如从缓存行（ Cacheline ）写回至内存时发生的异常
- 
- 连接CPU的不同针脚
- 可在**中断控制器**（ Interrupt Controller ）中配置

问题：多核CPU如何处理中断？

- 中断：如何避免打断所有核呢？

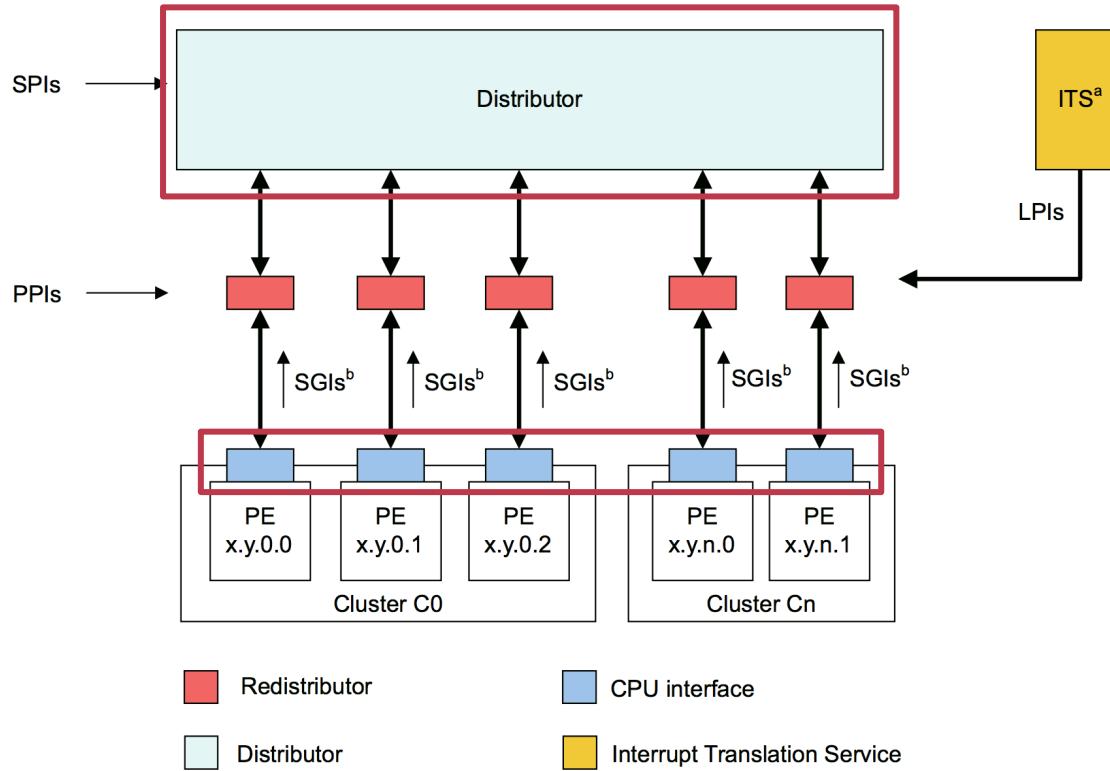


ARM 中断控制器——GIC



GIC

- Generic
- Distribu'
– 负责全
- CPU Inte'
– 类似 "



- a. The inclusion of an ITS is optional, and there might be more than one ITS in a GIC.
- b. SGIs are generated by a PE and routed through the Distributor.

Figure 3-2 GIC logical partitioning with an ITS

Distributor

- **中断分发器：**
 - 将当前最高优先级中断转发给对应CPU Interface
- **寄存器：GICD**
- **作用：**
 - 中断使能
 - 中断优先级
 - 中断分组
 - 中断的目的core
 - 中断触发方式
 - 中断状态管理

CPU Interface

- **CPU接口：**
 - 将GICD发送的中断信息，通过IRQ、FIQ管脚，发送给连接到interface的core
- **寄存器：GICC**
- **作用：**
 - 将中断请求发送给cpu
 - 中断确认（acknowledging an interrupt）
 - 中断完成（indicating completion of an interrupt）
 - 设置中断优先级屏蔽
 - 定义中断抢占策略

中断确认

- CPU开始响应中断：
 - IRQ状态：pending→active
- 寄存器：GICC_IAR，记录当前等待处理的中断号
- 通过访问GICC_IAR寄存器，来对中断进行确认

中断完成

思考：为什么
要将“中断完成”分成两步走？

- **CPU处理完中断：**
 - IRQ状态： $\text{active} \rightarrow \text{inactive}$
- **优先级重置 (priority drop) :**
 - 将当前中断屏蔽的最高优先级进行重置，以便能够响应低优先级中断
 - 寄存器：GICC_EOIR
- **中断无效 (interrupt deactivation) :**
 - 将中断的状态置为inactive状态
 - 寄存器：GICC_DIR

ARM中断的生命周期

- ① **Generate** : 外设发起一个中断
- ② **Distribute** : Distributor对收到的中断源进行仲裁，然后发送给对应的CPU Interface
- ③ **Deliver** : CPU Interface将中断传给core
- ④ **Activate** : core读 GICC_IAR 寄存器，对中断进行确认
- ⑤ **Priority drop**: core写 GICC_EOIR 寄存器，实现优先级重置
- ⑥ **Deactivate** : core写 GICC_DIR 寄存器，来无效该中断

案例分析

- kernel/arch/aarch64/machine/gic_v2.c

```
void gicv2_handle_irq(void)
{
    unsigned int irqnr = 0;
    unsigned int irqstat = 0;
    int r;

    irqstat = get32(GICC_IAR);
    irqnr = irqstat & 0x3ff;
    BUG_ON(irqnr >= MAX_IRQ_NUM);
    if (irq_handle_type[irqnr] == HANDLE_USER) {
        r = user_handle_irq(irqnr);
        BUG_ON(r);
        return;
    }

    switch (irqnr) {
        /* virtual timer interrupt */
        case 0x1b:
            handle_timer_irq();
            break;
        default:
            if (irqnr < 16)
                handle_ipi(irqnr);
            else
                kinfo("Unsupported IRQ 0x%x\n", irqstat);
    }

    put32(GICC_EOIR, irqstat);
    put32(GICC_DIR, irqstat);
}
```

• 中断确认

• 中断处理

• 中断完成 (两步走)

问题：如果有多个中断同时发生怎么办？

- 中断优先级：
 - 当多个中断同时发生时（NMI、软中断、异常），CPU首先响应高优先级的中断
- ARM Cortex-M 处理器的中断优先级如下所示：

类型	优先级
复位 (reset)	-3
不可屏蔽中断 (NMI)	-2
硬件故障 (Hard Fault)	-1
系统服务调用 (SVcall)	可配置
调试监控 (debug monitor)	可配置
系统定时器 (SysTick)	可配置
外部中断 (External Interrupt)	可配置

中断嵌套

- 中断也能被 “**中断**” !
- 在处理当前中断（ISR）时：
 - 更高优先级的中断产生；或者
 - 相同优先级的中断产生
- 那么该如何响应？
 - 允许高优先级抢占
 - 同级中断无法抢占
- ARM的FIQ能抢占任意IRQ，FIQ不可抢占

如何禁止中断被抢占？

- 中断屏蔽：
 - 屏蔽全局中断：不再响应任何外设请求
 - 屏蔽对应中断：只对对应IRQ停止响应
- 什么策略合适？
 - 屏蔽全局中断：
 - 1. 系统关键步骤（原子性）
 - 2. 保证任务响应的实时性
 - 屏蔽对应中断：通常都是这种情况，对系统的整体影响最小

如何设计中断处理函数（ISR）？

- **中断应该尽快响应**
 - 提高系统对外部的实时响应能力
- **尽量短**
 - Linux上半部：马上处理
- **可重入**
 - 应允许在中断过程的任意时刻被抢占

思考：中断上下文能否睡眠？

- 考虑如下场景：

- ① Process 1进入内核态
- ② Process 1获得 Lock A
- ③ 中断发生
- ④ ISR 试图拿锁 Lock A
- ⑤ ISR 调用sleep，等待Lock A被释放

- 死锁：

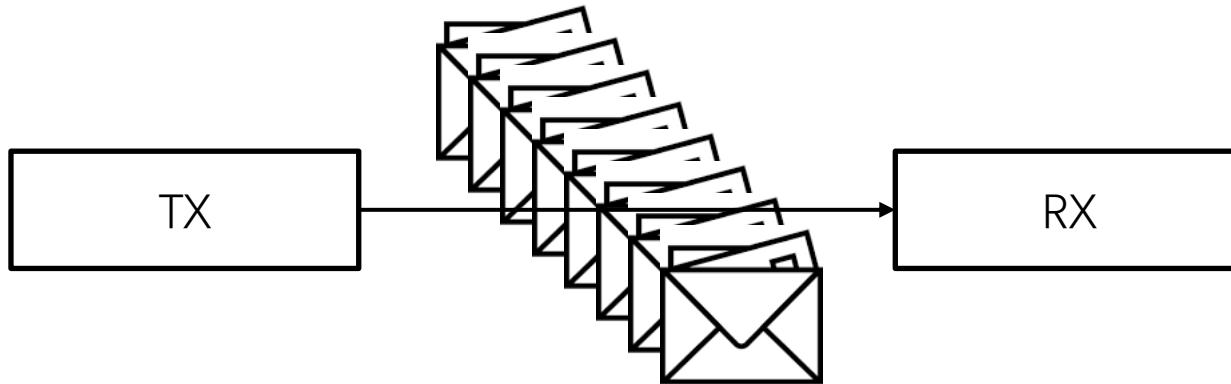
- Process 1必须等待ISR返回，但ISR在等待Process 1释放锁LockA

- 在中断上下文中睡眠，内核将被挂起

<https://stackoverflow.com/questions/1053572/why-kernel-code-thread-executing-in-interrupt-context-cannot-sleep>

思考：

- 如果发端拼命发数据，收端首先收到什么数据？**都可能**
 - 收端来不及处理数据，导致之前的数据被覆盖了？ **轮询**
 - 收端在处理中断上下文中，导致后来数据没有收到？ **中断**



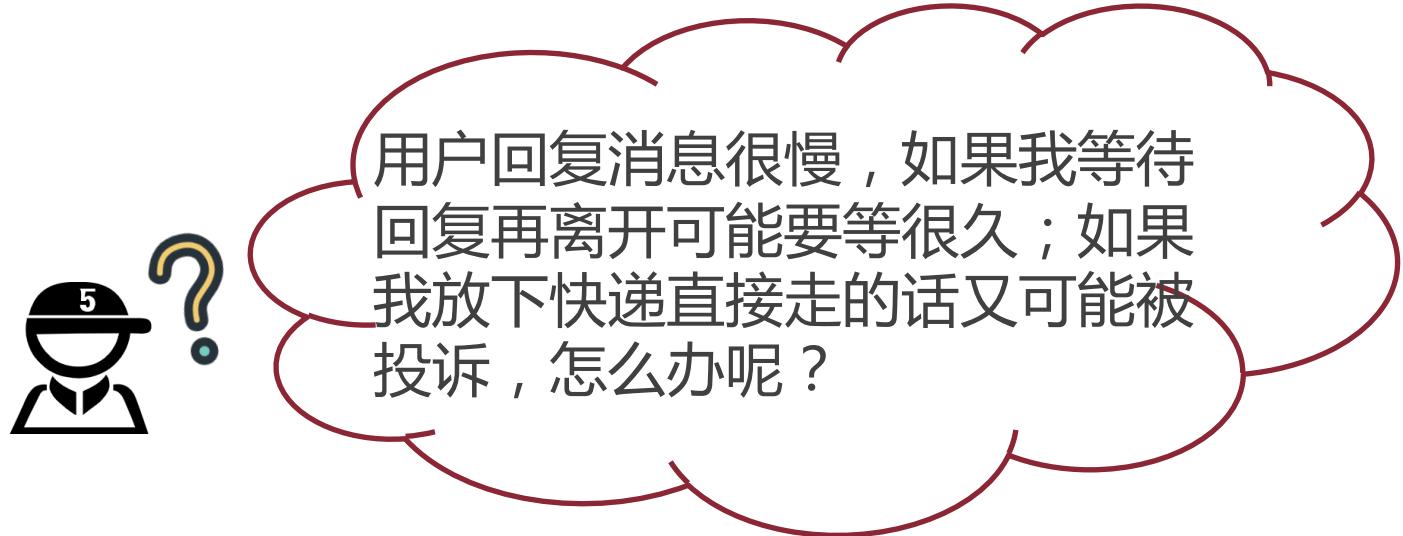
对策

- 静态配置法
 - UART驱动需要在初始化时指定“波特率”
- 动态协商法
 - TCP设计了流量控制机制，发端逐步试探出收端收包能力的上限
- 思考
 - UDP没有流量控制，那么用户会收到什么数据包（last or first?）

问题：如何更好地处理I/O请求？

- 和外设交互费事费力：
 - 每次I/O交互都要花费不少CPU周期
 - 部分设备运行较为低速
- 问题：OS应该如何选择时机，**恰到好处**地处理I/O请求？

回顾：IPC快递员的烦恼



- 快递员想要弄清楚自己是等待消息被确认呢(阻塞)
- 还是发送完消息就赶去送下一个快递呢(非阻塞)

I/O请求的阻塞与非阻塞

- **阻塞I/O：一直等待**
 - 进程请求读数据不得，将其挂起，直到数据来了再将其唤醒
 - 进程请求写数据不得，将其挂起，直到设备准备好了再将其唤醒
- **非阻塞I/O：不等待**
 - 读写请求后直接返回（可能读不到数据或者写失败）
- **异步I/O：稍后再来**
 - 用户提供缓存区，内核等成功操作完数据后来再通知用户，用户执行并不停滞（类似中断请求之于CPU）

EOI (End Of Interrupt)

- 告诉中断控制器完成处理
- 如果不显示
并屏蔽后继续

没有响应，

```
void gicv2_handle_irq(void)
{
    unsigned int irqnr = 0;
    unsigned int irqstat = 0;
    int r;

    irqstat = get32(GICC_IAR);
    irqnr = irqstat & 0x3ff;

    BUG_ON(irqnr >= MAX_IRQ_NUM);
    if (irq_handle_type[irqnr] == HANDLE_USER) {
        r = user_handle_irq(irqnr);
        BUG_ON(r);
        return;
    }

    switch (irqnr) {
        /* virtual timer interrupt */
        case 0x1b:
            handle_timer_irq();
            break;
        default:
            if (irqnr < 16)
                handle_ipi(irqnr);
            else
                kinfo("Unsupported IRQ 0x%x\n", irqstat);
    }
    put32(GICC_EOI, irqstat);
    put32(GICC_DIR, irqstat);
}
```

中断处理的原则

- 观察：
 - 中断抢占了CPU当前执行的任务
 - 中断导致用户任务无法得到快速响应
 - 处理中断时必须屏蔽当前号IRQ，设备缓冲区不够时会导致中断丢失
- 因此，中断处理应该**越快越好**：
 - ISR只做很小的一部分：将某些数据移出缓冲区、标记flag
 - 将更多非关键操作推迟到后期完成

中断子系统

案例：LINUX的上下半部

Top Half : 马上做

- **最小的化公共例程 :**
 - 保存寄存器、屏蔽中断
 - 恢复寄存器，返回现场
- **最重要：调用合适的由硬件驱动提供的中断处理handler**
- **因为中断被屏蔽，所以不要做太多事情（时间、空间）**
- **将请求放入队列（或设置flag），将其他处理推迟到 bottom half**

Top Half : 找到handler

思考：
为什么要共享IRQ ?

- 现代处理器中，多个I/O设备共享一个IRQ和中断向量
- 多个ISR (interrupt service routines)可以绑定在同一个向量上
- 调用每个设备对应的IRQ的ISR

Bottom Half : 延迟完成

- 提供可以推迟完成任务的机制
 - softirqs
 - tasklets (建立在softirqs之上)
 - 工作队列
 - 内核线程
- 这些机制都可以被中断

软中断 (Softirqs)

- 静态分配：在内核编译时期确定
- 数量有限：

<i>Priority</i>	<i>Type</i>
0	High-priority tasklets
1	Timer interrupts
2	Network transmission
3	Network reception
4	Block devices
5	Regular tasklets

Softirq特点

- **执行时间点：**
 - 中断之后（上半部之后）
 - 系统调用或是异常发生之后
 - 调度器显式执行ksoftirqd
- **并发：**
 - 可以在多核上同时执行
 - 必须是可重入的
 - 或根据需要加锁
- **可中断：Softirq运行时可被硬中断IRQ中断**

Softirq Rescheduling

- **软中断要求能被重调度**
 - 在处理软中断A时，能切换至软中断B（挂起A唤醒B）
- **问题：在处理软中断A时，软中断产生了B，怎么办？**
 - 不处理→B响应被延迟
 - 总是处理→如果软中断很长→用户程序被饿死？**活锁！**
- **方案：配额（quota）+ ksoftirqd**
 - Softirq调度器每次只运行有限数量的请求
 - 剩余请求有内核线程ksoftirqd代为执行，和用户进程抢CPU
 - ksoftirqd和用户进程都被调度器调度

Tasklet

- **问题**：Softirq是静态的
- **方案**：引入Tasklet
 - 基于Softirqs，但可以被动态创建和销毁！
 - 同一时期，同种类型的Tasklet一次只能运行在一个CPU上
 - 不同类型的Tasklet可以同时运行在不同CPU上
- **缺点**：
 - 和Softirq一样缺乏进程上下文，无法睡眠！

Tasklet的优势

- 可动态分配，数量不限
- 直接运行在调度它的CPU上（缓存亲和性）
- 执行期间不能被其它下半部抢占
 - 不存在重入的问题
 - 无需加锁
- 曾是最受原因的延迟处理机制

Tasklet的问题

- **难以正确实现**
 - 要防止休眠代码
- **不可抢占性：**
 - 比其他任务的优先级都高，影响任务实时性
 - 导致不可控的延迟
- **Linux社区一直在讨论是否要移除Tasklet**

工作队列 (Work Queues)

- Softirq和Tasklet使用中断上下文
- 工作队列使用进程上下文
 - 可以睡眠！
- 方式：
 - 在内核空间维护FIFO队列，workqueue内核进程不断轮询队列
 - 中断负责enqueue(fn, args)，workqueue负责dequeue并执行fn(args)
- 特点：
 - 只在内核空间，不和任何用户进程关联，没有跨模式切换和数据拷贝

内核线程 (Kernel Threads)

- **始终运行在内核态**
 - 和工作队列一样，没有用户空间上下文
- **中断线程化* (*threaded interrupt handlers*)**
 - Linux 2.6.30引入
 - 想要取代Tasklet和Workqueue
 - 每个中断线程都有自己的上下文

* [LWN] Moving interrupts to threads, <https://lwn.net/Articles/302043/>

总结

特点	ISR	SoftIRQ	Tasklet	WorkQueue	KThread
禁用所有中断？	Briefly	No	No	No	No
禁用相同优先级的中断？	Yes	Yes	No	No	No
比常规任务优先级更高？	Yes	Yes*	Yes*	No	No
在相同处理器上运行？	N/A	Yes	Yes	Yes	Maybe
允许在同一CPU上有多个实例同时运行？	No	No	No	Yes	Yes
允许在多个CPU上运行同时多个相同实例？	Yes	Yes	No	Yes	Yes
完整的模式切换？	No	No	No	Yes	Yes
能否睡眠（拥有自己的内核栈）？	No	No	No	Yes	Yes
能否访问用户空间？	No	No	No	No	No

* 可以由 ksoftirqd 调度

为什么ARM中断完成确认分为两步走？

- ARM的中断完成确认分为两步：
 - ① 优先级重置：允许响应低优先级
 - ② 中断无效：将IRQ状态置为inactive
- 和Linux上下半部的结合：
 - 上半部：确认中断源，并将当前最高优先级重置，此时低优先级的中断不再被屏蔽（低优先级不必等到高优先级完成）
 - 下半部：处理完中断再通知GIC
 - 提高整体系统中断响应的实时性

为什么要共享中断？

- **IRQ是有限资源**
 - 可以通过多个设备共享同一中断号来解决需求
- **Linux将同一中断的ISR组成链表**
 - IRQ到来后，内核对每个中断处理程序都要执行
 - 所有该中断的“订阅者”都会查询自己的设备寄存器，以确定当前中断是不是自己的设备发出的
- **对于慢速设备，就会造成很大的开销**

下次课内容

- 网络