中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

1924-2024 中山大學 世纪华诞 100th ANNIVERSARY SUN YAT-SEN UNIVERSITY

# 设备管理II

SSE202/204: 操作系统原理

苏玉鑫

suyx35@mail.sysu.edu.cn

助教：龙玉丹 单诗雯 毛晨希 沈志轩 郑灿峰 胡伟峰

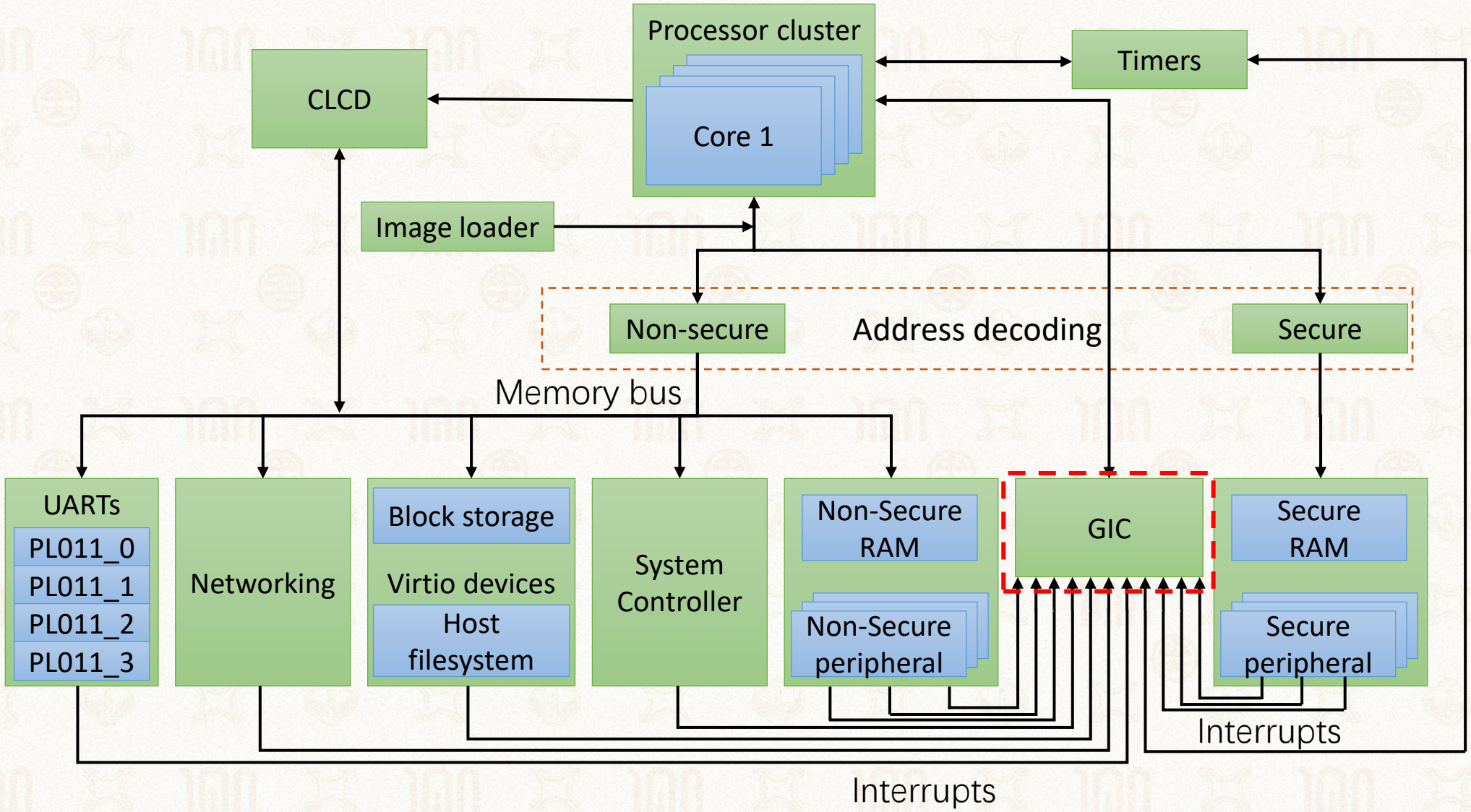
# 版权信息

- 部分内容来自：上海交通大学并行与分布式系统研究所操作系统课件
  - https://ipads.se.sjtu.edu.cn/courses/os/
- 其它参考资料：
  - 清华大学操作系统公开课
    - https://open.163.com/newview/movie/courseintro?newurl=ME1NSA351
    - 介绍标准内容，适合考研
  - 南京大学计算机软件研究所
    - http://jyywiki.cn/OS/2025/
    - https://space.bilibili.com/202224425/channel/collectiondetail?sid=192498
    - 比较有趣

# 大纲

- 设备连接
- 设备类型抽象
  - 字符设备
  - 块设备
  - 网络设备
- 设备与操作系统的交互
  - 可编程I/O
  - 直接内存访问
- 操作系统如何响应设备：中断
  - 中断优先级
  - 硬中断
  - 软中断
- 操作系统如何管理设备
  - 驱动程序
    - 驱动模型
  - 设备树

# 中断控制器


Processor cluster
Core 1
CLCD
Timers
Image loader
Non-secure
Address decoding
Secure
Memory bus
UARTs
PL011_0
PL011_1
PL011_2
PL011_3
Networking
Block storage
Virtio devices
Host filesystem
System Controller
Non-Secure RAM
Non-Secure peripheral
GIC
Secure RAM
Secure peripheral
Interrupts
Interrupts

# AArch64中断的分类

- 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

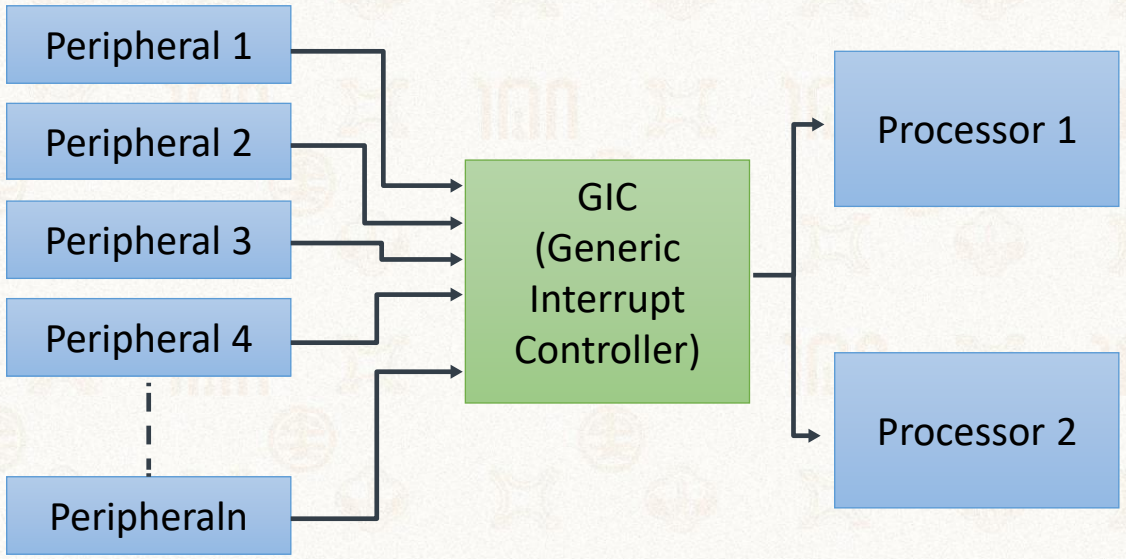
连接CPU的不同针脚

可在中断控制器中配置

- 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

# 中断控制器需要考虑的问题

- 如何指定不同中断的优先级
  - 低优先级中断处理中，出现了高优先级的中断
  - 嵌套中断
- 中断交给谁处理
- 如何与软件协同




- 主要功能
  - 分发：管理所有中断、决定优先级、路由
  - CPU接口：给每个CPU核有对应的接口

# 查看中断

```
root@iZm5e96bky2okvcid149vpZ:~# cat /proc/interrupts
           CPU0
  1:          9   IO-APIC   1-edge      i8042
  4:       1750   IO-APIC   4-edge      ttyS0
  6:          3   IO-APIC   6-edge      floppy
  8:          0   IO-APIC   8-edge      rtc0
 11:         31   IO-APIC  11-fasteoi   virtio3, uhci_hcd:usb1
 12:         15   IO-APIC  12-edge      i8042
 14:          0   IO-APIC  14-edge      ata_piix
 24:          0   PCI-MSI 65536-edge      virtio1-config
 27:   13792406   PCI-MSI 49153-edge      virtio0-input.0
 28:   17143741   PCI-MSI 49154-edge      virtio0-output.0
NMI:          0   Non-maskable interrupts
LOC:   74135788   Local timer interrupts
SPU:          0   Spurious interrupts
PMI:          0   Performance monitoring interrupts
IWI:    1014709   IRQ work interrupts
RTR:          0   APIC ICR read retries
RES:          0   Rescheduling interrupts
CAL:          0   Function call interrupts
TLB:          0   TLB shootdowns
TRM:          0   Thermal event interrupts
THR:          0   Threshold APIC interrupts
```

- 这是一个单核系统的中断状态

```
root@iZm5e96bky2okvcid149vpZ:~# cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 85
model name      : Intel(R) Xeon(R) Platinum 8163
CPU @ 2.50GHz
stepping        : 4
microcode       : 0x1
cpu MHz         : 2499.996
cache size      : 33792 KB
physical id     : 0
siblings        : 1
core id         : 0
cpu cores       : 1
```

# 查看中断

```
os@ubuntu:~$ cat /proc/interrupts
           CPU0       CPU1
  0:          2          0   IO-APIC    2-edge      timer
  1:         20         36   IO-APIC    1-edge      i8042
  8:          1          0   IO-APIC    8-edge      rtc0
  9:          0          0   IO-APIC    9-fasteoi   acpi
 12:        241       1225   IO-APIC   12-edge      i8042
 14:          0          0   IO-APIC   14-edge      ata_piix
 15:          0          0   IO-APIC   15-edge      ata_piix
 16:        325        502   IO-APIC   16-fasteoi   vmwgfx
 17:      12570          0   IO-APIC   17-fasteoi   ehci_hcd:usb1, ioc0
 18:          0         65   IO-APIC   18-fasteoi   uhci_hcd:usb2
 19:          2        172   IO-APIC   19-fasteoi   ens33
 24:          0          0   PCI-MSI 344064-edge      PCIe PME, pciehp
 56:          0        111   PCI-MSI 1130496-edge      ahci[0000:02:05.0]
NMI:          0          0   Non-maskable interrupts
 LOC:     12575      11820   Local timer interrupts
 SPU:         0          0   Spurious interrupts
 PMI:         0          0   Performance monitoring interrupts
 IWI:         0          0   IRQ work interrupts
RES:       181        448   Rescheduling interrupts
 CAL:     20028      24616   Function call interrupts
 TLB:       371        611   TLB shootdowns
```

- 这是一个双核系统的中断状态
  - 每个逻辑核心接受到的中断类型和次数是不一样的
- 查看中断信息是一个了解系统运行细节的好途径

在多核环境下运行的多线程程序，容易使得不同核心TLB之间产生不一致

# 中断优先级

- 如果有多个中断同时发生怎么办?

- 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

| 类型 | 优先级（值越低，优先级越高） |
| --- | --- |
| 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

# 大纲

- 设备连接
- 设备类型抽象
  - 字符设备
  - 块设备
  - 网络设备
- 设备与操作系统的交互
  - 可编程I/O
  - 直接内存访问
- 操作系统如何响应设备：中断
  - 中断优先级
  - 硬中断
  - 软中断
- 操作系统如何管理设备
  - 驱动程序
    - 驱动模型
  - 设备树

在中断发生后，进入中断处理的程序属于

用户程序

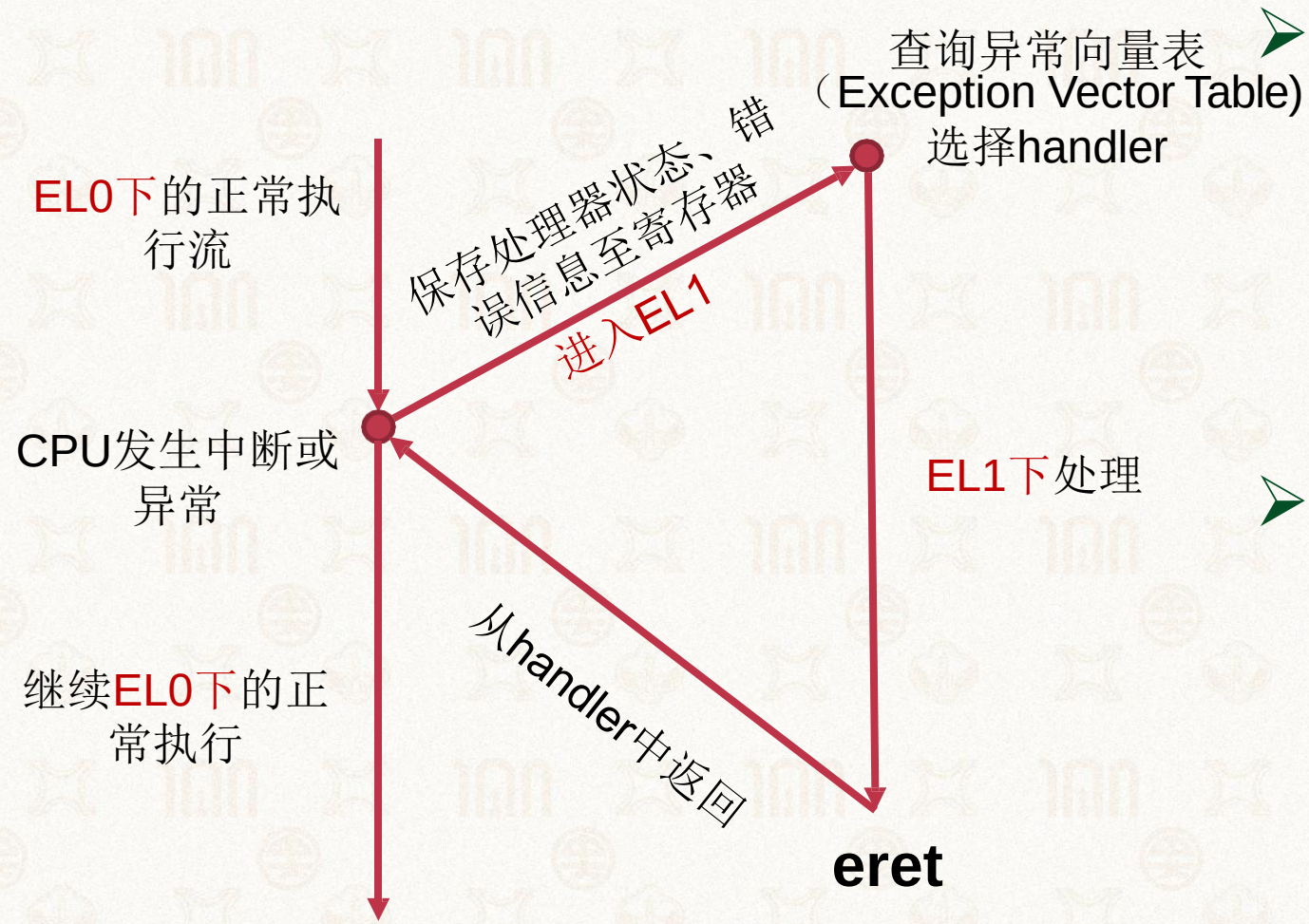
可能是用户程序，也可能是操作系统程序

操作系统程序

单独的程序，既不是用户程序也不是操作系统程序

提交

# 如何设计中断处理函数(Interrupt Service Routine, ISR)



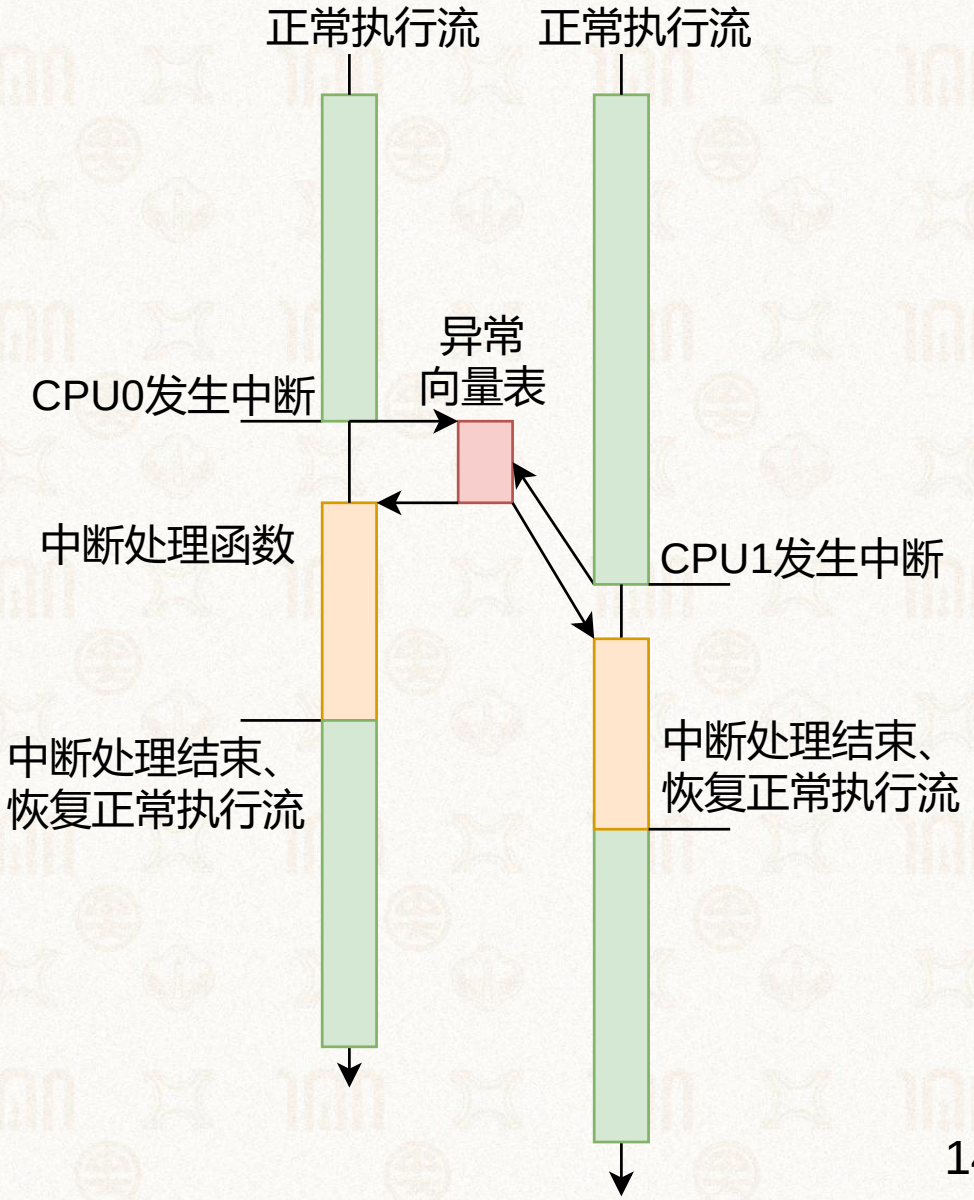

- 观察：
  - 中断抢占了CPU当前执行的任务
  - 中断导致用户任务无法得到快速响应
  - 处理中断时必须屏蔽当前号IRQ，设备缓冲区不够时会导致中断丢失
- 因此，中断处理应该越快越好：
  - ISR只做很小的一部分：将某些数据移出缓冲区、标记flag
  - 将更多非关键操作推迟到后期完成

```
static inline int __must_check request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
            const char *name, void *dev) {
    return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}
```

https://elixir.bootlin.com/linux/v5.16.14/source/include/linux/interrupt.h#L165

# Linux的上下半部

- Top Half：马上做
  - 调用合适的由硬件驱动提供的中断处理handler (处理硬中断)
  - 因为中断被屏蔽，所以不要做太多事情（时间、空间）
- Bottom Half：延迟完成
  - 每个硬中断都可再注册下半部任务
  - 重、非关键操作
  - 提供可以推迟完成任务的机制
    - 软中断(softirqs)
    - tasklets (建立在softirqs之上)
    - 工作队列
  - 这些机制都可以被中断



# 大纲

- 设备连接
- 设备类型抽象
  - 字符设备
  - 块设备
  - 网络设备
- 设备与操作系统的交互
  - 可编程I/O
  - 直接内存访问
- 操作系统如何响应设备：中断
  - 中断优先级
  - 硬中断
  - 软中断
- 操作系统如何管理设备
  - 驱动程序
    - 驱动模型
  - 设备树

# 下半部：软中断（Softirqs）

- 静态分配：在内核编译时期确定
- 数量有限：

| 优先级 | 名称 | 含义 |
|---|---|---|
| 0 | HI_SOFTIRQ | 用于处理TASKLET_HI，优先级最高 |
| 1 | TIMER_SOFTIRQ | 用于处理每个核上的定时器软中断 |
| 2 | NET_TX_SOFTIRQ | 用于处理发送网卡数据包的软中断 |
| 3 | NET_RX_SOFTIRQ | 用于处理接收网卡数据包的软中断 |
| 4 | BLOCK_SOFTIRQ | 用于处理块设备中断 |
| 5 | IRQ_POLL_SOFTIRQ | 用于执行I/O轮询的回调函数 |
| 6 | TASKLET_SOFTIRQ | 用于处理Tasklet |
| 7 | SCHED_SOFTIRQ | 用于执行调度相关的负载均衡操作 |
| 8 | HRTIMER_SOFTIRQ | 用于处理高精度定时器的软中断 |
| 9 | RCU_SOFTIRQ | 用于处理RCU锁的软中断，优先级最低 |

https://elixir.bootlin.com/linux/v5.16.14/source/include/trace/events/irq.h#L13

# 软中断（Softirqs）的特点

- 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运行时可再被中断抢占

# 可重入函数

- 不可重入函数：
  - 使用了全局变量，多个线程调用同一个函数可能出现不一致的结果

```
int i;

int fun1() {
    return i * 5;
}
int fun2() {
    return fun1() * 5;
}
```

- 可重入函数：
  - 多个线程调用同一个函数均可保持一致

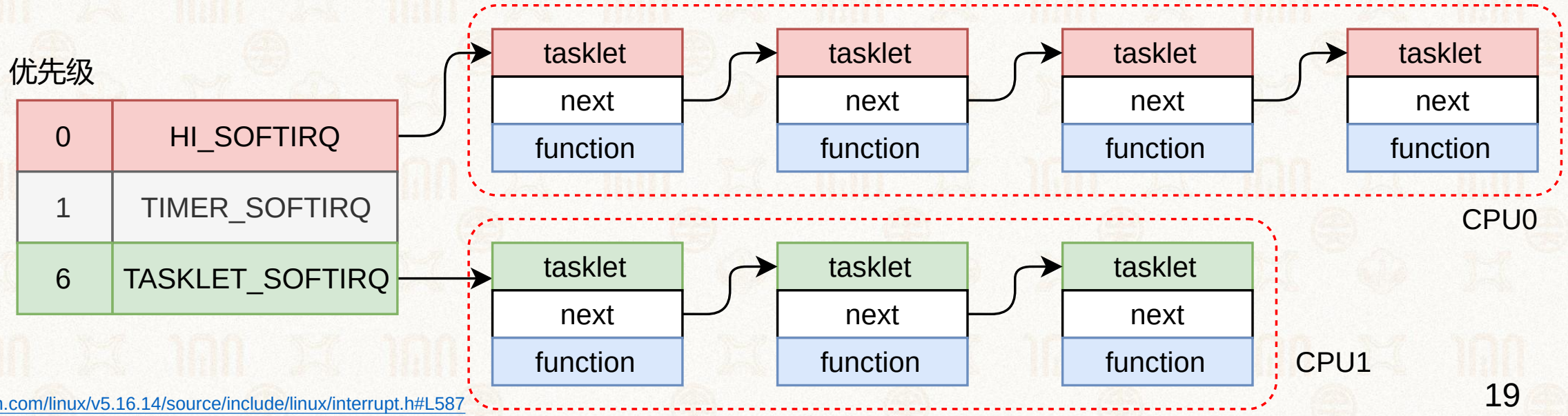
```
int fun1(int i) {
    return i * 5;
}

int fun2(int i) {
    return fun1(i) * 5;
}
```

# 下半部：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一次只能运行一个
  - 不同类型的Tasklet可以同时运行在不同CPU上

```
struct tasklet_struct {
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    bool use_callback;
    union {
        void (*func)(unsigned long data);
        void (*callback)(struct tasklet_struct *t);
    };
    unsigned long data;
};
```




https://elixir.bootlin.com/linux/v5.16.14/source/include/linux/interrupt.h#L587

# 下半部：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(单独创建一个进程负责处理中断函数)
  - 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)

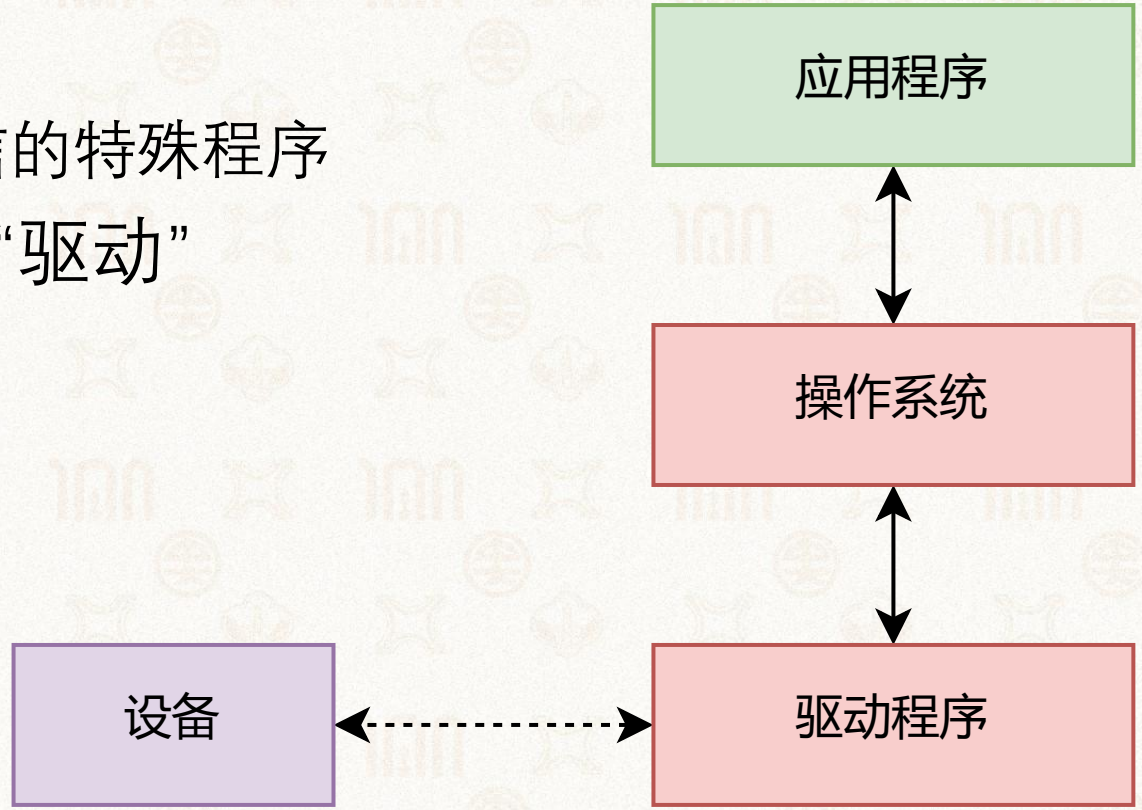
- 特点:
  - 只在内核空间，不和任何用户进程关联，没有跨模式切换和数据拷贝

# 大纲

- 设备连接
- 设备类型抽象
  - 字符设备
  - 块设备
  - 网络设备
- 设备与操作系统的交互
  - 可编程I/O
  - 直接内存访问
- 操作系统如何响应设备：中断
  - 中断优先级
  - 硬中断
  - 软中断
- 操作系统如何管理设备
  - 驱动程序
    - 驱动模型
  - 设备树

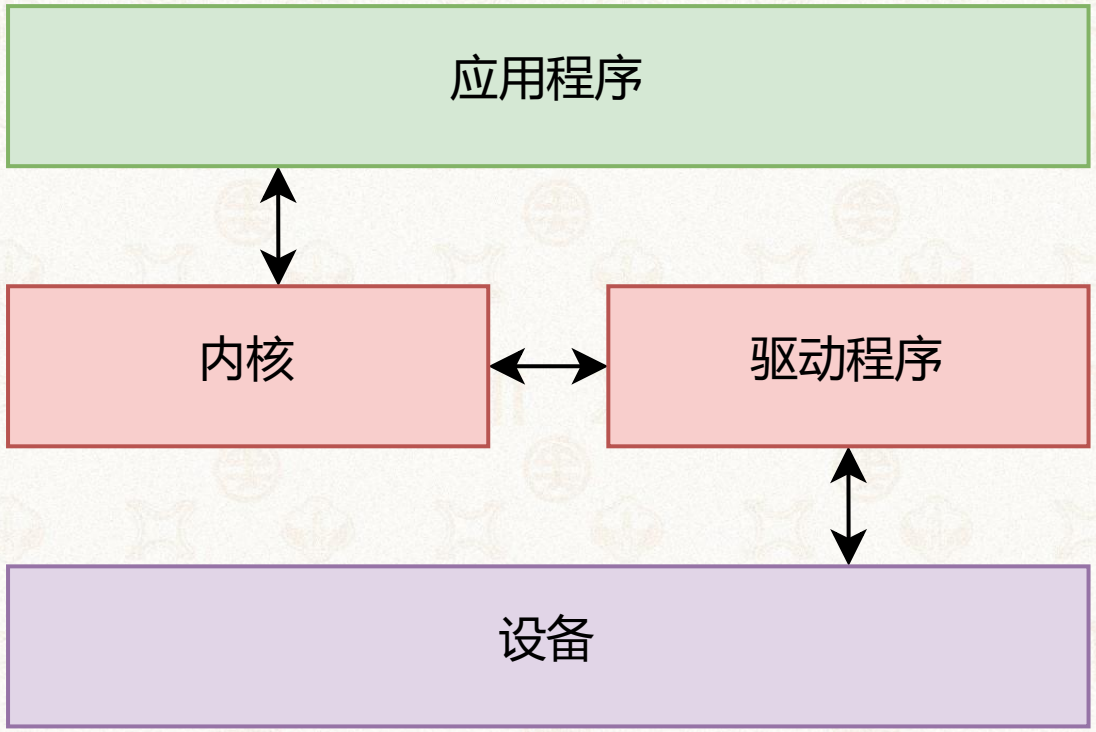
# 操作系统如何管理设备

- 设备的代码：
- 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”

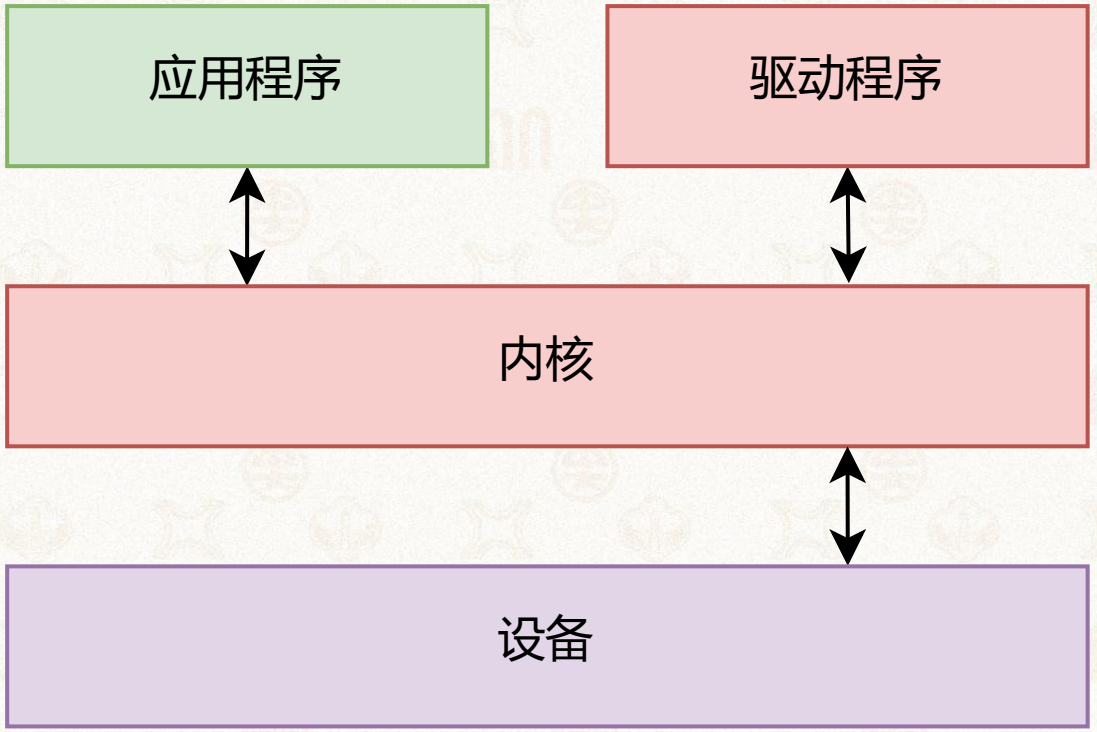


# 宏内核vs微内核的驱动

- 宏内核
  - 驱动在内核态
  - 优势：性能更好
  - 劣势：容错性差

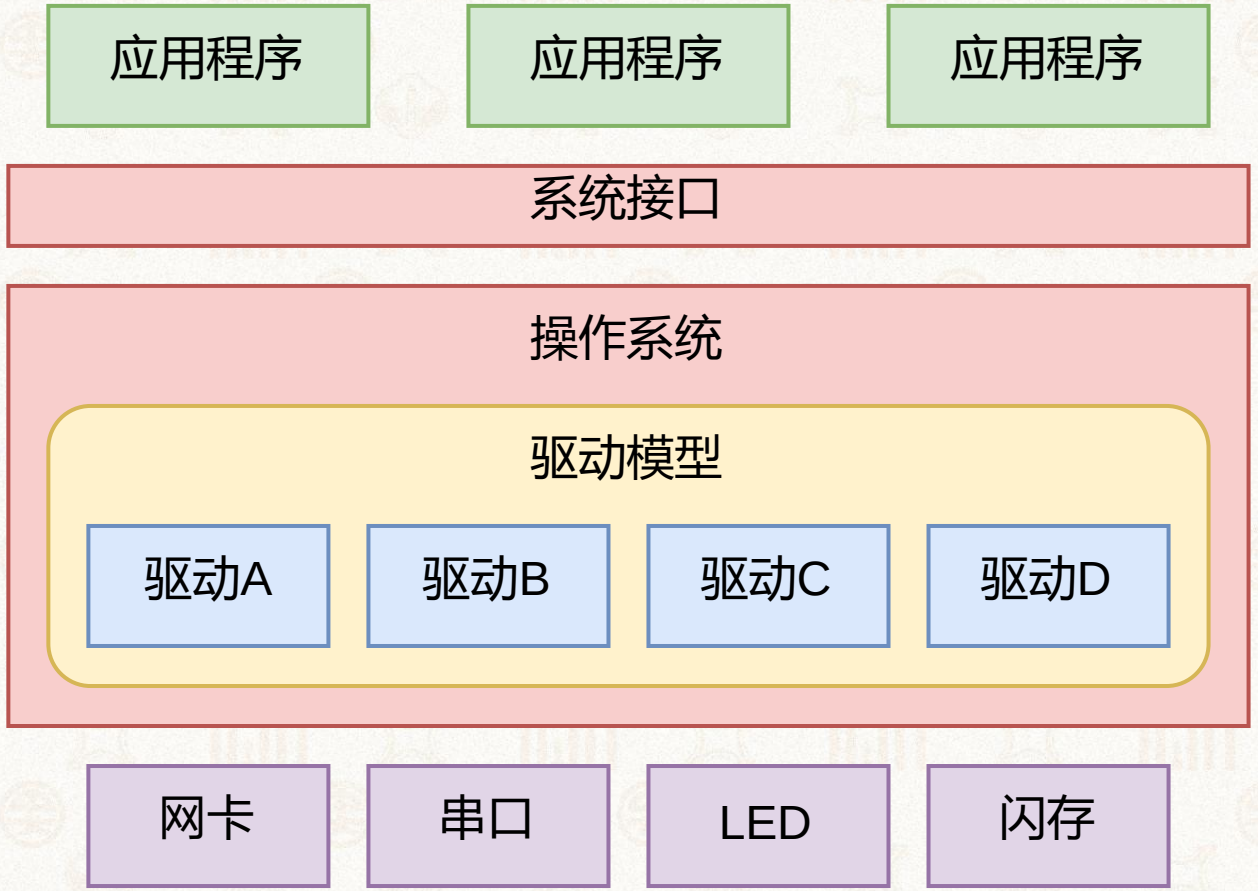



- 微内核
  - 驱动在用户态
  - 优势：可靠性好
  - 劣势：性能开销（IPC）



# 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



# 驱动程序：系统接口(ioctl)的示例

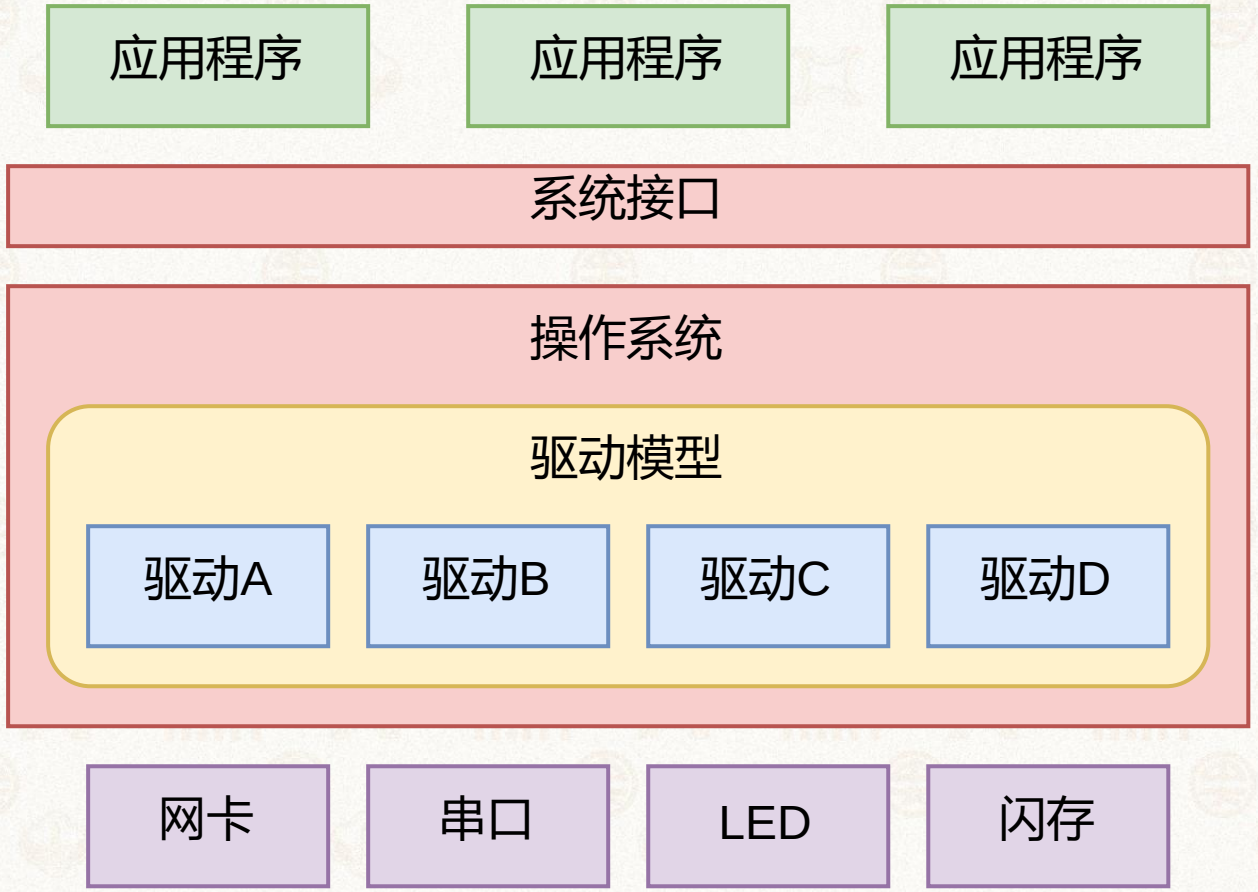
```
static int ioctl(struct tty_struct *tty, unsigned int cmd, unsigned long arg) {
    struct slgt_info *info = tty->driver_data;
    void __user *argp = (void __user *)arg; int ret;
    if (sanity_check(info, tty->name, "ioctl"))
        return -ENODEV;
    DBGINFO(("%s ioctl() cmd=%08X\n", info->device_name, cmd));
    switch (cmd) {
    case MGSL_IOCWAITEVENT:
        return wait_mgsl_event(info, argp);
    case TIOCMIWAIT:
        return modem_input_wait(info, (int)arg);
    case MGSL_IOCSGPIO:
        return set_gpio(info, argp);
    case MGSL_IOCGGPIO:
        return get_gpio(info, argp);
    case MGSL_IOCWAITGPIO:
        return wait_gpio(info, argp);
    case MGSL_IOCGXSYNC:
        return get_xsync(info, argp);
    // ...
    }
    mutex_lock(&info->port.mutex);
    // ...
}
```

可编程I/O

https://elixir.bootlin.com/linux/v5.16.14/source/drivers/tty/synclink_gt.c#L965

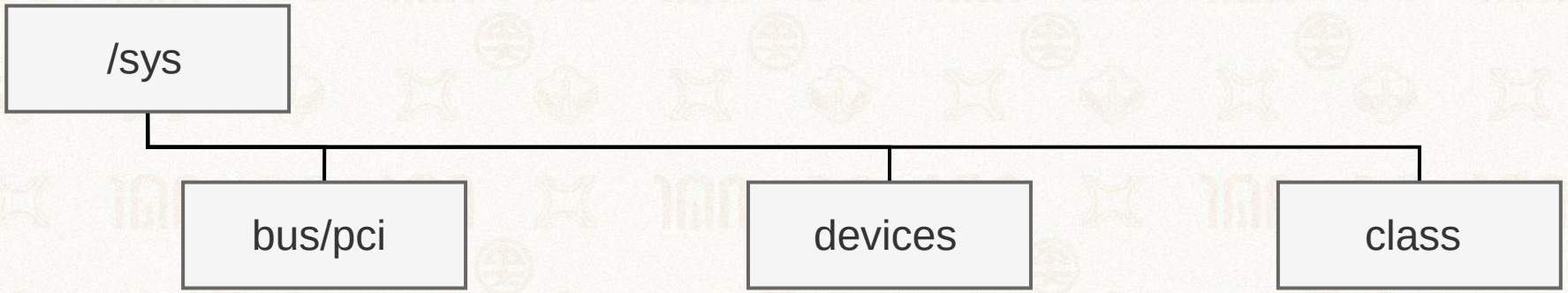
# 为什么需要驱动模型

- Linux在2.4以前没有驱动模型（一样可以用）
  - 2001年发布
- 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 (想想Java里面的接口)
  - 将驱动开发简化为对数据结构的填充和实现

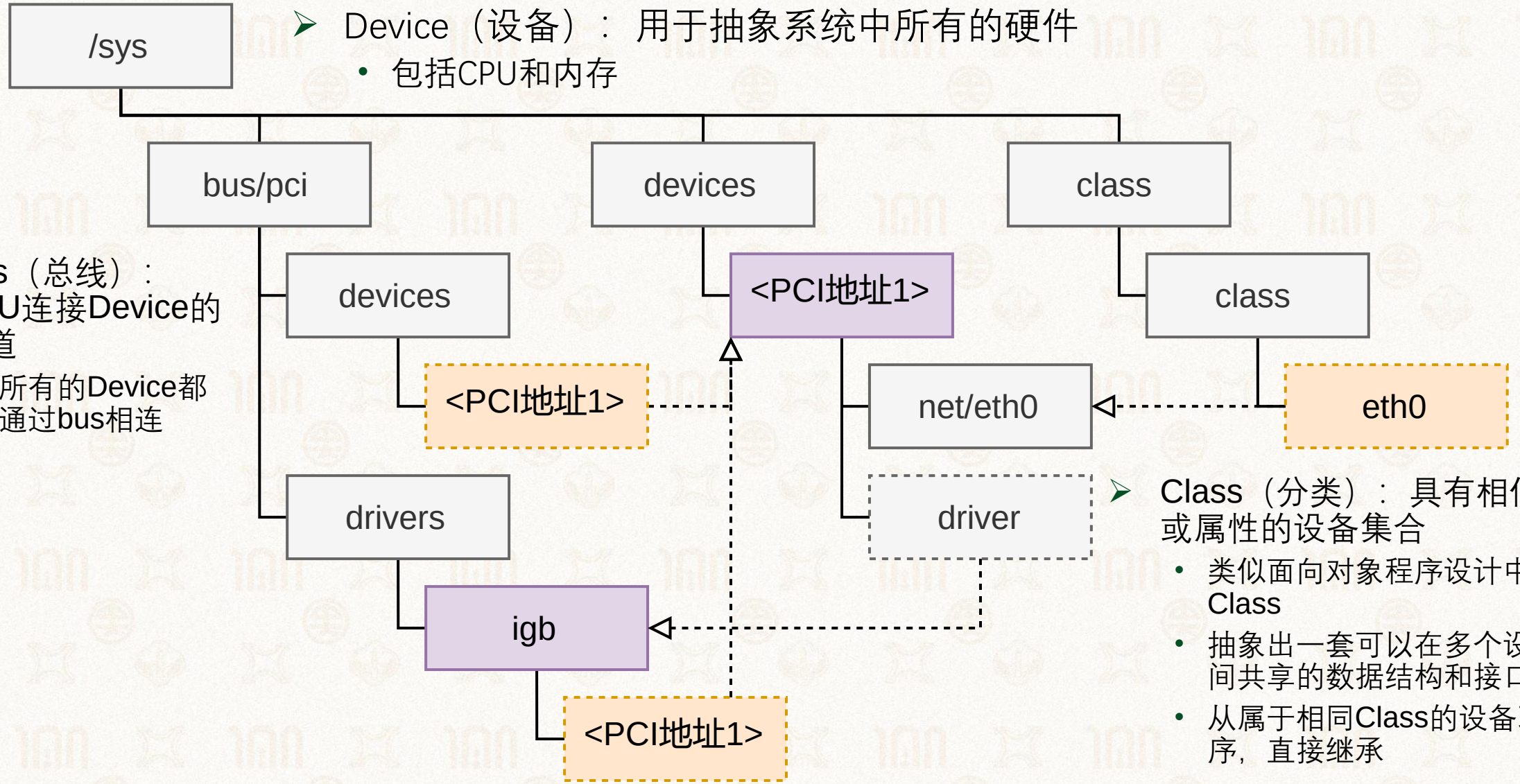


# 驱动模型：Linux Device Driver Model (LDDM)

- 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



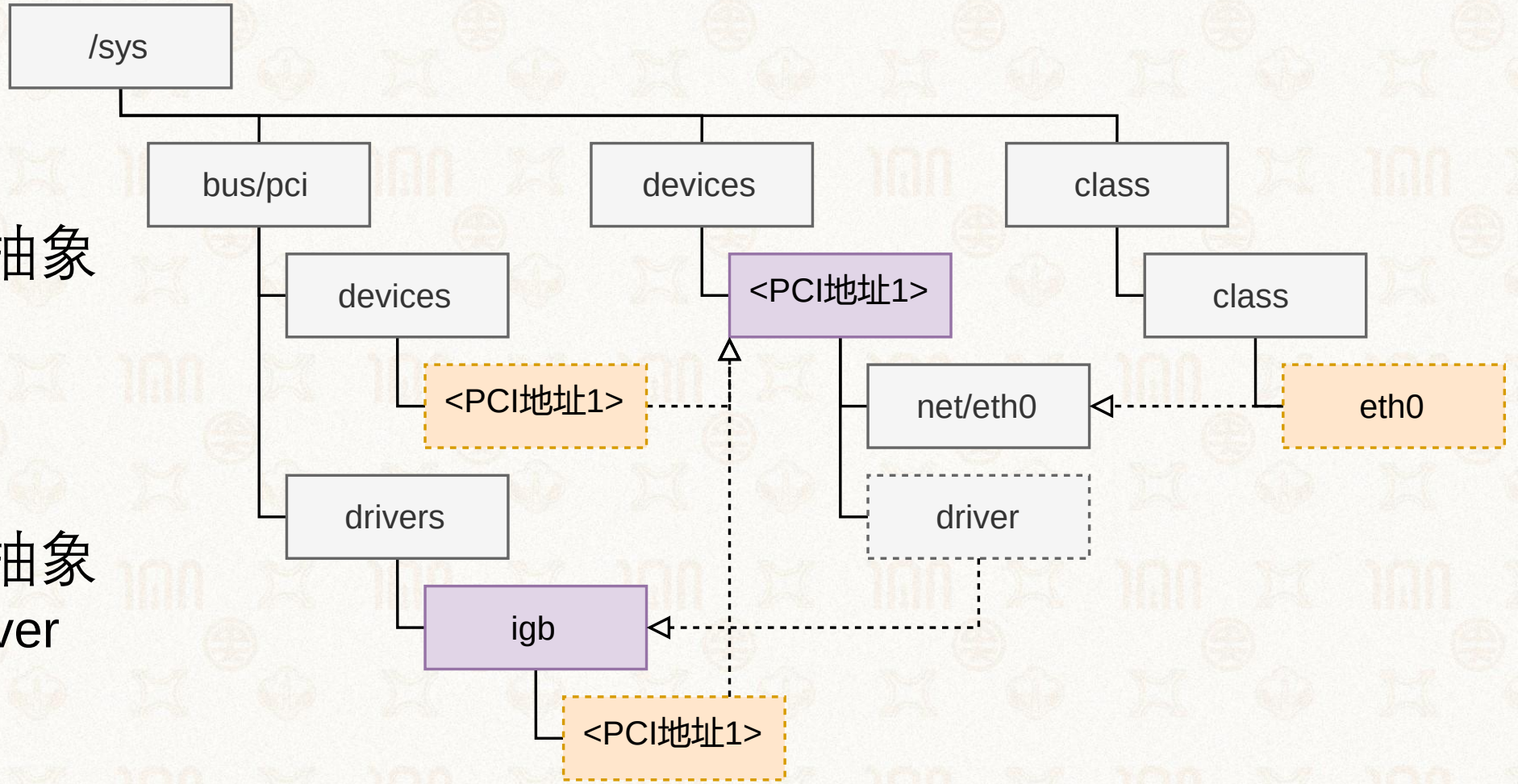
# Linux设备驱动抽象: sysfs




- 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;
    const char      *dev_name;
    struct device       *dev_root;
    const struct attribute_group **bus_groups;
    const struct attribute_group **dev_groups;
    const struct attribute_group **drv_groups;
    int (*match)(struct device *dev, struct device_driver *drv);
    int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
    int (*probe)(struct device *dev);
    void (*sync_state)(struct device *dev);
    void (*remove)(struct device *dev);
    void (*shutdown)(struct device *dev);
    int (*online)(struct device *dev);
    int (*offline)(struct device *dev);
    int (*suspend)(struct device *dev, pm_message_t state);
    int (*resume)(struct device *dev);
    int (*dma_configure)(struct device *dev);
    const struct dev_pm_ops *pm;
    const struct iommu_ops *iommu_ops;
    struct subsys_private *p;
    struct lock_class_key lock_key;
    bool need_parent_lock;
};
```

总线名称：PCI、USB、I2C、SPI

总线属性

添加设备或驱动到该总线时被调用

任何属于该bus的device发生变化时被调用

初始化，保证device所属的bus已被初始化

移除函数

若注册成功，新的总线将被添加进系统，可在 /sys/bus 下看到

```
extern int __must_check bus_register(struct bus_type *bus);

extern void bus_unregister(struct bus_type *bus);
```

https://elixir.bootlin.com/linux/v5.16.14/source/include/linux/device/bus.h#L82

# 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)`

https://elixir.bootlin.com/linux/v5.16.14/source/include/linux/device.h#L473

# 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 */
    bool suppress_bind_attrs;
    enum probe_type probe_type;
    const struct of_device_id   *of_match_table;
    const struct acpi_device_id *acpi_match_table;
    int (*probe) (struct device *dev); /* 热插：检测到设备  */
    void (*sync_state)(struct device *dev);
    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);

    const struct attribute_group **groups;
    const struct attribute_group **dev_groups;
    const struct dev_pm_ops *pm;
    void (*coredump) (struct device *dev);
    struct driver_private *p;
};
```

驱动的注册:

```
int driver_register(struct device_driver *drv);

void driver_unregister(struct device_driver *drv);
```

https://elixir.bootlin.com/linux/v5.16.14/source/include/linux/device/driver.h#L96

# LDDM特点

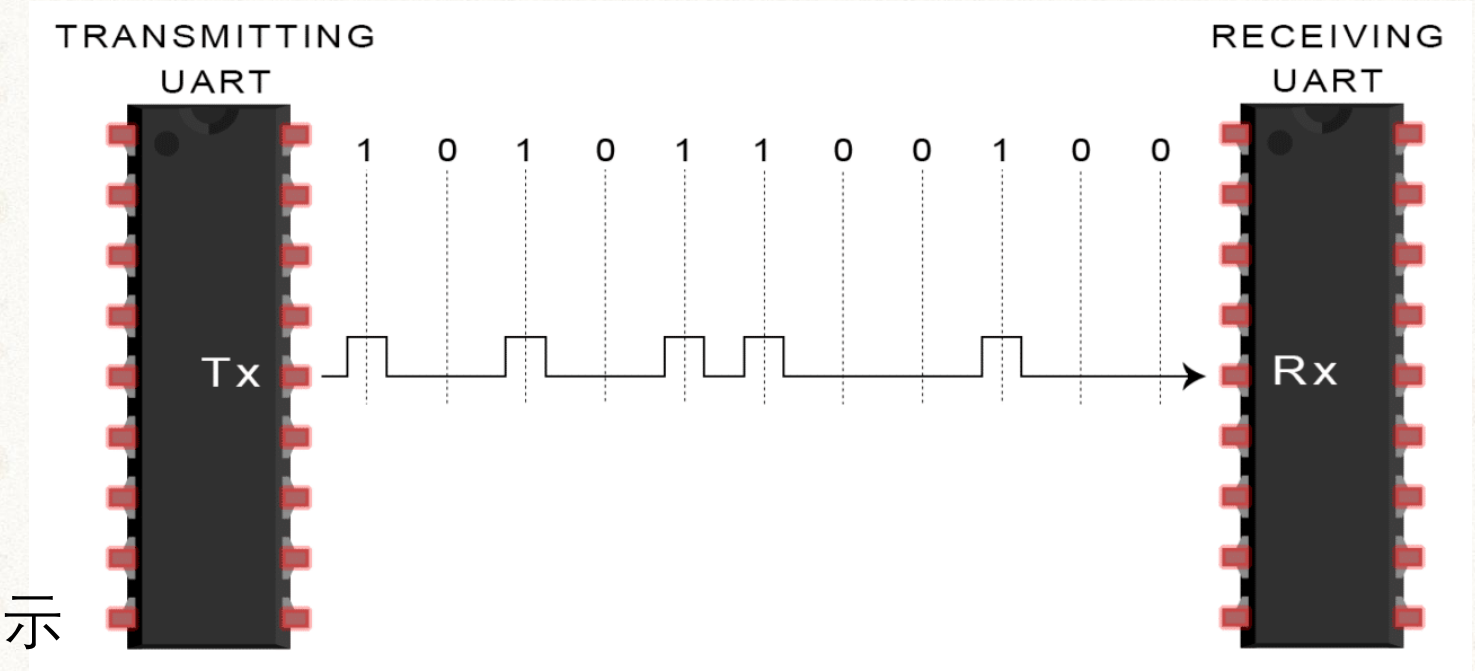
- device_driver和device都注册到bus上
- bus_type的match()
  - 如果设备与驱动相匹配，将调用device_driver的probe()，交给device_driver来完成余下工作
- device_driver的probe()
  - 驱动程序的入口，probe成功后内核生成设备实例，驱动注册的file_operations可以被应用程序所访问
- device为bus_type 、device_driver父类的多继承
  - 要实例化一个device，先实例化父类driver和bus

# 大纲

- 设备连接
- 设备类型抽象
  - 字符设备
  - 块设备
  - 网络设备
- 设备与操作系统的交互
  - 可编程I/O
  - 直接内存访问
- 操作系统如何响应设备：中断
  - 中断优先级
  - 硬中断
  - 软中断
- 操作系统如何管理设备
  - 驱动程序
    - 驱动模型
  - 设备树

# 问题：如何描述一个设备？

- 以串口为例：
  - 名称： uart0
  - 波特率： 115200




用C语言的结构体进行表示

```c
struct uart_struct {
    const char *name;
    unsigned baudrate;
}

uart = {
    .name = "uart0",
    .baudrate = 115200,
};
```

用JSON结构表示

```json
{
    "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

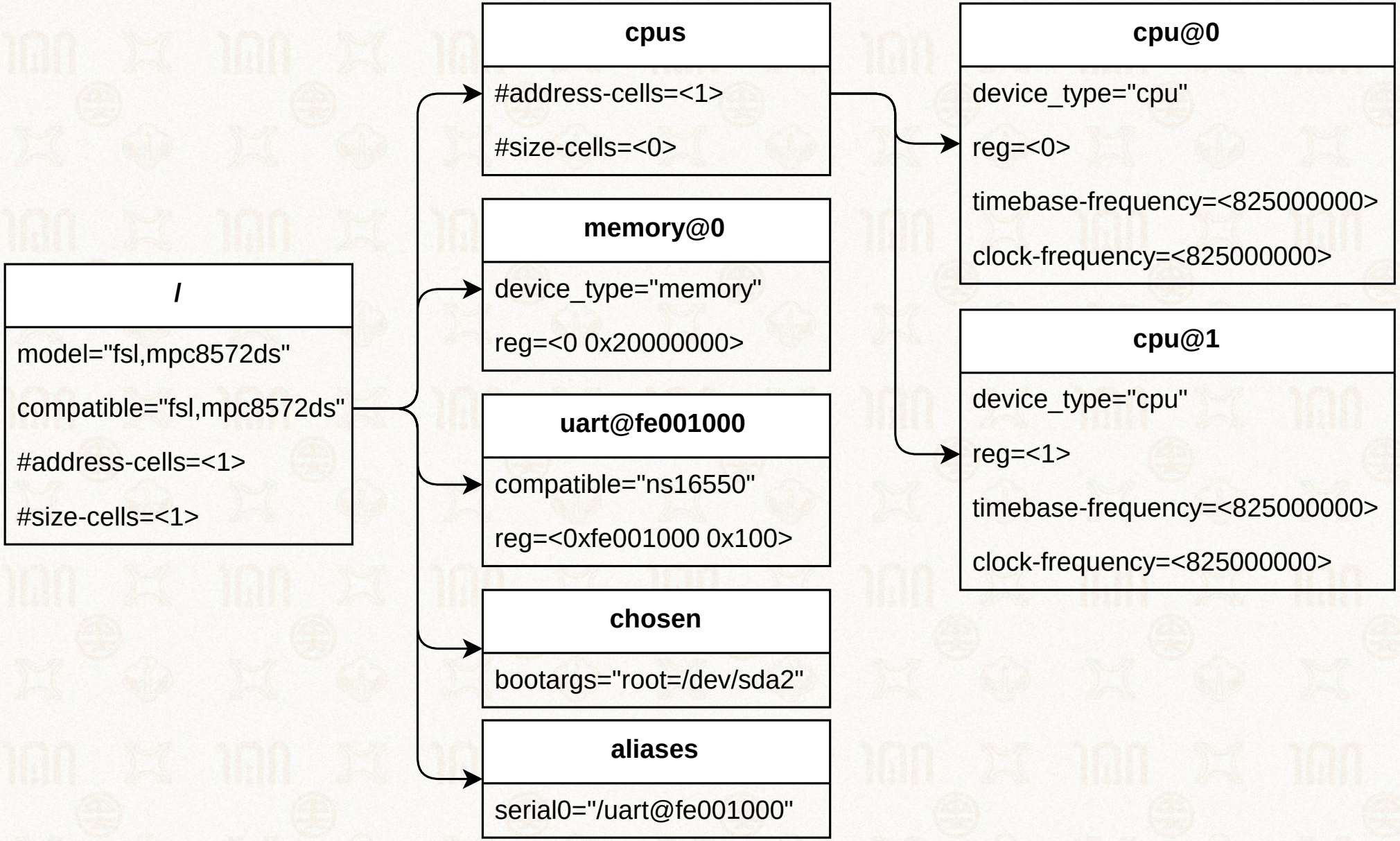
- 硬件信息可通过 Device Tree Source (DTS) 传递给内核
  - 避免在内核中编码大量硬件细节
  - 冗余的硬件信息可以包含和引用

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

属性名　　属性值

https://www.devicetree.org/

# 设备树模型


/
model="fsl,mpc8572ds"
compatible="fsl,mpc8572ds"
#address-cells=<1>
#size-cells=<1>
cpus
#address-cells=<1>
#size-cells=<0>
memory@0
device_type="memory"
reg=<0 0x20000000>
uart@fe001000
compatible="ns16550"
reg=<0xfe001000 0x100>
chosen
bootargs="root=/dev/sda2"
aliases
serial0="/uart@fe001000"
cpu@0
device_type="cpu"
reg=<0>
timebase-frequency=<825000000>
clock-frequency=<825000000>
cpu@1
device_type="cpu"
reg=<1>
timebase-frequency=<825000000>
clock-frequency=<825000000>

# 真实的DTS

```
/ {
    usart2: serial@40004400 {
        compatible = "st,stm32-uart";
        reg = <0x40004400 0x400>;
        interrupts = <38>;
        clocks = <&rcc 0 STM32F4_APB1_CLOCK(UART2)>;
        status = "disabled";
    };

    usart3: serial@40004800 {
        compatible = "st,stm32f7-uart";
        reg = <0x40004800 0x400>;
        interrupts = <39>;
        clocks = <&rcc 1 CLK_USART3>;
        status = "disabled";
    };
};
```

https://elixir.bootlin.com/linux/v5.16.14/source/arch/arm/boot/dts/stm32f746.dtsi#L307

# DTS的声明

```
/ {                                   节点名@地址
根节点   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];                    二进制值
            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{};
        };
};
```

# 驱动对DTS的使用

- 通过compatible属性进行匹配

```
static const struct of_device_id stm32_match[] = {
    { .compatible = "st,stm32-uart", .data = &stm32f4_info},
    { .compatible = "st,stm32f7-uart", .data = &stm32f7_info},
    { .compatible = "st,stm32h7-uart", .data = &stm32h7_info},
    {},
};

static struct platform_driver stm32_serial_driver = {
    .probe      = stm32_usart_serial_probe,
    .remove     = stm32_usart_serial_remove,
    .driver = {
        .name   = DRIVER_NAME,
        .pm = &stm32_serial_pm_ops,
        .of_match_table = of_match_ptr(stm32_match),
    },
};
```

https://elixir.bootlin.com/linux/v5.16.14/source/drivers/tty/serial/stm32-usart.c#L1367
https://elixir.bootlin.com/linux/v5.16.14/source/drivers/tty/serial/stm32-usart.c#L1833

# 驱动对DTS的使用

```
static int __init stm32_usart_init(void) {
    static char banner[] __initdata = "STM32 USART driver initialized";
    int ret;

    pr_info("%s\n", banner);

    ret = uart_register_driver(&stm32_usart_driver);
    if (ret)
        return ret;

    ret = platform_driver_register(&stm32_serial_driver);
    if (ret)
        uart_unregister_driver(&stm32_usart_driver);

    return ret;
}

module_init(stm32_usart_init);
module_exit(stm32_usart_exit);
```

驱动是以模块的形式动态加载

https://elixir.bootlin.com/linux/v5.16.14/source/drivers/tty/serial/stm32-usart.c#L1843

# 大纲

- 设备连接
- 设备类型抽象
  - 字符设备
  - 块设备
  - 网络设备
- 设备与操作系统的交互
  - 可编程I/O
  - 直接内存访问
- 操作系统如何响应设备：中断
  - 中断优先级
  - 硬中断
  - 软中断
- 操作系统如何管理设备
  - 驱动程序
    - 驱动模型
  - 设备树

# 谢谢

微信：suyuxin
钉钉：苏玉鑫
B站：https://space.bilibili.com/502854403
软工集市课程专区：https://ssemarket.cn/new/course
匿名提问箱：https://suask.me/ask-teacher/106/苏玉鑫