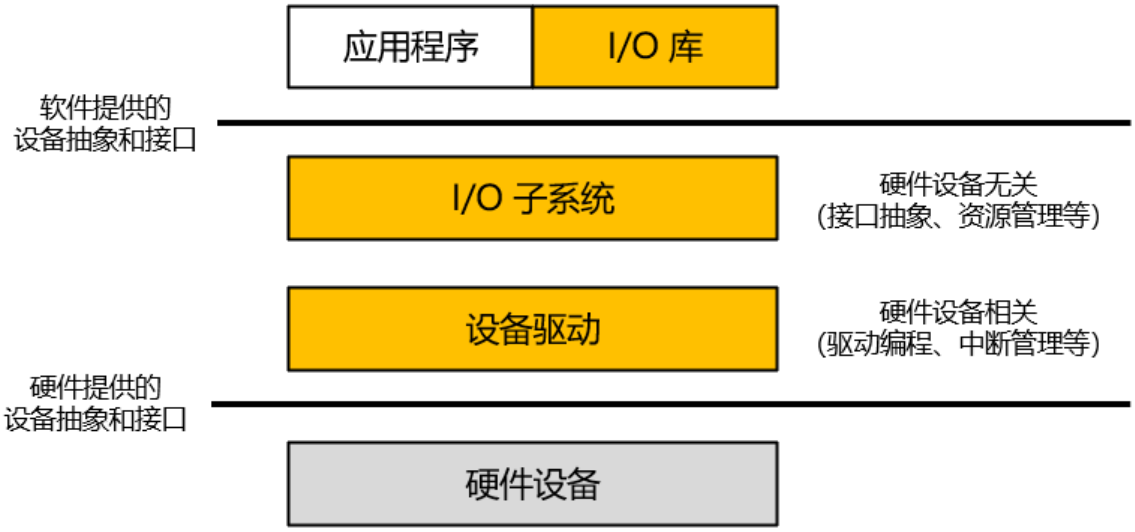
# Lecture18 Device

> 操作系统的I/O层次




## 1. 认知设备

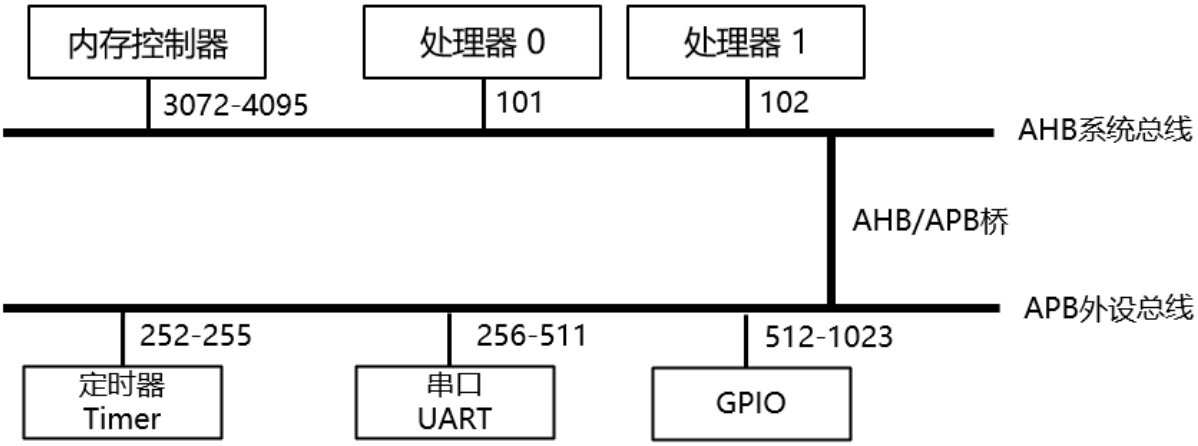
- Raspiberry Pi 3 Model B+
- GPIO LED

  > General Purpose I/O

- 8042(PS/2键盘控制器)
- UART（串口）
- Flash闪存
- Ethernet网卡

## 2. 设备与CPU的连接

### 硬件总线：以AMBA为例




> AHB：Advanced High-performance Bus
>
> APB：Advanced Peripheral Bus
>
> 它们在同一个地址空间中

## 硬件总线的特点

- 一组电线
  - 将各个I/O模块连接到一起，包含了地址总线、数据总线和控制总线
- 使用广播
  - 每个模块都能收到消息
  - 总线地址：表示了预期的接收方
- 仲裁协议
  - 决定哪个模块可以在什么时间收发消息
  - 总线仲裁器：用于选择哪些模块可以使用该总线

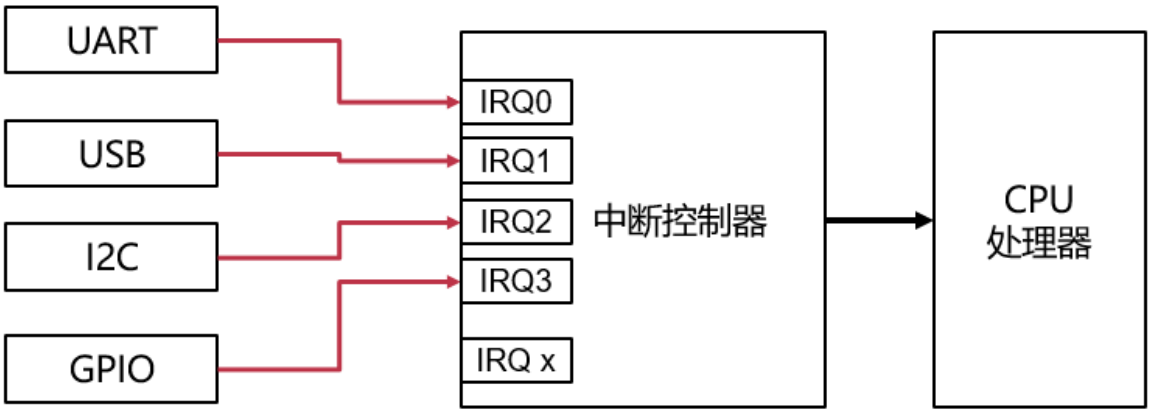
## 同步 VS. 异步

- 同步数据传输
  - 源和目标借助**共享时钟**进行协作
- 异步数据传输
  - 源和目标借助**显示信号**进行协作

## 总线事务

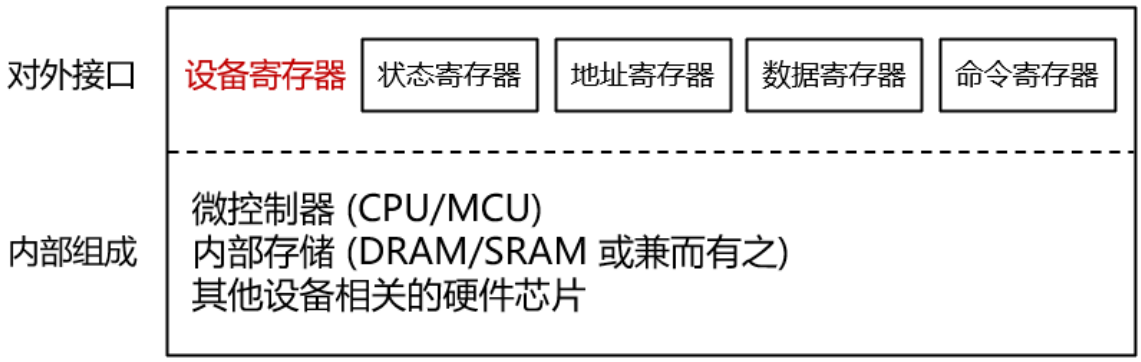
1. 源获取总线的使用权（具有排他性）
2. 源将目标的地址写到总线上
3. 源发出READY信号，提醒其它模块（广播）
4. 目标在拷贝数据后，发出ACK信号
   - 同步模式下，无需READY和ACK，只要在每个时钟周期进行检查即可
5. 源释放总线

### 中断线

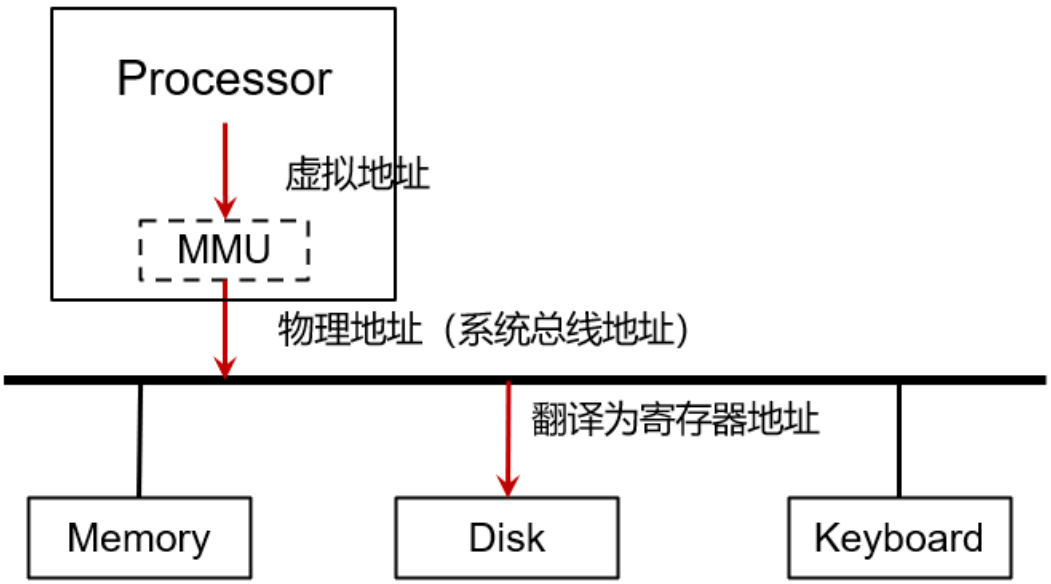



## 3. 设备与CPU的交互

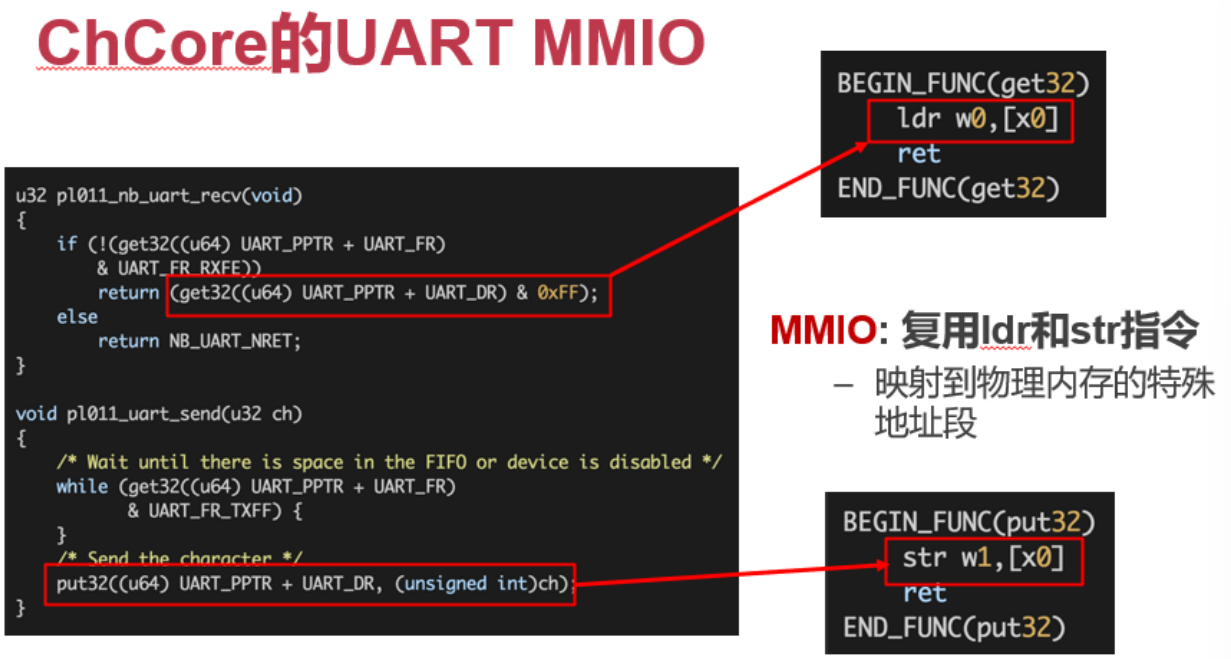
### 硬件设备的接口：设备寄存器




### 内存映射I/O(MMIO)



> 大部分是Memory的地址，但也有少部分是Disk和Keyboard等外设的虚拟地址




- **MMIO地址应使用Volatile关键字**

```
void main(void)
{
    void          *pdev = (void *) 0x40400000;
    size_t         size = (1024*1024);
    int           *base;
    volatile int  *pcid, cid;

    base = mmap(pdev, size, PROT_READ|PROT_WRITE,
                MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
    if (base == MAP_FAILED) errx(1, "mmap failure");

    pcid = (int *) (((void *) base) + 0xf0704);
    cid = *pcid;
    printf("cid = %d\n", cid);
    cid = *pcid;
    printf("cid = %d\n", cid);

    munmap(base, size);
}
```

若不加volatile，编译器会认为这两个printf()多余，并消除第二个内存加载操作

## 可编程I/O(Programmable I/O)

- 形式1：MMIO(Memory-mapped I/O)

  > 代价是：用volatile和non-cachable避免误判

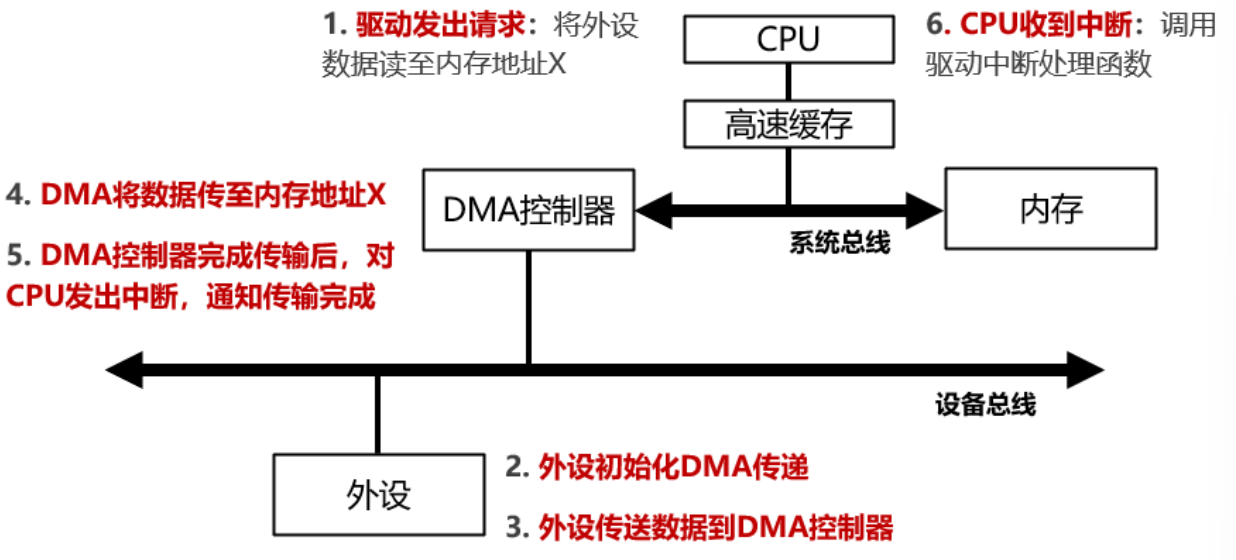
  - 将设备映射到连续物理内存中
  - 使用内存访问指令(load/store)
  - 行为与内存不完全一样，读写有**副作用**(需要volatile)
  - 在ARM，RISC-V等架构中使用
- 形式2：PIO(Port I/O)
  - I/O设备具有独立的地址空间

- 使用专门的PIO指令(in/out)
- 在x86架构中使用

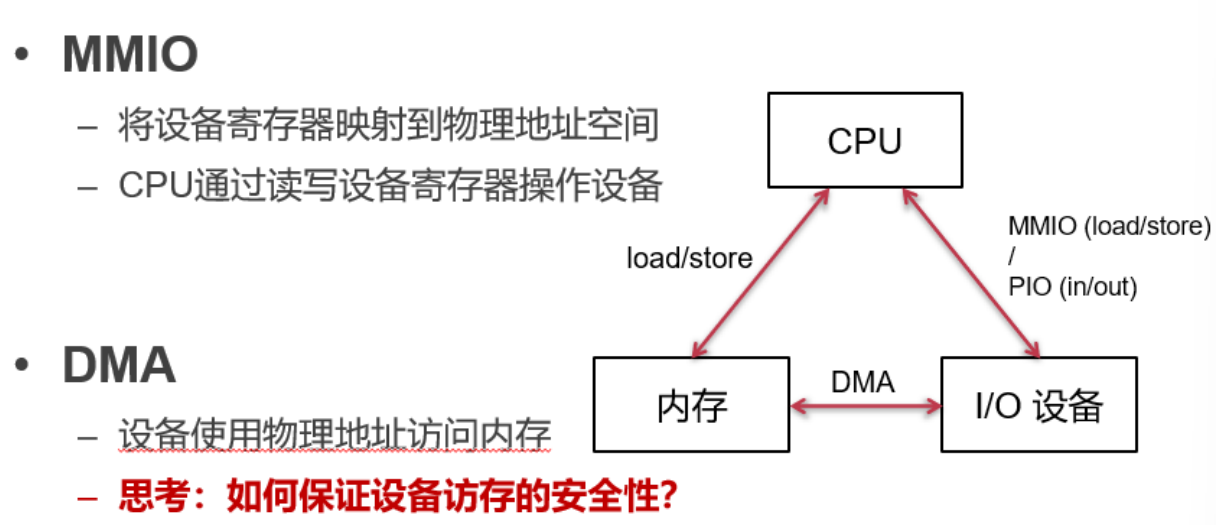
## 是否存在更高效的数据交互方式?

- 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
  - 适合于简单小型设备
- **直接内存访问(Direct Memory Access, DMA)**
  - 设备可直接访问总线
  - DMA与内存互相传输数据，传输不需要CPU参与
  - 适合高吞吐量I/O

## DMA

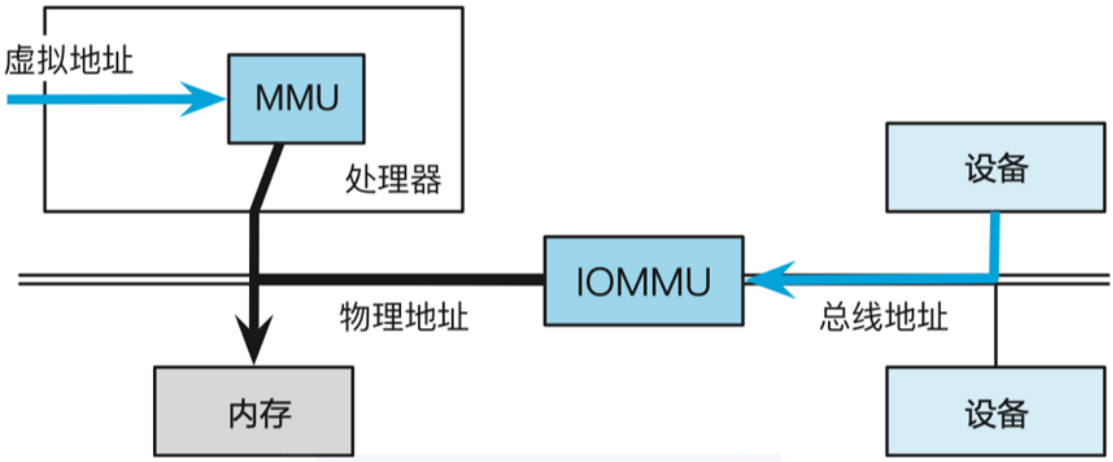



## CPU访问设备方式小结



> 解决方案：IOMMU

- 避免设备直接使用物理地址访问内存
  - 设备所使用的地址，由IOMMU翻译为实际的物理地址
  - 广泛应用于虚拟机场景中（允许虚拟机独占某个设备）




> 在总线上添加一个IOMMU

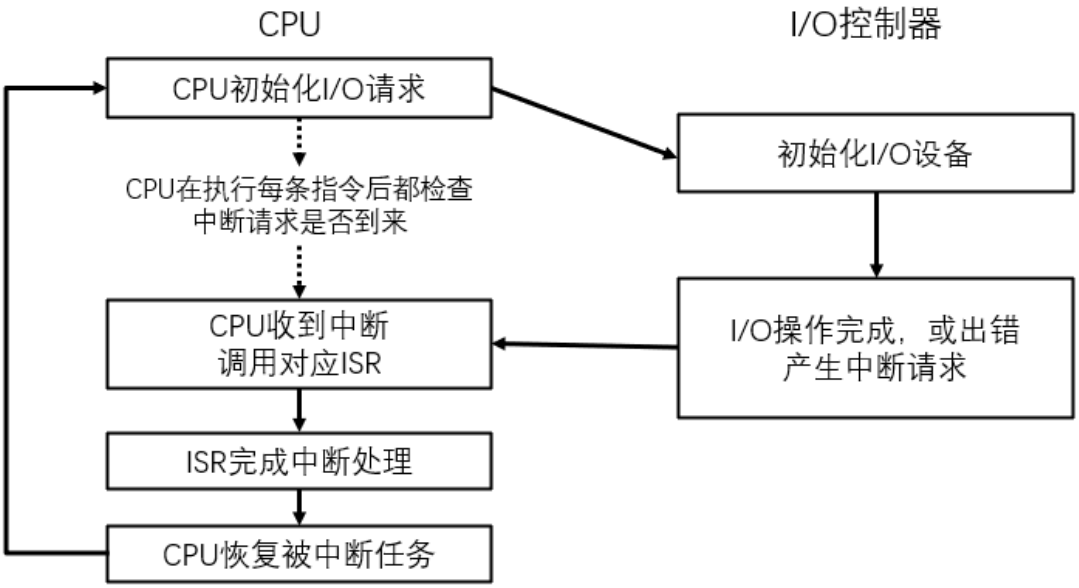
## DMA的内存一致性

- 现代CPU通常有高速缓存（CPU Cache）
- 当DMA发生时，DMA缓冲区的数据仍然在cache中怎么办?
- 解决方法:
  - 方案1：将DMA区域映射为non-cachable
  - 方案2：由软件负责维护一致性，软件主动刷缓存
  - 部分架构在硬件上保证了DMA一致性，如总线监视技术

# 4. 中断与中断响应

> 中断存在的原因是：CPU太快，硬件太慢

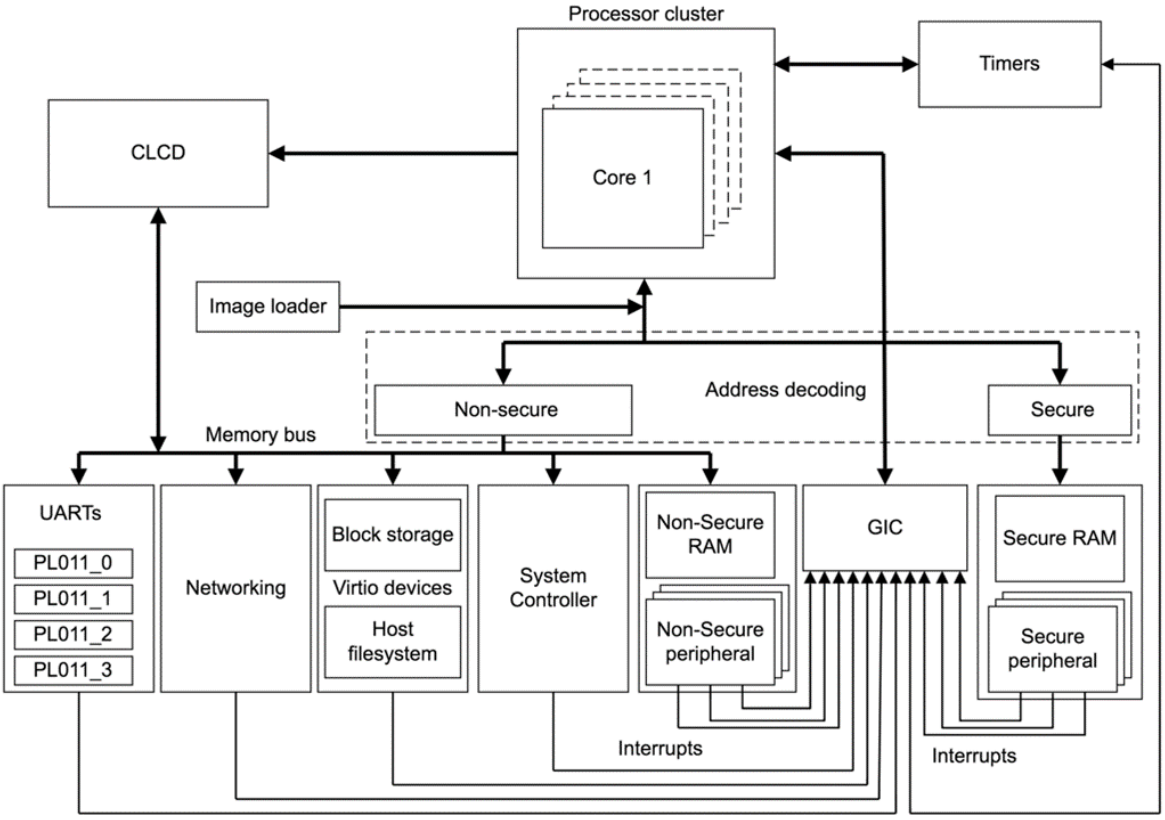
## CPU中断处理流程




## AArch64的中断分类

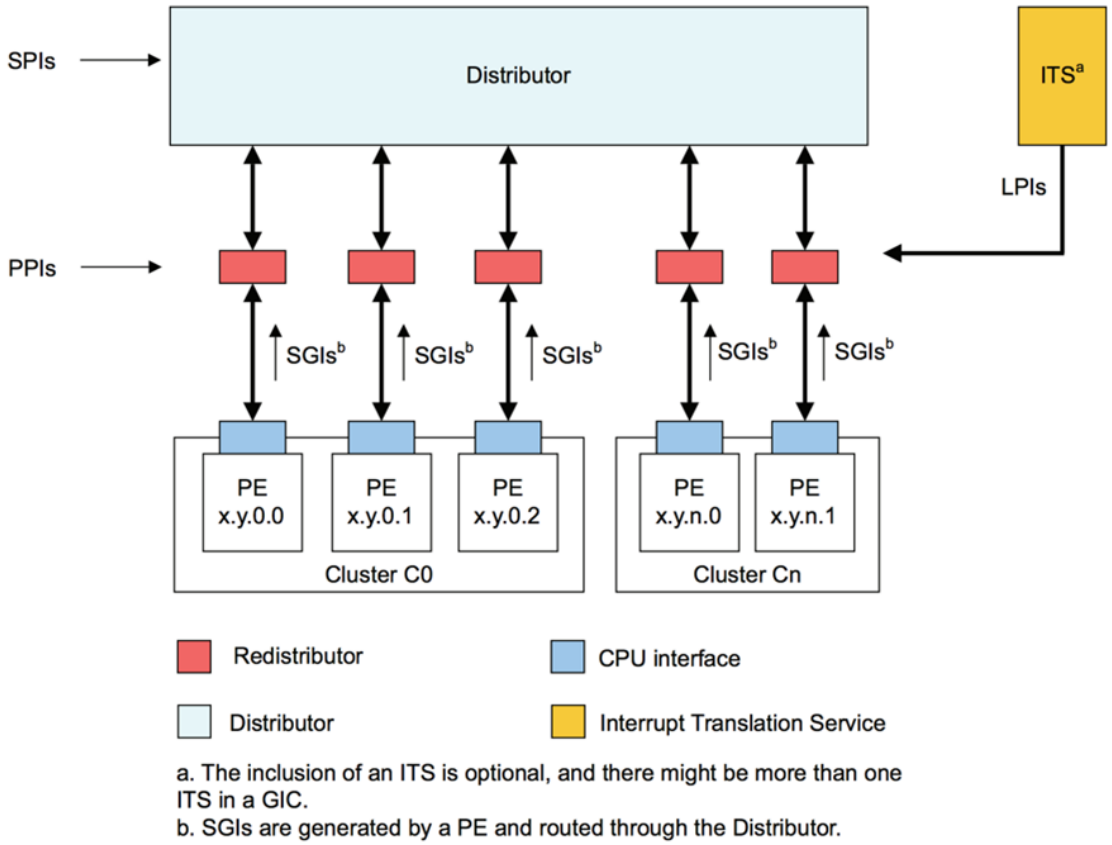
- IRQ(Interrupt Request)
  - 普通中断，优先级低，处理慢
- FIQ(Fast Interrupt Request)
  - 一次只能由一个FIQ
  - 快速中断，优先级高，处理快
  - 常为可信任的中断源预留

## ARM中断控制器-GIC




> GIC：Generic Interrupt Controller

- 组件1：Distributor
  - 负责全局中断的分发和管理
- 组件2：CPU Interface
  - 类似于“门卫”，判断中断是否要发给CPU处理




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**

# GIC 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
  - 中断触发方式
  - 中断的目的core

# GIC: CPU Interface

- **CPU接口：**
  - 将GICD发送的中断，通过IRQ中断线发给连接到 interface 的核心
- **寄存器：GICC**
- **作用：**
  - 将中断请求发给CPU
  - 配置中断屏蔽
  - 中断确认（acknowledging an interrupt）
  - 中断完成（indicating completion of an interrupt）
  - 核间中断（Inter-Processor Interrupt，IPI），用于核间通信

## 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，实现优先级配置
6. Deactivate：core写GICC_DIR寄存器，来无效该中断

## 多个中断

> 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 |

- 在处理当前中断（ISR）时：

  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本身不可抢占
- 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：只停止对应IRQ的请求
    - 屏蔽策略：
        - 屏蔽全局中断：
            1. 系统关键步骤（原子性）
            2. 保证任务相应的实时性
        - 屏蔽对应中断：通常是这种情况，对系统的影响整体最小

## 高频中断的问题：活锁

- 网络场景下的中断使用（网卡设备）
    - 当每个网络包到来时都发送中断请求时，OS可能进入活锁
    - **活锁**：CPU只顾着响应中断，无法调度用户进程和处理中断发来的数据
- **解决方案：合二为一（中断+轮询），兼顾各方优势**
    - 默认使用中断
    - 网络中断发生后，使用轮询处理后续到达的网络包
    - 如果没有更多中断，或轮询中断超过时间限制，则回到中断模式
    - 该方案在Linux网络驱动中成为NAPI(New API)

## 中断合并(Interrupt Coalescing)

- 中断合并
    - 设备在发送中断前，需要等待一小段时间
    - 在等待期间，其他中断可能也会马上到来，因此将多个中断合并为一个中断，进而降低频繁中断带来的开销
- 注意：
    - 等待过长时间会导致中断响应时延增加
    - 这是系统中常见的“折衷”(trade-off)

# 5. 案例：Linux的上下半部

- 面临的问题
    - 中断处理过程中若运行复杂逻辑，会导致系统失去响应更久
    - 中断处理时不能调用会导致系统block的函数
- 将中断处理分为两部分
    - 上半部：尽量快，提高对外设的响应能力

- 下半部：将中断的处理推迟完成

## Top Half：马上做

- 最小化公共例程：
  - 保存寄存器、屏蔽中断
  - 恢复寄存器，返回现场
- Top half要做的事情：
  - 将请求放入队列（或设置flag），将其他处理推迟到bottom half
    - 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)可以绑定在同一向量上
  - 调用每个设备对应的IRQ的ISR

## Bottom Half：延迟去做

- 提供可以推迟完成的机制
  - 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


| 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 |

| 特点 | ISR | SoftIRQ | Tasklet | WorkQueue | KThread |
| --- | --- | --- | --- | --- | --- |
| 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 |

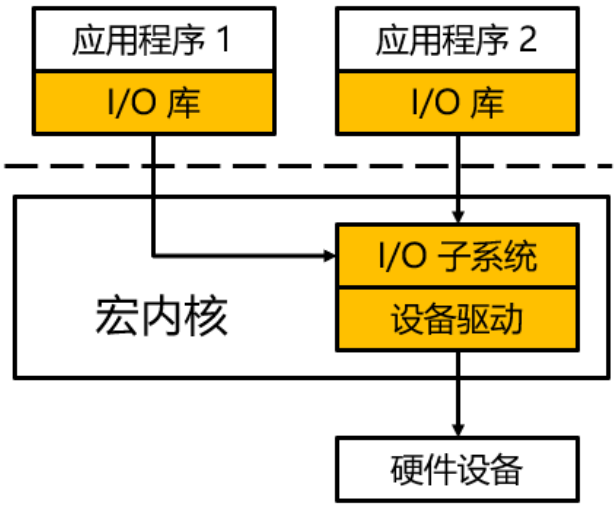
# 6. 设备驱动

## 设备驱动

- 设备驱动
  - 专门用于操作硬件设备的代码集合
  - 通常由硬件制造商负责提供
  - 驱动程序包含中断处理程序
- 驱动特点
  - 和设备功能高度相关
  - 不同设备间的驱动复杂度差异巨大
  - 是操作系统bugs的主要来源

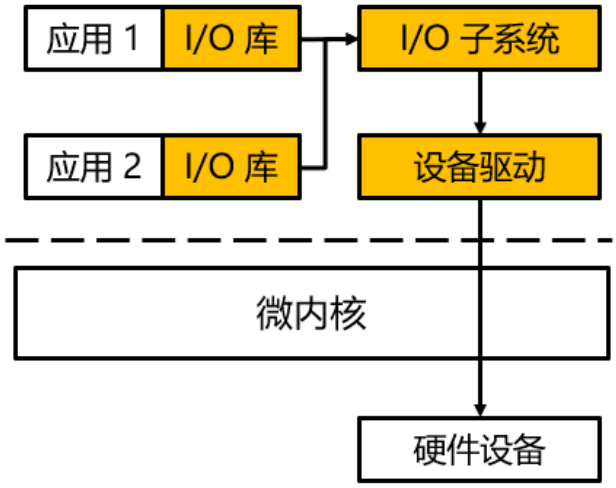
## 宏内核I/O架构

- **宏内核I/O架构**
  - 设备驱动在内核态
  - 优势：通常性能更好
  - 劣势：容错性差
  - 中断形式为内核ISR
- **案例：**
  - Linux、BSD
  - Windows



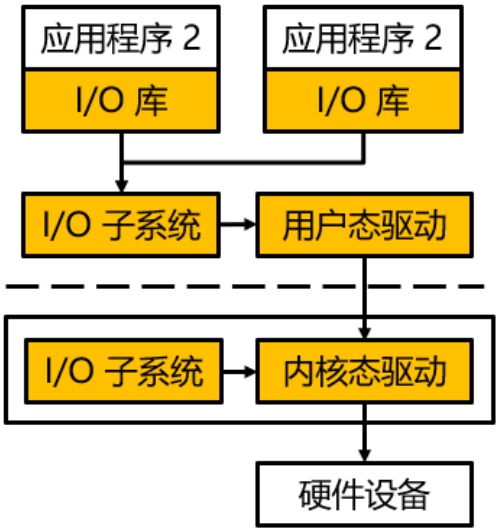
## 微内核I/O架构

- **微内核I/O架构**
  - 设备驱动主体在用户态
  - 优势：可靠性和容错性更好
  - 劣势：IPC性能开销
  - 中断为用户态驱动线程
- **案例：**
  - 谷歌Fuchsia手机系统
  - ChCore微内核系统

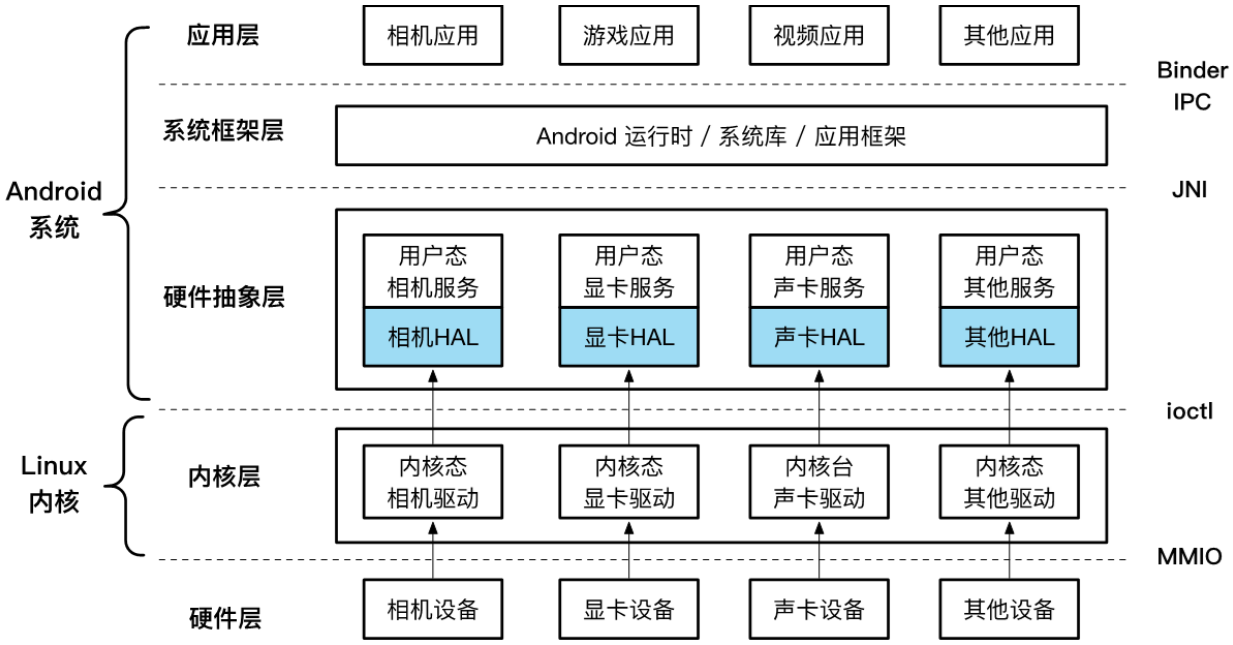



## 混合I/O架构

- **混合I/O架构**
  - 设备驱动分解为用户态和内核态
  - 优势1：驱动开发和Linux内核解耦
  - 优势2：允许驱动以闭源形式存在
    保护硬件厂商的知识产权
- **案例：**
  - 谷歌安卓系统：硬件抽象层（HAL）
  - 华为鸿蒙系统：硬件驱动框架（HDF）




## 案例：安卓的硬件抽象层




> HAL理论上可以不用开源

## 设备驱动的复杂性在提高

- 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，复杂度提高
- 驱动开发者的需求：
  - 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

## 驱动模型的好处

- **电源管理：**
  - 描述设备在系统中的拓扑结构（树形结构）
  - 保证能正确控制设备的电源，先关闭设备和再关闭总线
- **驱动开发者：**
  - 允许同系列设备的驱动代码之间的复用
  - 将设备和驱动联系起来，方便相互索引
- **系统管理员：**
  - 帮助用户枚举系统设备，观察设备间拓扑和设备的工作状态

### 案例：Linux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**
- **维护内核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
  - 启动设备时会自动初始化依赖的对象，直到启动条件满足为止

## 7. I/O子系统

### I/O子系统的目标

- 提供统一接口，涵盖不同设备
  - 如下代码对各种设备通用：

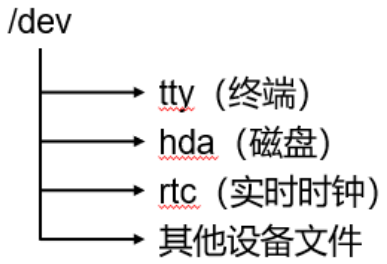
```
FILE fd = fopen("/dev/something","rw");
for (int i = 0; i < 10; i++) {
        fprintf(fd,"Count %d\n",i);
}
close(fd);
```

- 满足I/O硬件管理的共同需求，提供统一抽象
  - 管理硬件资源，隐藏硬件细节

## 统一抽象——设备文件

- 为应用程序提供的相同的设备抽象：设备文件
- 操作系统将外设细节和协议封装在文件接口的内部
- 复用文件系统接口：open(), read(), write(), close, etc.

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




## 设备操作的专用接口：ioctl

- 应用程序和驱动程序事先协商好“操作码”和对应语义

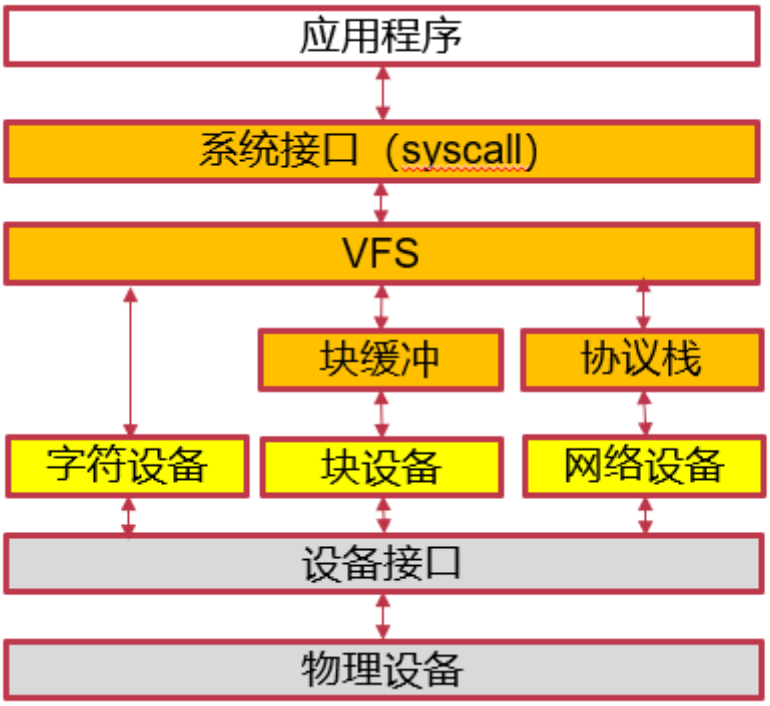
```
#define LED_ALL_ON    _IO('L',0x1234)
#define LED_ALL_OFF   _IO('L',0x5678)

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;
}
```

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

long led_ioctl(..., unsigned int cmd, ...)
{
    switch (cmd) {
        case LED_ALL_ON:
            *gpco_data |= 0x3<<3; break;
        case LED_ALL_OFF:
            *gpco_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,
};
```

## 设备的逻辑分类

- Linux的设备分类

1. 字符设备
2. 块设备
3. 网络设备




**字符设备(cdev)**

- **例子：**
  - 键盘、鼠标、串口、打印机等
  - 大多数伪设备：/dev/null, /dev/zero, /dev/random
- **访问模式：**
  - **顺序访问**，每次读取一个字符
  - 调用驱动程序和设备直接交互
- **文件抽象：**
  - open(), read(), write(), close()

**块设备(blkdev)**

- **例子：**
  - 磁盘、U盘、闪存等（以存储设备为主）
- **访问模式：**
  - **随机访问**，以块为单位进行寻址（如512B、4KB不等）
  - 通常为块设备增加一层缓冲，避免频繁读写I/O导致的慢速
- **通常使用内存抽象：**
  - 内存映射文件(Memory-Mapped File)：mmap()访问块设备
  - 提供文件形式接口和原始I/O接口（绕过缓冲）

#### 网络设备(netdev)

- 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
  - 在驱动层之上维护多种协议，支持不同策略
- 套接字抽象
  - socket(), send(), recv(), close(), etc

#### 设备逻辑分类小结

- **设备分类：**
  - 字符设备（cdev）：键盘、鼠标、串口、打印机等
  - 块设备（blkdev）：磁盘、U盘、闪存等存储设备
  - 网络设备（netdev）：以太网、WiFi、蓝牙等通信设备
- **设备接口：**
  - 字符设备： read(), write()
  - 块设备： read(), write(), lseek(), mmap()
  - 网络设备：socket(), send(), recv()
    - 同时兼容文件接口（也可以用read(), write()读写socket）

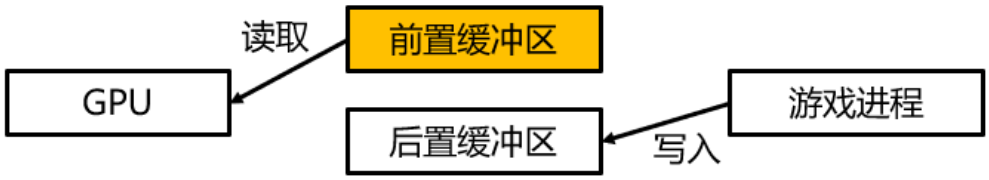
## 设备的缓冲管理

#### 单缓冲区

- **问题：**
  - 读写性能不匹配：慢速的存储设备 vs. 高速的CPU
  - 读写粒度不匹配：小数据的访问存在读写放大的问题
- **解决方法：**
  - 开辟内存缓冲区，避免频繁读写I/O
- **单缓冲区例子：Linux的page cache**

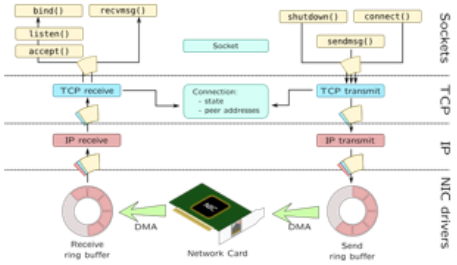

**双缓冲区**

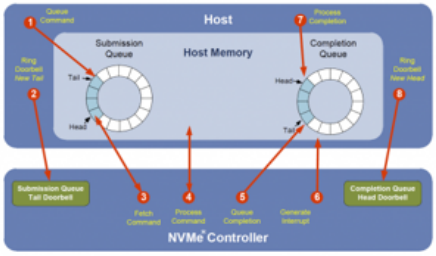
- **维护两个缓冲区，轮流使用**
- **第一个缓冲区被填满但没被读取前，使用第二个缓冲区填充数据**
- **双缓冲区例子：显存刷新，防止屏幕内容出现闪烁或撕裂**
  - 前置缓冲区被读取后，通过“交换”（swap）将前置和后置身份互换
- **游戏中甚至启用三重缓冲**




**环形缓冲区**

- **容许更多缓冲区存在，提高I/O带宽**
- **组成：一段连续内存区域+两个指针，读写时自动推进指针**
  - 读指针：指向有效数据区域的开始地址
  - 写指针：指向下一个空闲区域的开始地址
- **环形缓冲区例子：网卡DMA缓冲区、NVMe存储的命令队列**




**设备的缓冲区管理小结**

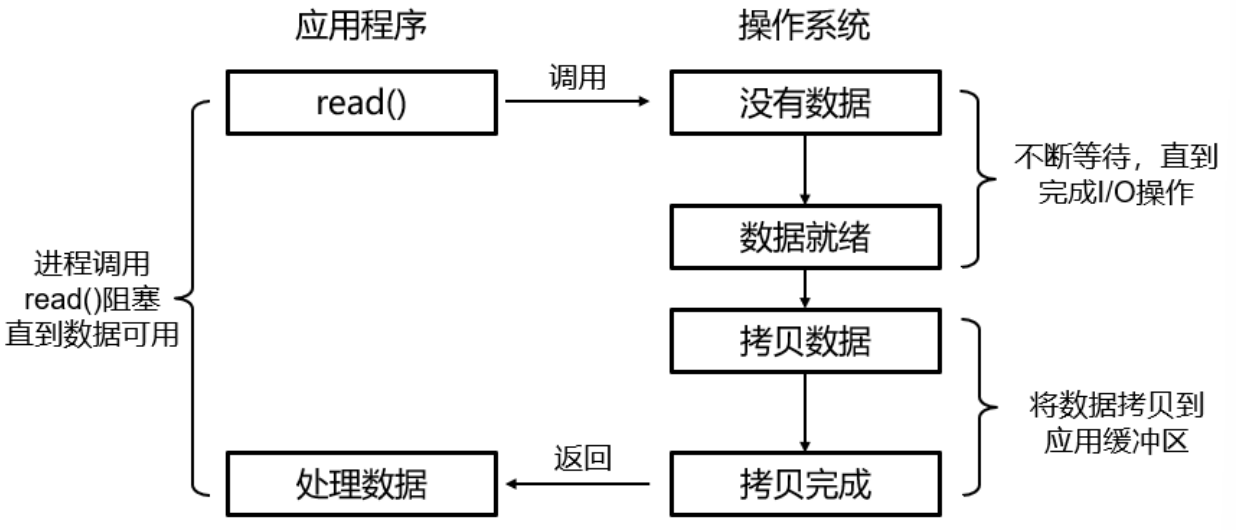
- **多种缓冲区实现方式**
  - 单缓冲区：块设备的缓冲区
  - 双缓冲区：常用于游戏或流媒体的显存同步
  - 环形缓冲区：网卡的数据交互和NVMe存储的命令交互

- **思考题：缓冲区是否总能提高性能?**
  - 缓冲区意味着数据的多次拷贝，使用过多反而损伤性能

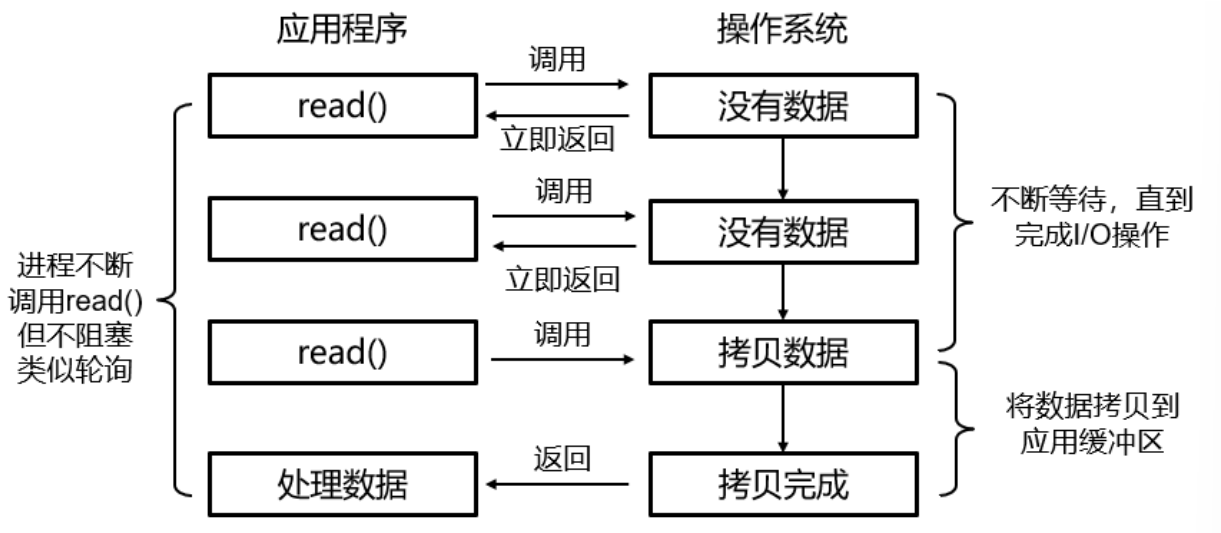
**问题：如何同时监听多个设备?**

- **如果要支持多个fd怎么办?**
  - 阻塞I/O：一旦一个阻塞，则其余无法响应
- **改成非阻塞可以么?**
  - 非阻塞I/O：需要程序不断轮询设备情况：浪费CPU
- **能否采用异步通知机制？让内核主动来通知?**
  - 异步I/O：允许程序先做其他事，等设备数据就绪再接收通知
  - 多路复用I/O：仍为阻塞，但一旦设备数据就绪就收到通知

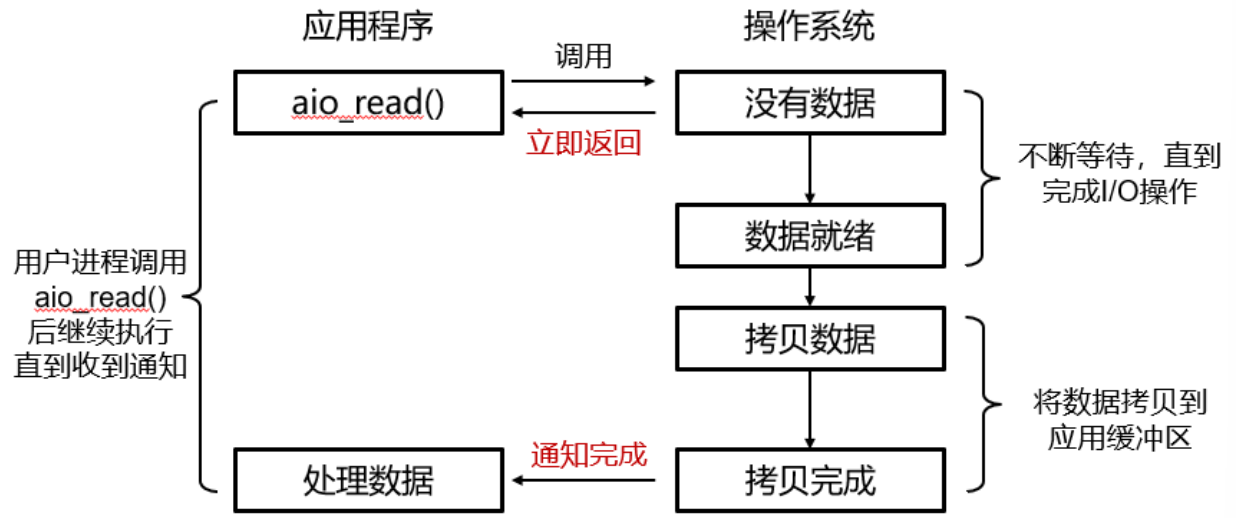
**I/O模型：阻塞I/O模型**



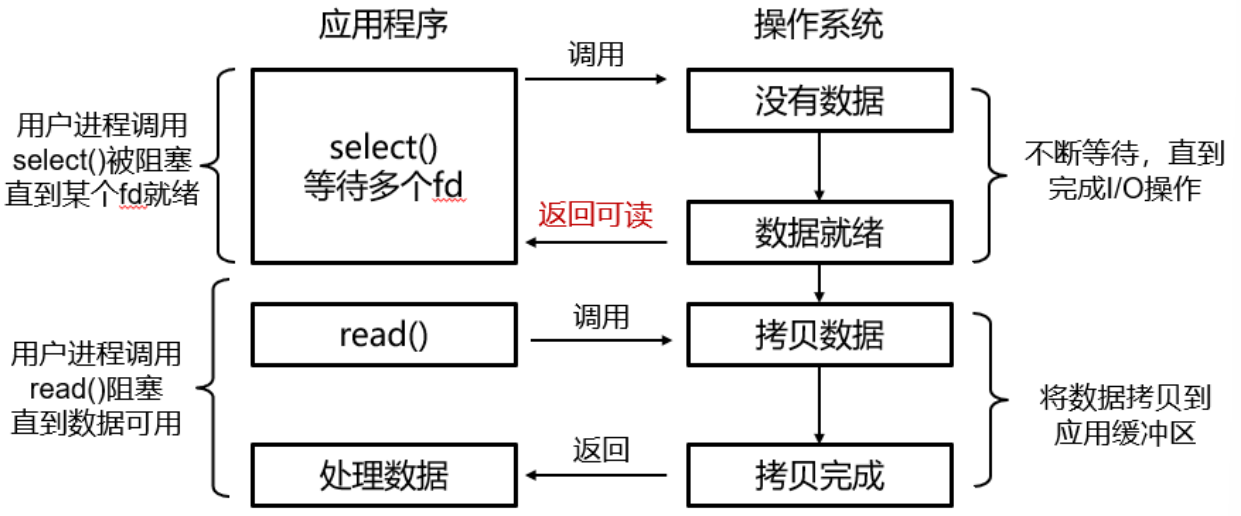

**I/O模型：非阻塞I/O模型**




**I/O模型：异步I/O模型**




**I/O模型：I/O多路复用模型**




**I/O模型小结**

- **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**
  - 等读写请求成功后再通知用户
  - 用户执行并不停滞（类似DMA之于CPU）
- **I/O多路复用：同时监听多个请求，只要有一个就绪就不再等待**