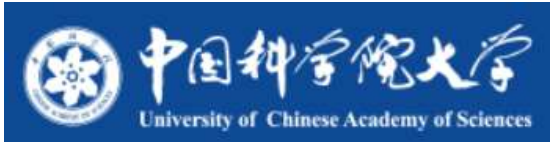
Institute of Software,Chinese Academy of Sciences

设备管理

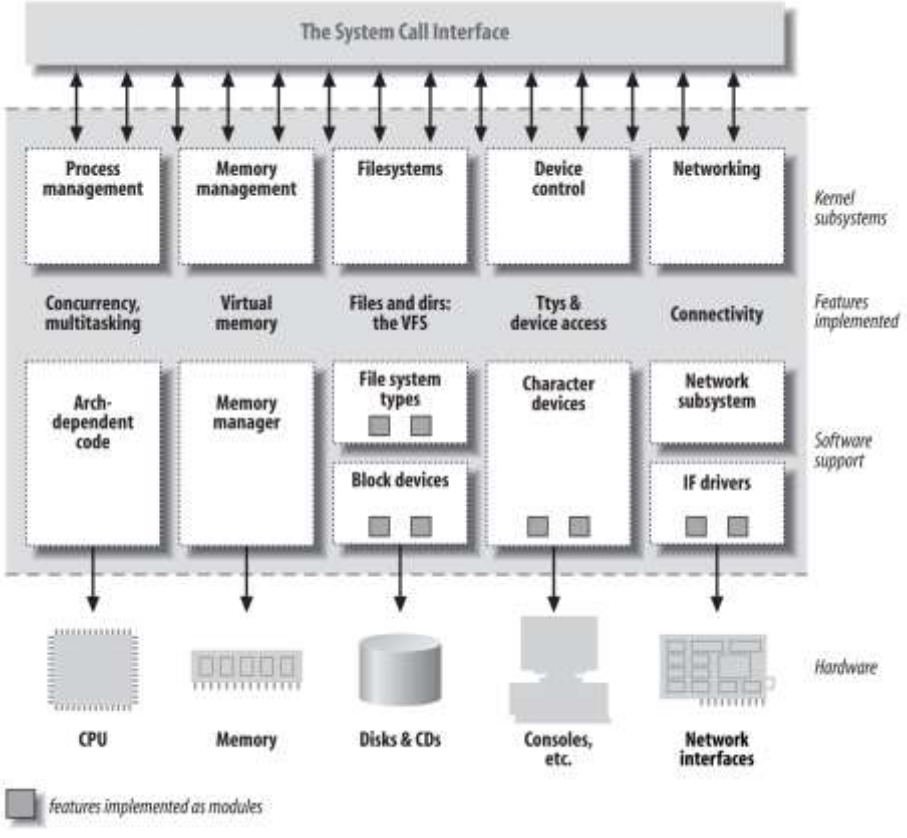
李鹏

改编声明

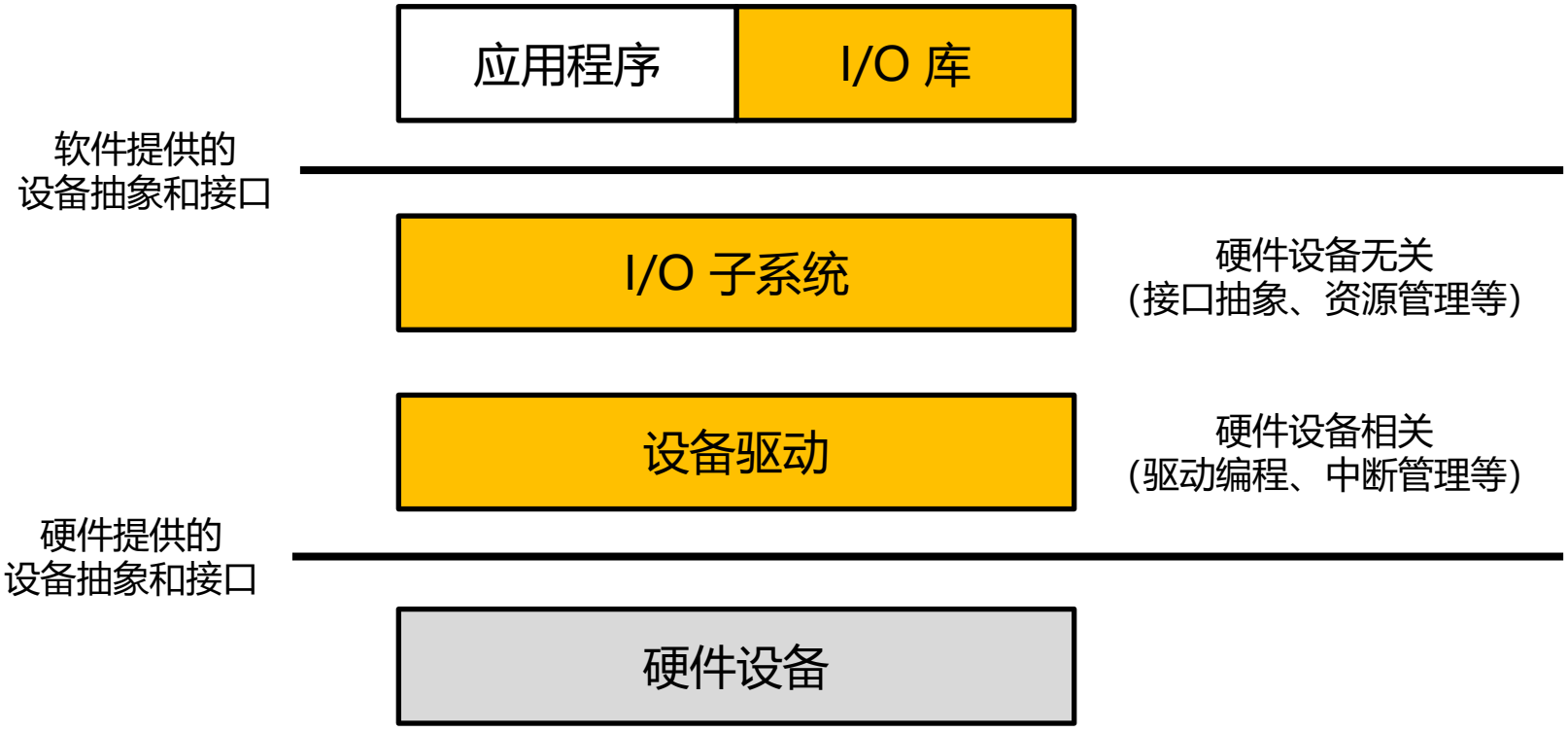
- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，已获得原作者授权，原课程官网：
  - https://ipads.se.sjtu.edu.cn/courses/os/index.shtml
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。



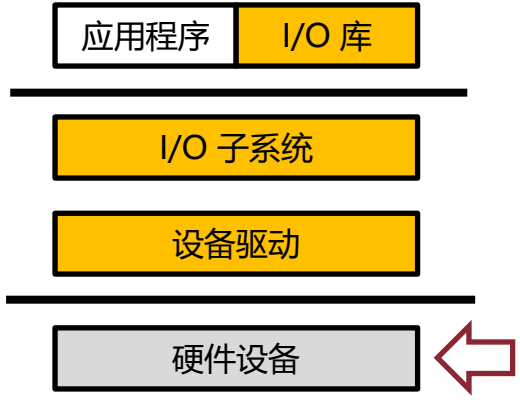
操作系统内核剖分图


The System Call Interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs: the VFS
Ttys & device access
Connectivity
Features implemented
Arch-dependent code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
CPU
Memory
Disks & CDs
Consoles, etc.
Network interfaces
Hardware
features implemented as modules

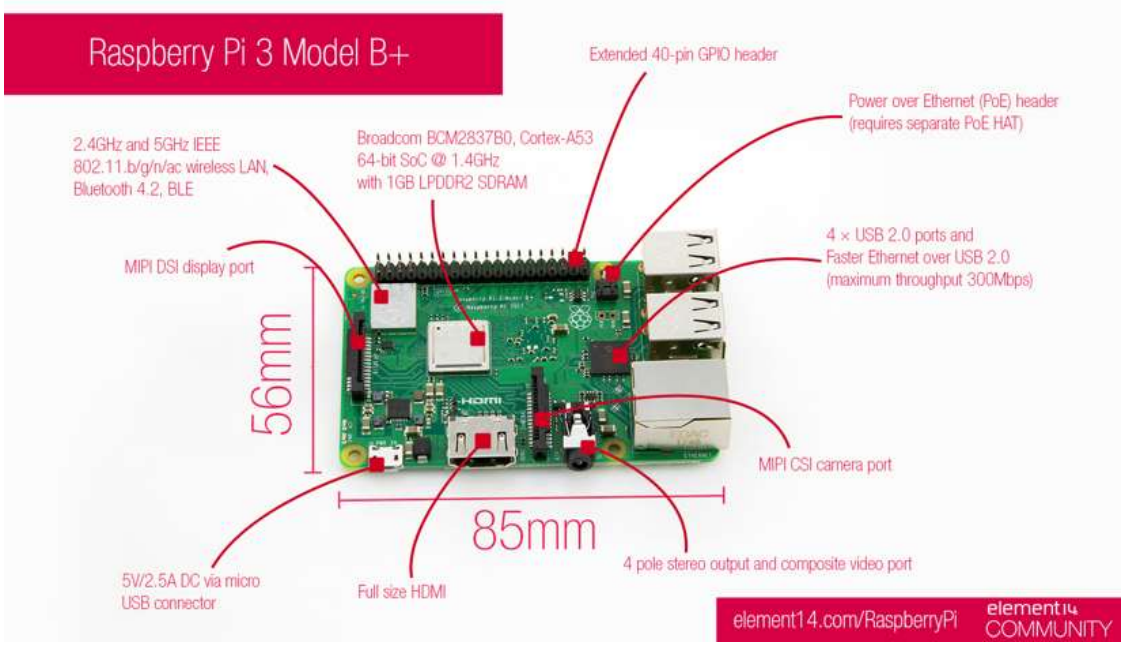
操作系统的I/O层次


应用程序
I/O 库
软件提供的
设备抽象和接口
I/O 子系统
硬件设备无关
(接口抽象、资源管理等)
设备驱动
硬件设备相关
(驱动编程、中断管理等)
硬件提供的
设备抽象和接口
硬件设备

认知设备



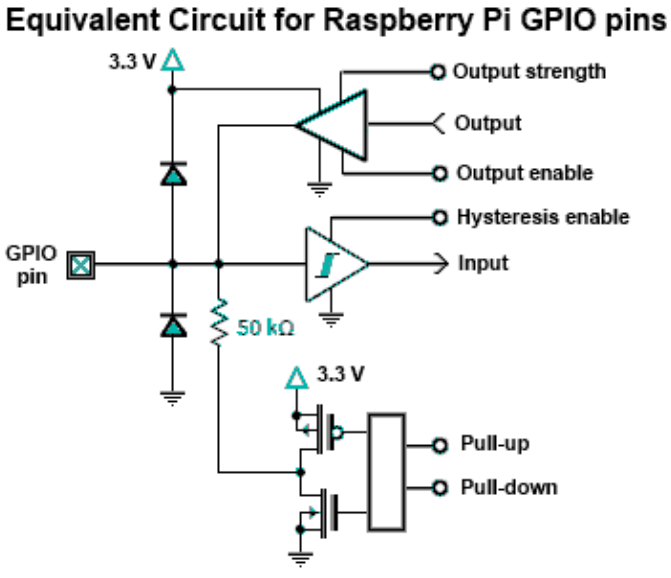
计算机系统上的硬件设备很多



- 种类繁多 ☹
- 规范不同 ☹
- 接口不同 ☹

GPIO LED

- 有专门的“输入/输出”引脚
- 通过引脚控制LED状态
- 每种01组合只呈现一种发光状态

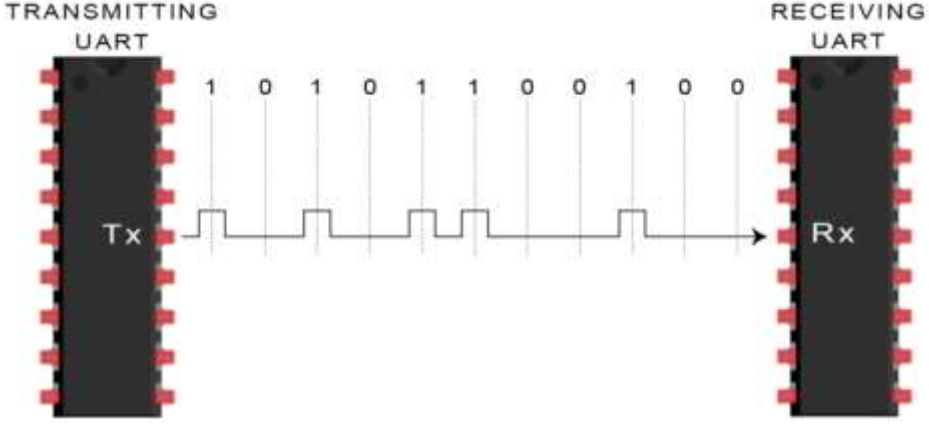


8042（PS/2 键盘控制器）

- 电信号→数字信号→编码（Scan Code）
- 每次只能键入一个字符

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**



Flash 闪存

- 按照页/块的粒度进行读写/擦除

- 支持页/块随机访问

Ethernet 网卡

- 每次传输一帧数据（以太网帧）

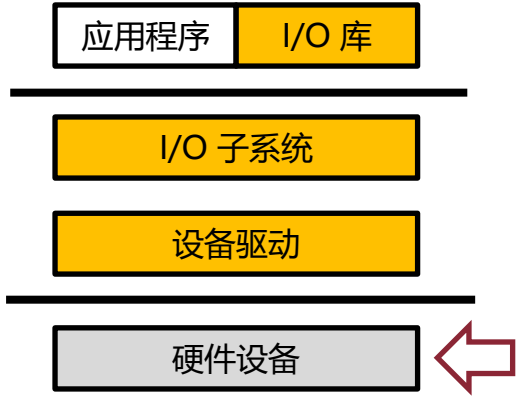
- Wifi、蓝牙类似

思考题

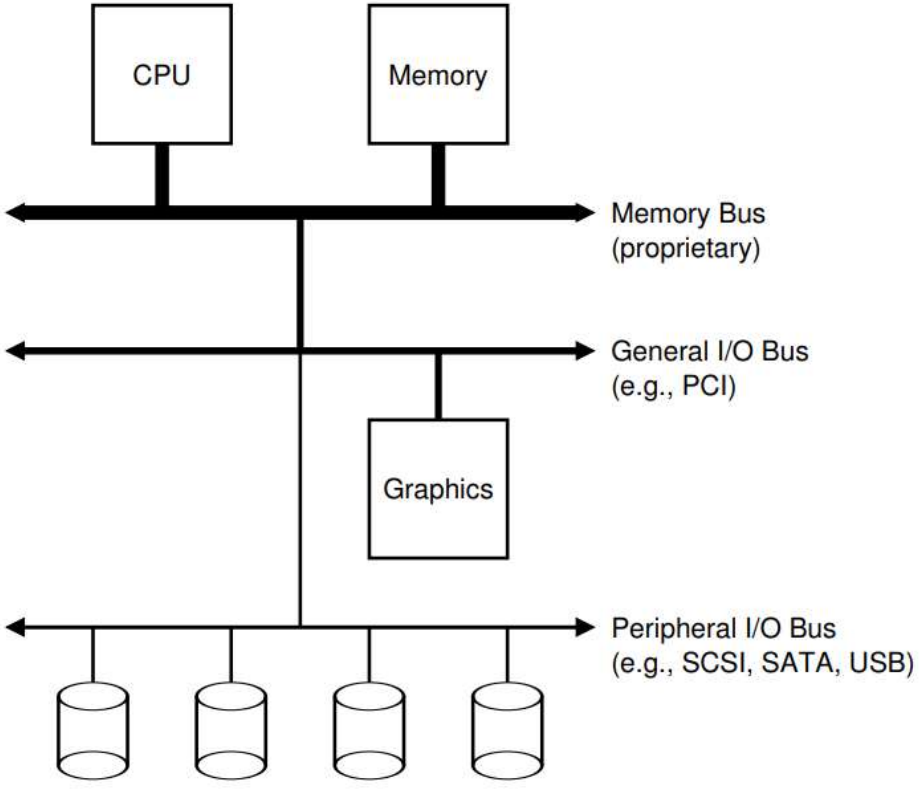
- **树莓派上的这些设备可以和前面提到的哪些设备归为一类?**
  - SD 存储卡
  - RTC 实时时钟
  - DS18B20 温度传感器
  - CSI 摄像头
  - 板载无线蓝牙

硬件视角

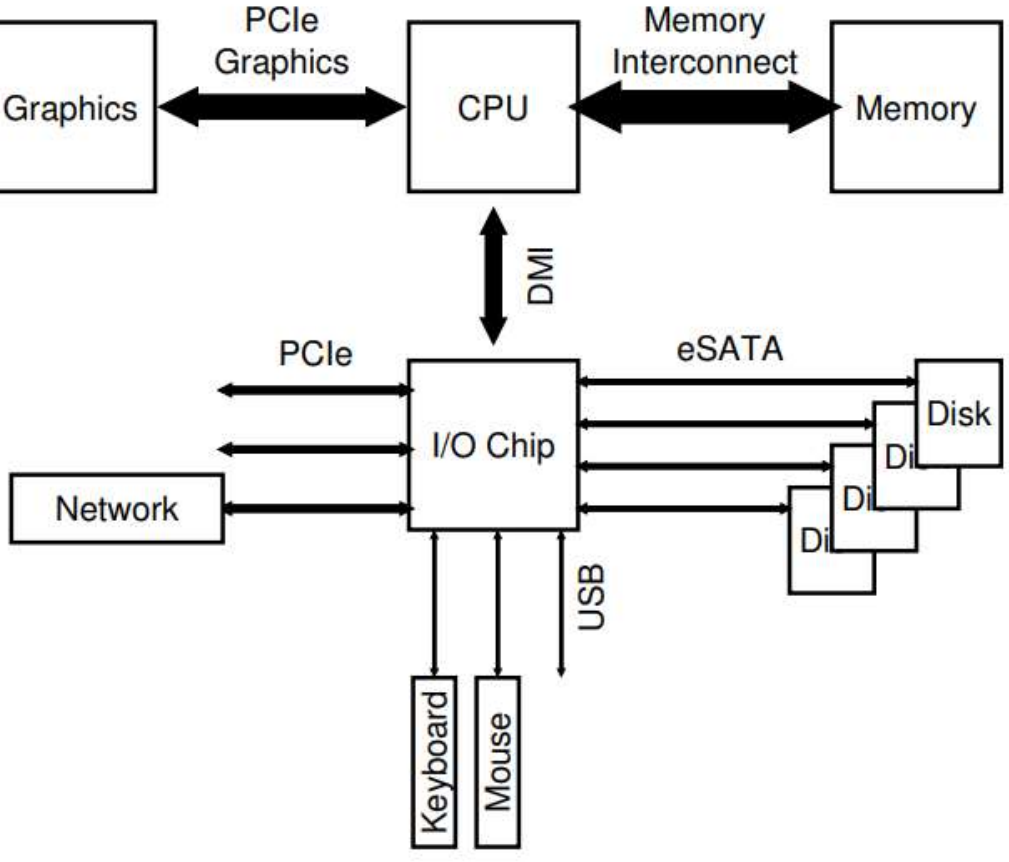
设备与CPU的连接



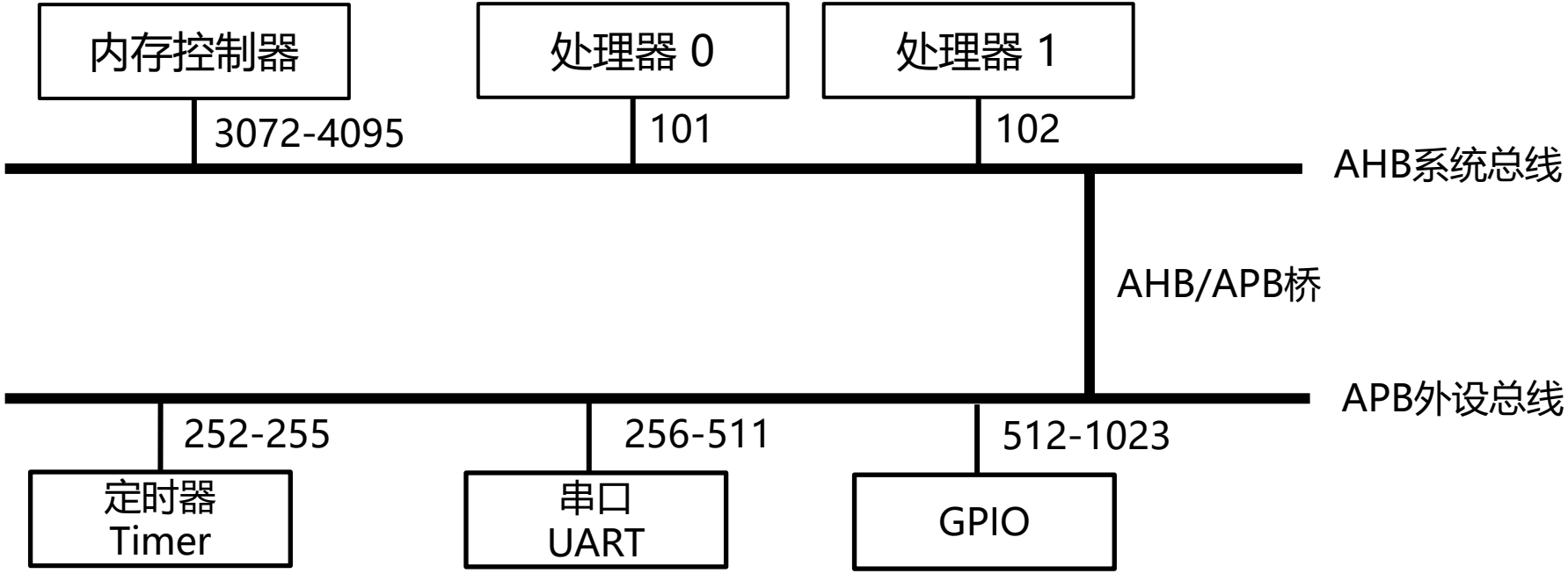
经典计算机系统架构


CPU
Memory
Memory Bus
(proprietary)
General I/O Bus
(e.g., PCI)
Graphics
Peripheral I/O Bus
(e.g., SCSI, SATA, USB)

现代计算机系统架构


Graphics
PCIe
Graphics
CPU
Memory
Interconnect
Memory
DMI
PCIe
eSATA
I/O Chip
Disk
Network
USB
Keyboard
Mouse

移动设备系统结构


内存控制器
处理器 0
处理器 1
3072-4095
101
102
AHB系统总线
AHB/APB桥
APB外设总线
252-255
256-511
512-1023
定时器
Timer
串口
UART
GPIO

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：标识了预期的接收方
- **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（Source）和目标（destination）借助共享时钟进行协作
  - 例子：DDR内存访问

- **异步数据传输**
  - 源（Source）和目标（destination）借助显式信号进行协作
  - 例子：对信号的确认（ack）

总线事务

① **源（发送方）获取总线的使用权（具有排他性）**

② **源（发送方）将目标（接收方）的地址写到总线上**

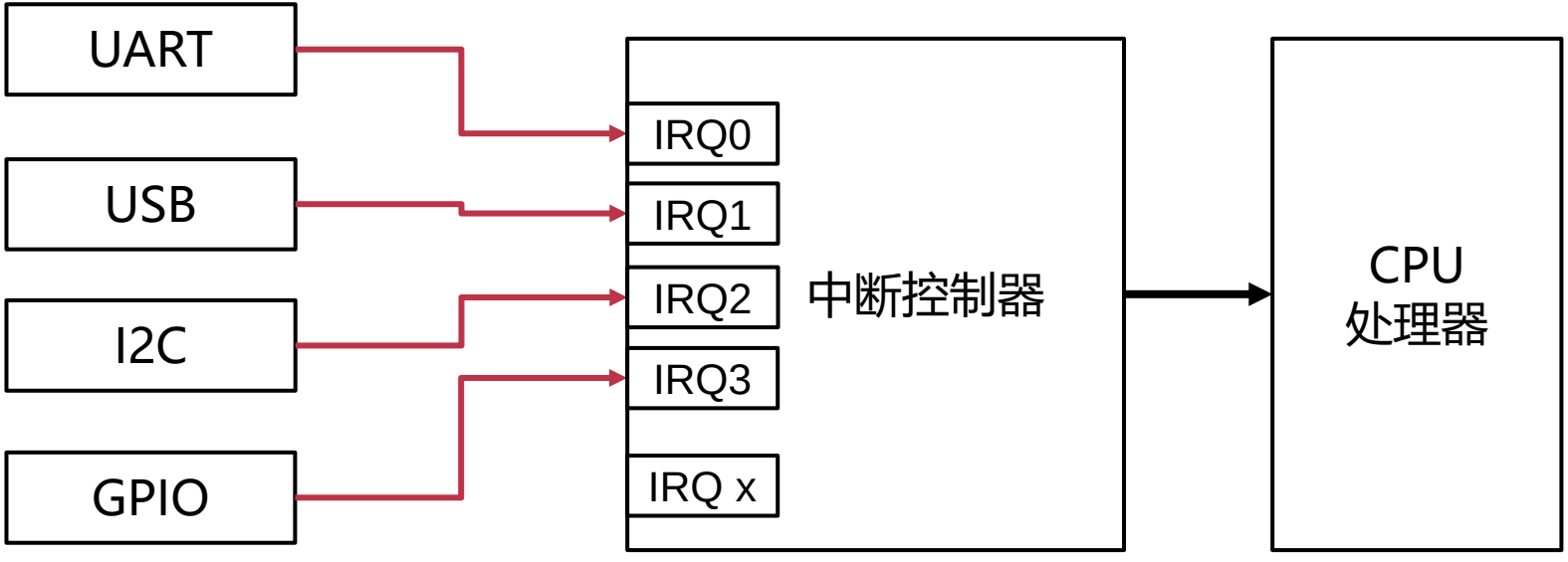
③ **源（发送方）发出 READY 信号，提醒其他模块（广播）**

④ **目标（接收方）在拷贝完数据后，发出 ACKNOWLEDGE 信号**

- 同步模式下，无需 READY 和 ACKNOWLEDGE，只要在每个时钟周期进行检查即可

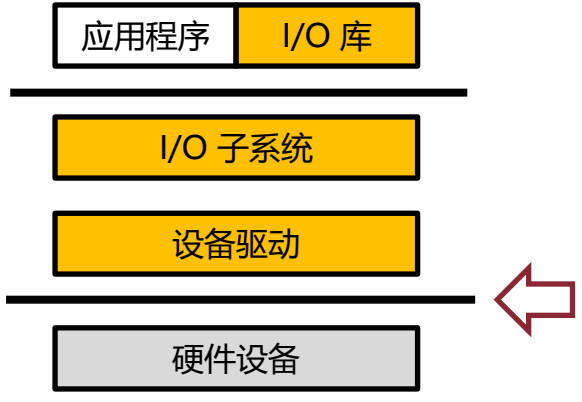
⑤ **源（发送方）释放总线**

中断线

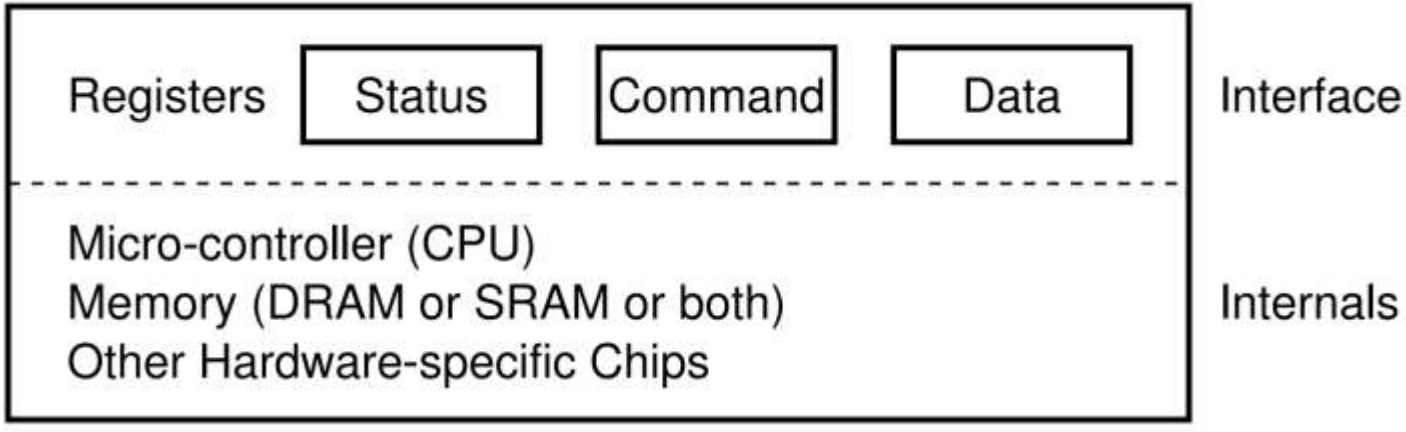


驱动视角

设备与CPU的交互

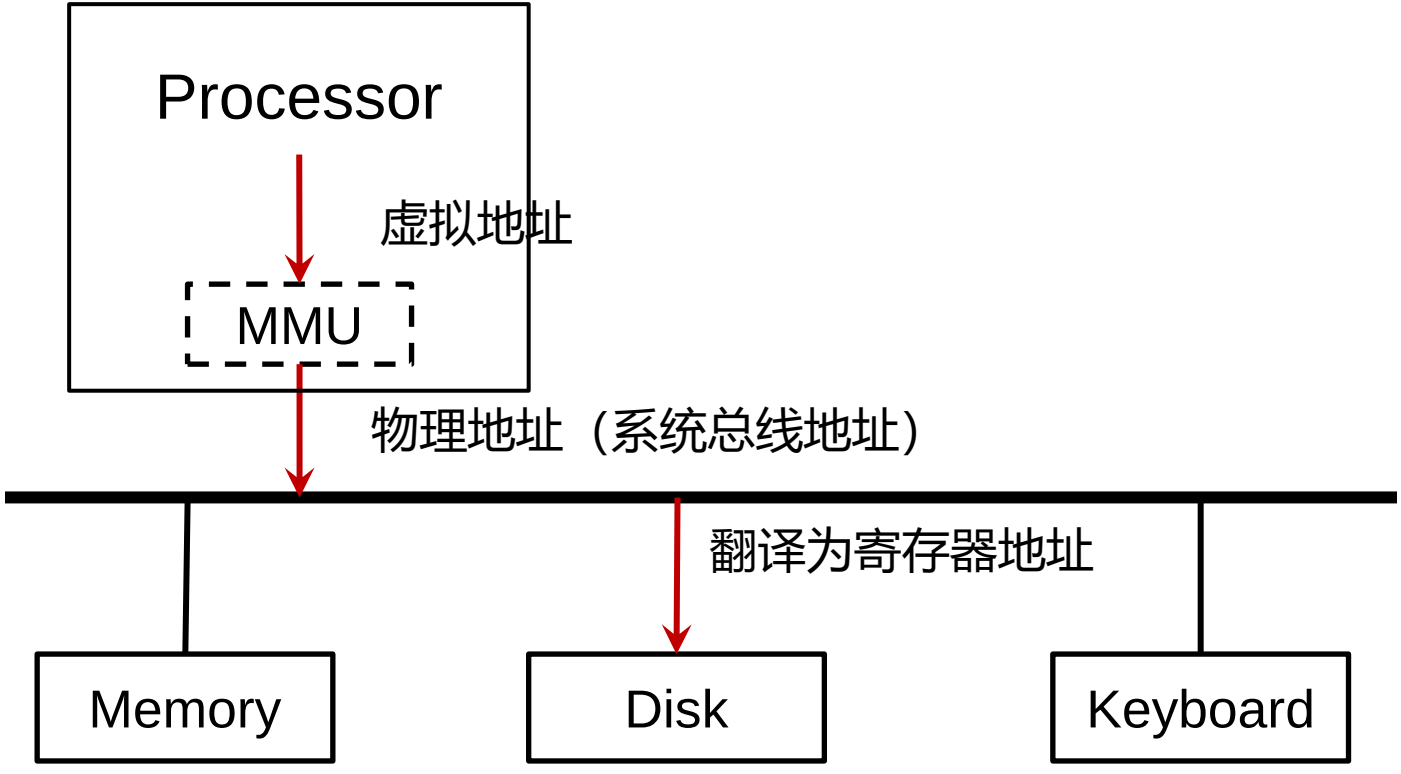


硬件设备的接口：设备寄存器




```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

内存映射 I/O (MMIO)


Processor
虚拟地址
MMU
物理地址（系统总线地址）
翻译为寄存器地址
Memory
Disk
Keyboard

UART MMIO代码

```
u32 pl011_nb_uart_recv(void)
{
    if (!(get32((u64) UART_PPTR + UART_FR)
        & UART_FR_RXFE))
        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)
```

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)**
  - 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)**
  - IO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

轮询 (Polling)

- 等待直到设备状态变为非“忙”
  - 不停轮询（loop）
- 好处
  - 简单
- 坏处
  - 慢
  - 浪费CPU
- 例子
  - 如果一个设备的速度是100 ops/sec，CPU需要等待10毫秒
  - 对于1GHz的CPU，意味着 1千万个CPU 时钟周期

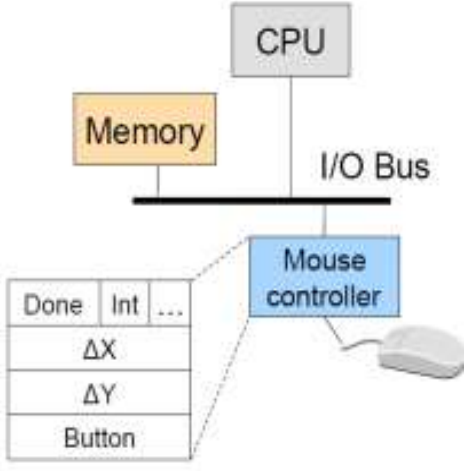
中断

·例子：鼠标
·简单的鼠标控制器
- 状态寄存器（完成、中断、……）
- 数据寄存器（ΔX，ΔY，按键）

·输入

鼠标
- 等待直到设备状态变为“完成”
- 将ΔX，ΔY和按键的值保存到数据寄存器
- 请求中断

CPU（中断处理）
- 清除“完成”标志
- 将ΔXΔY和按键的值读到内核（变量）
- 置“完成”标志
- 调用调度器



中断是否一定比轮询更好?

DMA (Direct Memory Access)

·例子：磁盘

·一个简单的磁盘适配器

- 状态寄存器（完成、中断、…）
- DMA命令
- DMA内存地址和大小
- DMA数据缓冲区

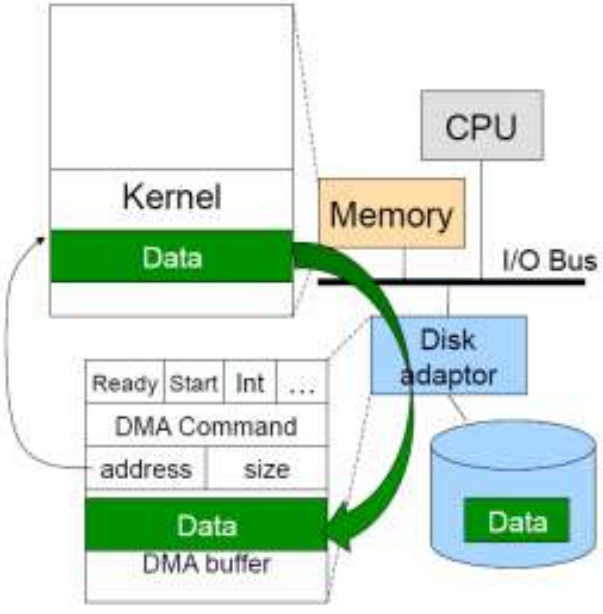
·DMA写

CPU

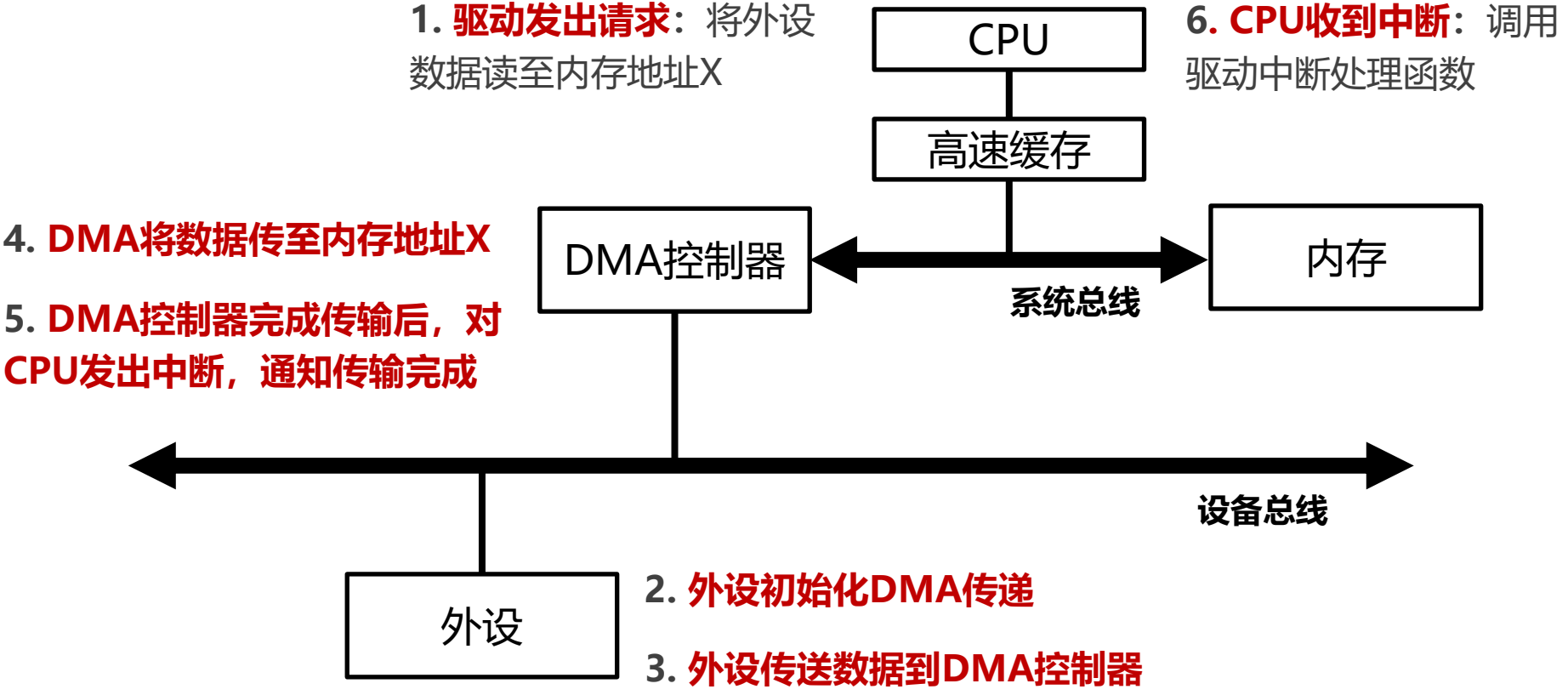
- 等待DMA设备状态为就绪
- 清除“就绪”
- 设置DMA命令为wite，地址和大小
- 设置“开始”
- 阻塞当前的线程/进程

磁盘适配器

- DMA方式将数据传输到磁盘（size-- addr++）
- 当size==0，请求中断
- CPU（中断处理）
- 将被该DMA阻塞的线程/进程加到就绪队列

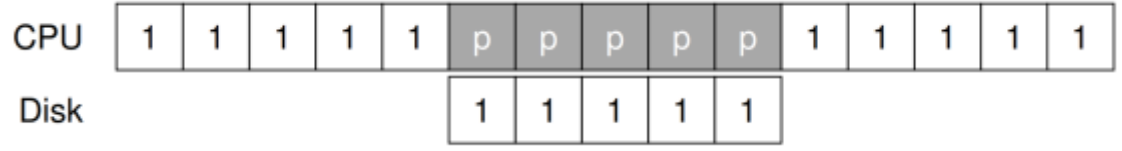


DMA

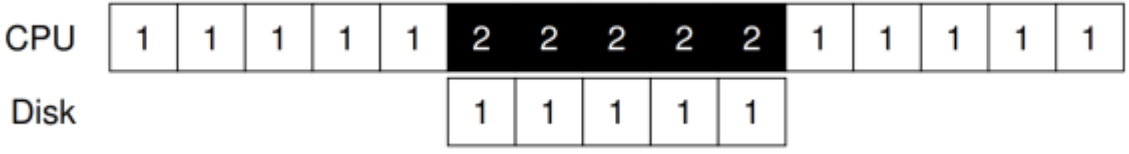

1. 驱动发出请求：将外设数据读至内存地址X
CPU
6. CPU收到中断：调用驱动中断处理函数
高速缓存
4. DMA将数据传至内存地址X
DMA控制器
内存
系统总线
5. DMA控制器完成传输后，对CPU发出中断，通知传输完成
设备总线
外设
2. 外设初始化DMA传递
3. 外设传送数据到DMA控制器

轮询vs中断vsDMA

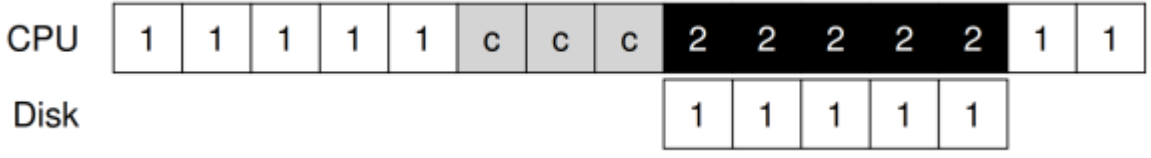
轮询



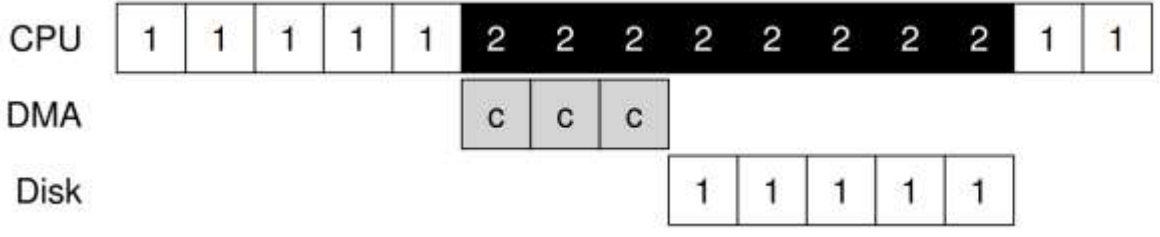
中断



中断（Copy）

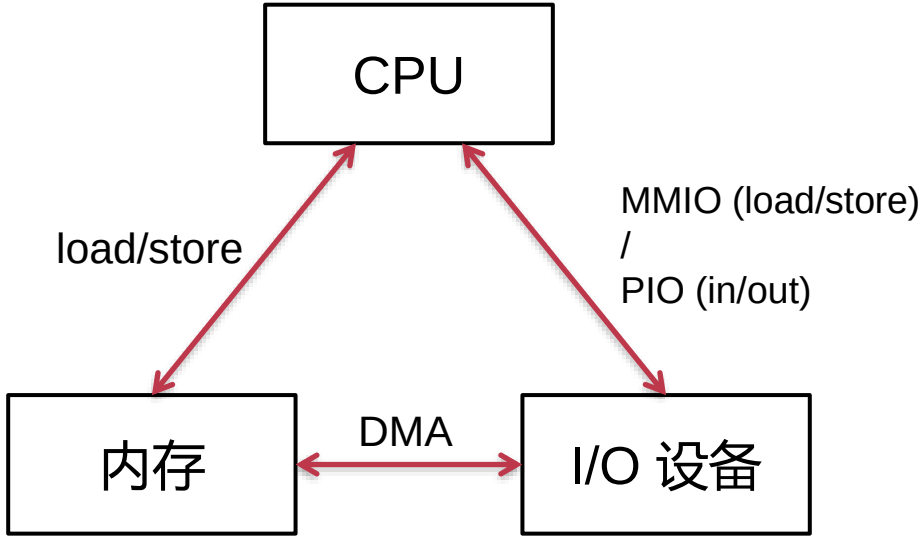


DMA



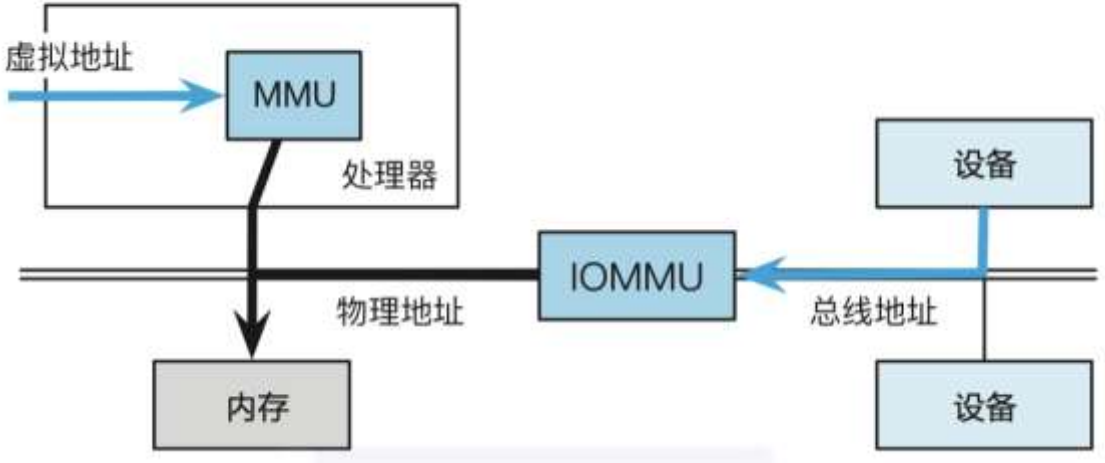
CPU访问设备方式小结

- **MMIO**
  - 将设备寄存器映射到物理地址空间
  - CPU通过读写设备寄存器操作设备
- **DMA**
  - 设备使用物理地址访问内存
  - **思考：如何保证设备访存的安全性?**



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）



思考题：DMA的内存一致性

- **现代处理器通常带有高速缓存（CPU Cache）**
- **当DMA发生时，DMA缓冲区的数据仍在cache中怎么办?**

- **解决方法：**
  - 方案1：将DMA区域映射为non-cacheable
  - 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

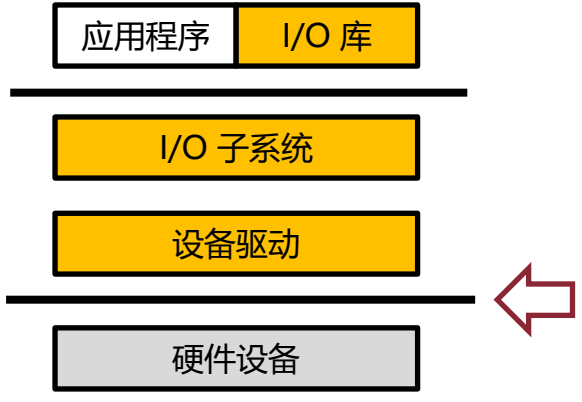
I/O的性能标准

- 开销
  - CPU用于启动设备进行操作的时间
- 延迟
  - 传输1字节的时间
  - 开销 + 将1字节传输到目的地的时间
- 带宽
  - 启动设备后I/O传输的速率
  - Bytes/sec
- 一般化
  - 不同的传输速率
  - 对字节传输的抽象
  - 以块为传输粒度，从而分摊开销

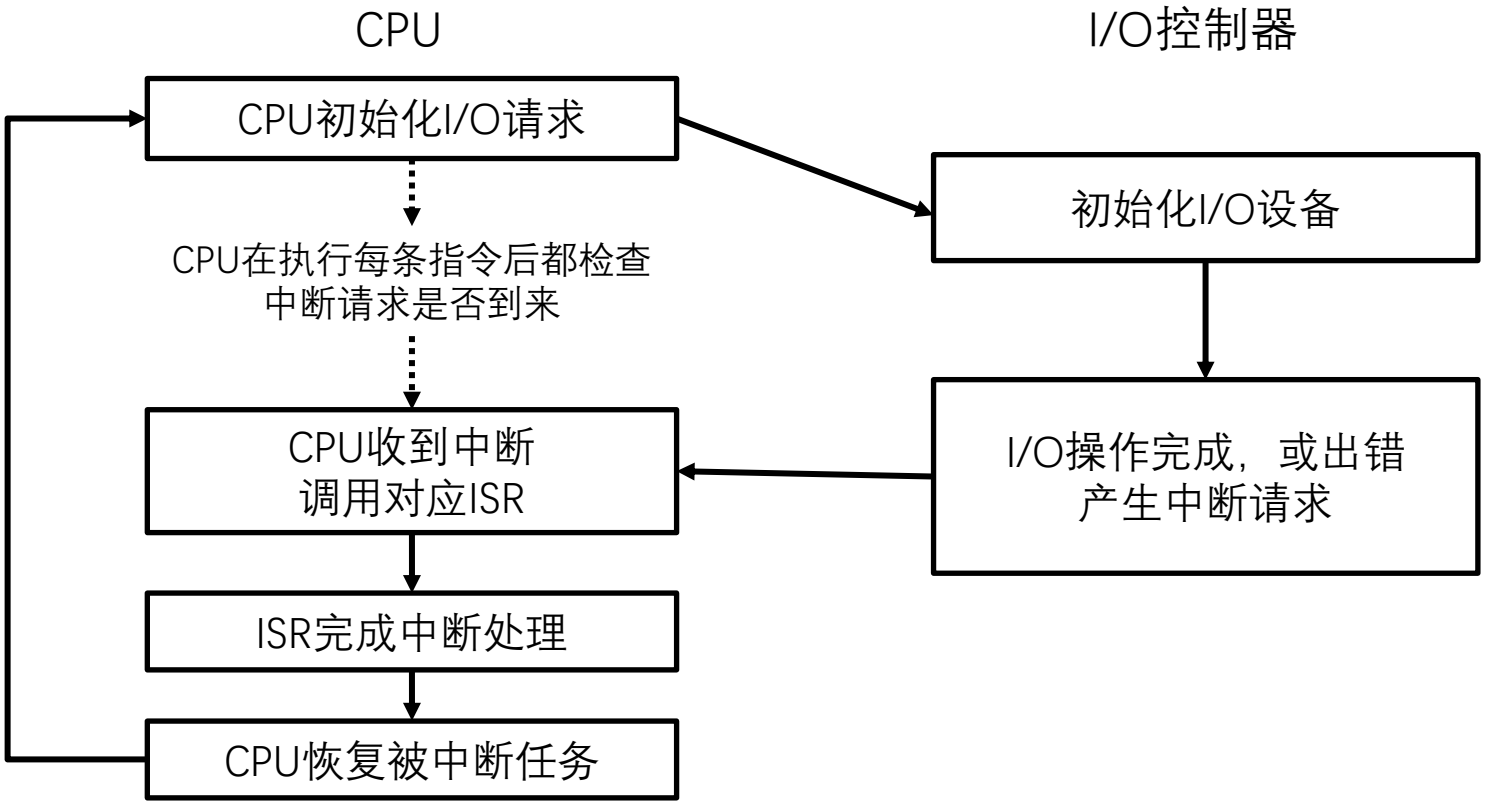
Device	Transfer rate
Keyboard	10Bytes/sec
Mouse	100Bytes/sec
...	...
10GE NIC	1.2GBytes/sec

设备通知CPU的方式

中断与中断响应

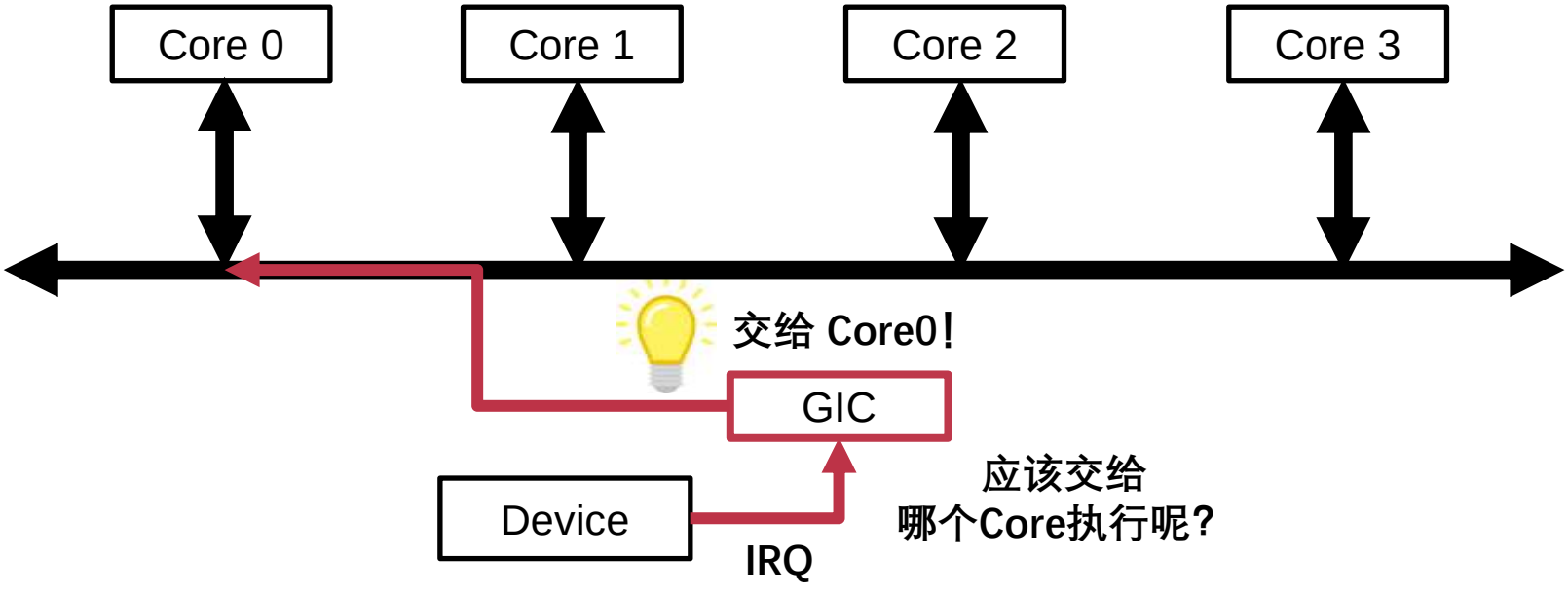


CPU中断处理流程

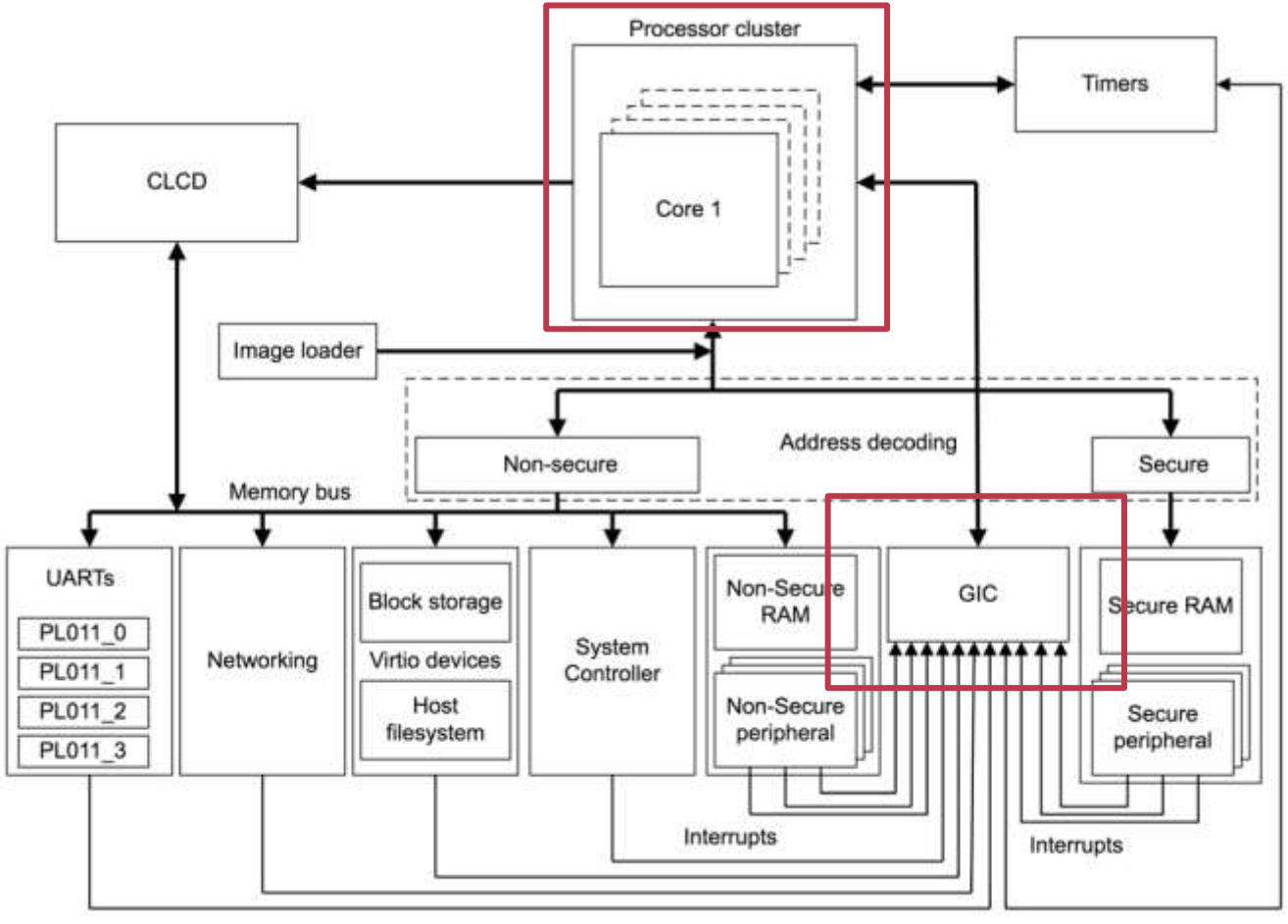

CPU
I/O控制器
CPU初始化I/O请求
初始化I/O设备
CPU在执行每条指令后都检查
中断请求是否到来
CPU收到中断
调用对应ISR
I/O操作完成，或出错
产生中断请求
ISR完成中断处理
CPU恢复被中断任务

问题：多核CPU如何处理中断?

- 如何避免中断一次性打断所有核呢?



中断控制器——GIC


Processor cluster
Core 1
Timers
CLCD
Image loader
Address decoding
Non-secure
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

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“门卫”，判断中断是否要发给CPU处理

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

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

问题：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
- **中断优先级**

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

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的响应
- **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

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，使用轮询处理后续达到的网络包
  - 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网络驱动中称为 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同一个中断，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）

设备管理的三种方式

设备管理主要的三类方式

- **第一类（内核直管）：静态I/O资源分配+向上提供文件接口**
  - 代表设备：Console（CGA+键盘）
  - 特征：中断号等固定，内核直接提供文件接口（无需外部驱动）
- **第二类（设备驱动）：动态I/O资源分配+向上提供文件接口**
  - 代表设备：PCI、USB，如硬盘、网卡等
  - 特征：中断号等动态分配，内核提供驱动框架，设备制造商提供驱动模块
- **第三类（用户态库）：动态I/O资源分配+向上提供内存接口**
  - 代表设备：智能网卡、智能SSD
  - 特征：内核直接将设备暴露给用户态，设备制造商提供用户态库

OS的设备管理：对上与对下

- **向下对接设备：分配必要的I/O资源**
  - 中断号 + 虚拟地址映射空间
    - 静态绑定：由硬件规范确定
    - 动态分配：动态扫描后再分配
- **对上提供接口：应用使用设备的方式**
  - 文件接口：通过系统调用
    - open、read、write、close、ioctl等
  - 内存接口：寄存器直接映射到用户态虚拟内存空间
    - 无需系统调用，应用可直接访问设备

第一类：内核直管

- **以Console为例**
  - 包含设备：显示器（CGA）+键盘
- **I/O资源静态绑定**
  - 中断号（即异常向量表）
  - 寄存器所映射的虚拟内存地址
- **文件接口**
  - read：读取键盘输入，通常以行为单位
  - write：向显示器输出字符并显示

```
273 int
274 consolewrite(struct inode *ip, char *buf, int n)
275 {
276   int i;
277
278   iunlock(ip);
279   acquire(&cons.lock);
280   for(i = 0; i < n; i++)
281     consputc(buf[i] & 0xff);
282   release(&cons.lock);
283   ilock(ip);
284
285   return n;
286 }
```

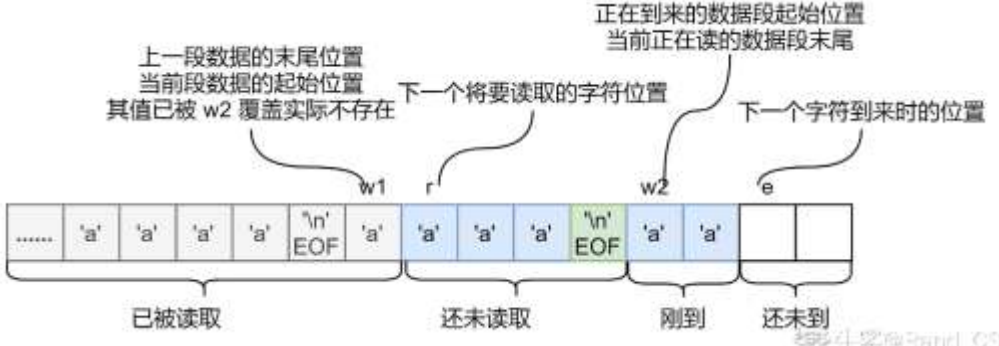
```
165 void
166 consputc(int c)
167 {
168   if(panicked){
169     cli();
170     for(;;)
171       ;
172   }
173
174   if(c == BACKSPACE){
175     uartputc('\b'); uartputc(' '); uartputc('\b');
176   } else
177     uartputc(c);
178   cgaputc(c);
179 }
```

```
131 static void
132 cgaputc(int c)
133 {
134   int pos;
135
136   // Cursor position: col + 80*row.
137   outb(CRTPORT, 14);
138   pos = inb(CRTPORT+1) << 8;
139   outb(CRTPORT, 15);
140   pos |= inb(CRTPORT+1);
141
142   if(c == '\n')
143     pos += 80 - pos%80;
144   else if(c == BACKSPACE){
145     if(pos > 0) --pos;
146   } else
147     crt[pos++] = (c&0xff) | 0x0700;  // black on white
148
149   if(pos < 0 || pos > 25*80)
150     panic("pos under/overflow");
151
152   if((pos/80) >= 24){  // Scroll up.
153     memmove(crt, crt+80, sizeof(crt[0])*23*80);
154     pos -= 80;
155     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
156   }
157
158   outb(CRTPORT, 14);
159   outb(CRTPORT+1, pos>>8);
160   outb(CRTPORT, 15);
161   outb(CRTPORT+1, pos);
162   crt[pos] = ' ' | 0x0700;
163 }
```

```
235 int
236 consoleread(struct inode *ip, char *dst, int n)
237 {
238   uint target;
239   int c;
240
241   iunlock(ip);
242   target = n;
243   acquire(&cons.lock);
244   while(n > 0){
245     while(input.r == input.w){
246       if(myproc()->killed){
247         release(&cons.lock);
248         ilock(ip);
249         return -1;
250       }
251       sleep(&input.r, &cons.lock);
252     }
253     c = input.buf[input.r++ % INPUT_BUF];
254     if(c == C('D')){  // EOF
255       if(n < target){
256         // Save ^D for next time, to make sure
257         // caller gets a 0-byte result.
258         input.r--;
259       }
260       break;
261     }
262     *dst++ = c;
263     --n;
264     if(c == '\n')
265       break;
266   }
267   release(&cons.lock);
268   ilock(ip);
269
270   return target - n;
271 }
```

```
#define INPUT_BUF 128
struct {
  char buf[INPUT_BUF];
  uint r;  // Read index
  uint w;  // Write index
  uint e;  // Edit index
} input;
```

input.buf 何时更新?



键盘中断

```
36 void
37 trap(struct trapframe *tf)
38 {
39   if(tf->trapno == T_SYSCALL){
40     if(myproc()->killed)
41       exit();
42     myproc()->tf = tf;
43     syscall();
44     if(myproc()->killed)
45       exit();
46     return;
47   }
48
49   switch(tf->trapno){
50   case T_IRQ0 + IRQ_TIMER:
51     if(cpuid() == 0){
52       acquire(&tickslock);
53       ticks++;
54       wakeup(&ticks);
55       release(&tickslock);
56     }
57     lapiceoi();
58     break;
59   case T_IRQ0 + IRQ_IDE:
60     ideintr();
61     lapiceoi();
62     break;
63   case T_IRQ0 + IRQ_IDE+1:
64     // Bochs generates spurious IDE1 interrupts.
65     break;
66   case T_IRQ0 + IRQ_KBD:
67     kbdintr();
68     lapiceoi();
69     break;
```

```
46 void
47 kbdintr(void)
48 {
49   consoleintr(kbdgetc);
50 }
```

```
191 void
192 consoleintr(int (*getc)(void))
193 {
194   int c, doprocdump = 0;
195
196   acquire(&cons.lock);
197   while((c = getc()) >= 0){
198     switch(c){
199     case C('P'):  // Process listing.
200       // procdump() locks cons.lock indirectly; invoke later
201       doprocdump = 1;
202       break;
203     case C('U'):  // Kill line.
204       while(input.e != input.w &&
205             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
206         input.e--;
207         consputc(BACKSPACE);
208       }
209       break;
210     case C('H'): case '\x7f':  // Backspace
211       if(input.e != input.w){
212         input.e--;
213         consputc(BACKSPACE);
214       }
215       break;
216     default:
217       if(c != 0 && input.e-input.r < INPUT_BUF){
218         c = (c == '\r') ? '\n' : c;
219         input.buf[input.e++ % INPUT_BUF] = c;
220         consputc(c);
221         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
222           input.w = input.e;
223           wakeup(&input.r);
224         }
225       }
226       break;
227     }
228   }
229   release(&cons.lock);
230   if(doprocdump) {
231     procdump();  // now call procdump() wo. cons.lock held
232   }
233 }
```

```
6  int
7  kbdgetc(void)
8  {
9    static uint shift;
10   static uchar *charcode[4] = {
11     normalmap, shiftmap, ctlmap, ctlmap
12   };
13   uint st, data, c;
14
15   st = inb(KBSTATP);
16   if((st & KBS_DIB) == 0)
17     return -1;
18   data = inb(KBDATAP);
19
20   if(data == 0xE0){
21     shift |= E0ESC;
22     return 0;
23   } else if(data & 0x80){
24     // Key released
25     data = (shift & E0ESC ? data : data & 0x7F);
26     shift &= ~(shiftcode[data] | E0ESC);
27     return 0;
28   } else if(shift & E0ESC){
29     // Last character was an E0 escape; or with 0x80
30     data |= 0x80;
31     shift &= ~E0ESC;
32   }
33
34   shift |= shiftcode[data];
35   shift ^= togglecode[data];
36   c = charcode[shift & (CTL | SHIFT)][data];
37   if(shift & CAPSLOCK){
38     if('a' <= c && c <= 'z')
39       c += 'A' - 'a';
40     else if('A' <= c && c <= 'Z')
41       c += 'a' - 'A';
42   }
43   return c;
44 }
```

```
69 int
70 sys_read(void)
71 {
72   struct file *f;
73   int n;
74   char *p;
75
76   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
77     return -1;
78   return fileread(f, p, n);
79 }
```

```
96 int
97 fileread(struct file *f, char *addr, int n)
98 {
99   int r;
100
101   if(f->readable == 0)
102     return -1;
103   if(f->type == FD_PIPE)
104     return piperead(f->pipe, addr, n);
105   if(f->type == FD_INODE){
106     ilock(f->ip);
107     if((r = readi(f->ip, addr, f->off, n)) > 0)
108       f->off += r;
109     iunlock(f->ip);
110     return r;
111   }
112   panic("fileread");
113 }
```

```
452 int
453 readi(struct inode *ip, char *dst, uint off, uint n)
454 {
455   uint tot, m;
456   struct buf *bp;
457
458   if(ip->type == T_DEV){
459     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
460       return -1;
461     return devsw[ip->major].read(ip, dst, n);
462   }
463
464   if(off > ip->size || off + n < off)
465     return -1;
466   if(off + n > ip->size)
467     n = ip->size - off;

   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
     bp = bread(ip->dev, bmap(ip, off/BSIZE));
     m = min(n - tot, BSIZE - off%BSIZE);
     memmove(dst, bp->data + off%BSIZE, m);
     brelse(bp);
   }
   return n;
}
```

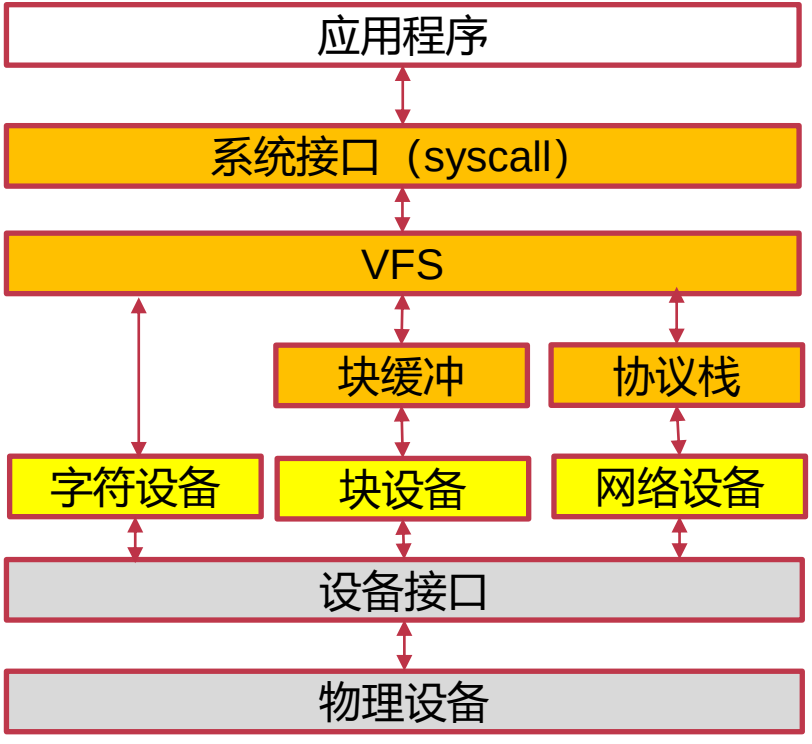
```
288 void
289 consoleinit(void)
290 {
291   initlock(&cons.lock, "console");
292
293   devsw[CONSOLE].write = consolewrite;
294   devsw[CONSOLE].read = consoleread;
295   cons.locking = 1;
296
297   ioapicenable(IRQ_KBD, 0);
298 }
299
```

第二类：设备驱动

- **操作系统的设备驱动框架**
  - 提供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
  - 方便操作系统统一组织和管理设备
- **操作系统为驱动提供的辅助功能**
  - VFS（对上提供文件接口，可复用权限检查等逻辑）
  - 与设备类型相关的框架（如：块设备层、网络协议栈等）
  - 模块动态插入框架（如：Linux的module插入）
  - 中断处理框架（如：Linux的 top half + bottom half）

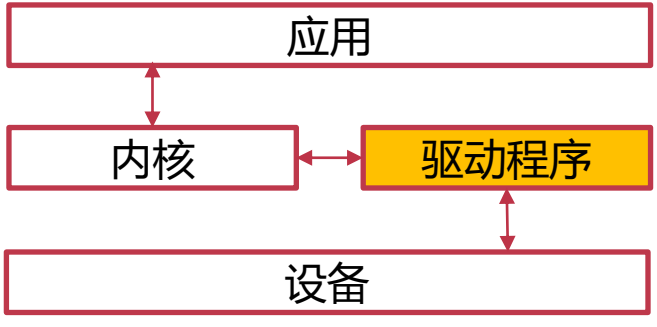
Linux的设备分类

- **Linux设备分类**
  - 字符设备
  - 块设备
  - 网络设备

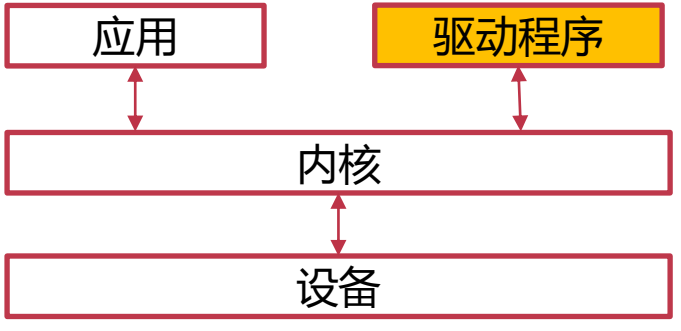


宏内核vs微内核的驱动

- **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）

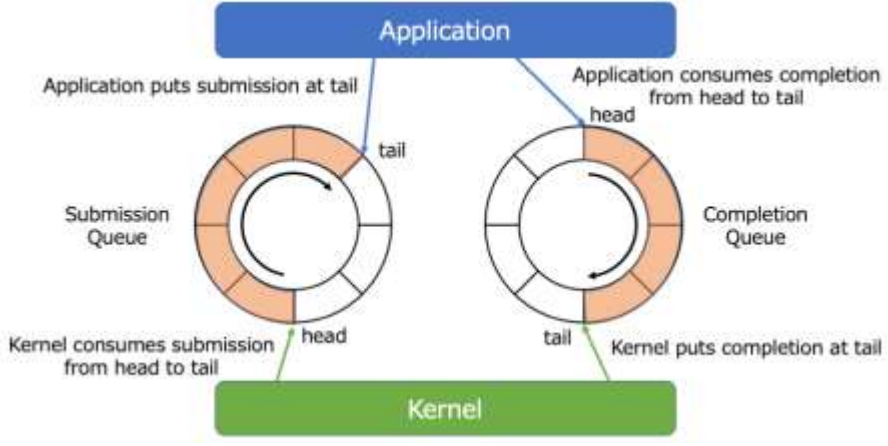


第三类：用户态库

- **操作系统内核负责：**
  - 将设备寄存器映射到用户可访问的虚拟地址空间
  - 通常只允许某一个应用访问该设备，防止发生冲突
- **用户态库负责：**
  - 100%控制硬件设备，并向上或对外提供服务
  - 通过轮询访问设备（问：为何不用中断？）
    - 通常需要独占一个CPU
  - 典型例子：Intel DPDK（网卡）、SPDK（存储）
  - 通常用于高性能场景（问：为什么性能高？）

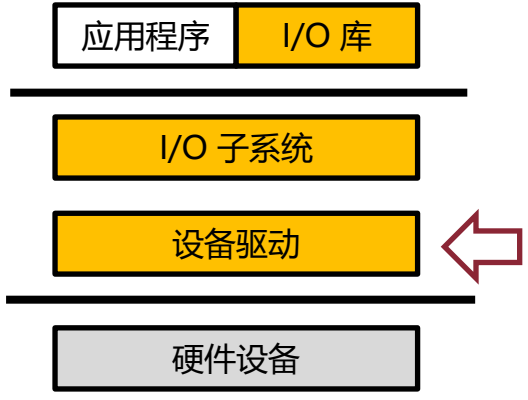
案例：io_uring (Linux 5.1)

- **问题：频繁的读写I/O请求导致高频模式切换开销**
- **提供“提交队列”和“完成队列”两个环形缓冲区**
  - 用户程序和内核共享队列，减少拷贝；允许批量处理多个I/O请求，减少切换
- **支持轮询模式，相比于异步通知机制有更低时延**



设备相关的I/O软件

设备驱动



操作系统：应对I/O设备做什么?

- **应用程序有访问I/O设备的需求**
  - printf
  - 浏览器
    - 网络
    - 绘制
  - 。。。
- **应用程序直接访问I/O设备的后果**
  - 无法写出portable的代码
    - printf
    - 浏览器
  - 应用之间需要互斥、同步、协商设备资源的使用
    - 复杂，不可能作对
    - 设备直连物理世界，可能造成物理性伤害

操作系统需要对设备进行抽象（虚拟化）

需求分析：I/O设备访问API设计

- **printf**
  - 希望向设备写入一个字符串
    - write(dev, “hello”);
- **浏览器**
  - 希望从某个TCP连接中读出数据
    - read(conn, buf, size);
  - 希望在当前的画布(canvas)上执行一些绘图指令
    - write(wm, draw_cmds, size);
- **I/O设备虽然复杂多样，但几乎都可以理解为字节的序列**
  - read 从设备某个指定位置读出数据
  - write 向设备某个指定位置写入数据
  - ioctl 读取/设置设备的状态
- **I/O设备的抽象：对文件的读写**

设备驱动

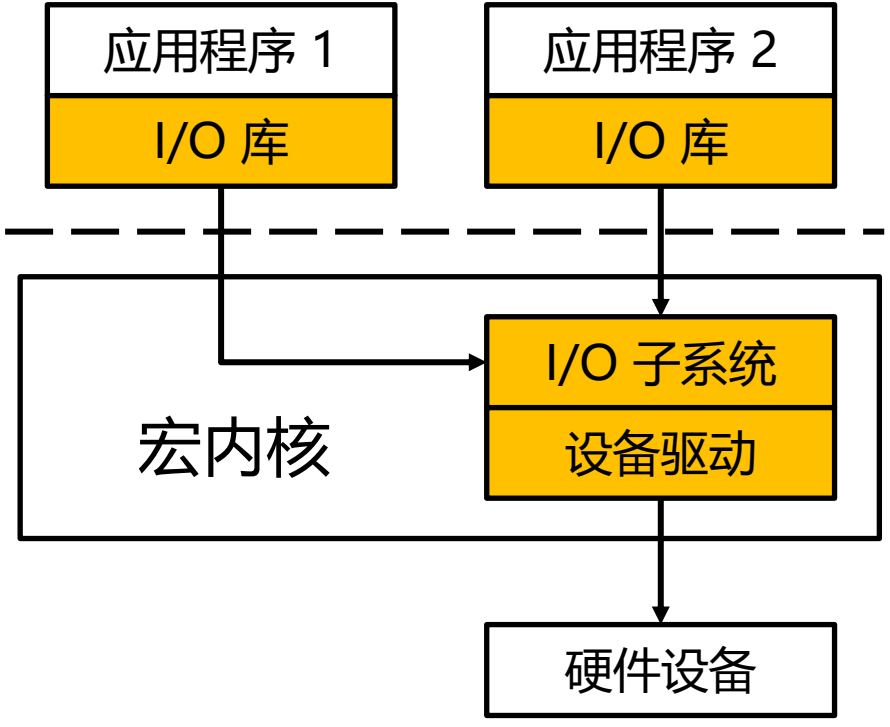
- **设备驱动**
  - 专门用于操作硬件设备的代码集合
  - 通常由硬件制造商负责提供
  - 驱动程序包含中断处理程序
- **驱动特点**
  - 和设备功能高度相关
  - 不同设备间的驱动复杂度差异巨大
  - 是操作系统 bugs 的主要来源

设备驱动的角色

- **机制 (mechanism) vs策略 (policy)**
  - 机制：设备提供的能力
  - 策略：如何使用这些能力
  - 例子
    - X server vs Window manager
  - Policy-free Drivers的典型特征
    - 提供同步和异步操作支持
    - 对外提供被多次打开的接口
    - 提供充分利用硬件能力的接口
    - 一些策略相关的支持

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



模块实例

- 简单module程序，驱动程序可以模块的方式加载

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
{
 printk(KERN_ALERT "Hello, world\n");
 return 0;
}
static void hello_exit(void)
{
 printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

指定加载时调用，此后换出内存

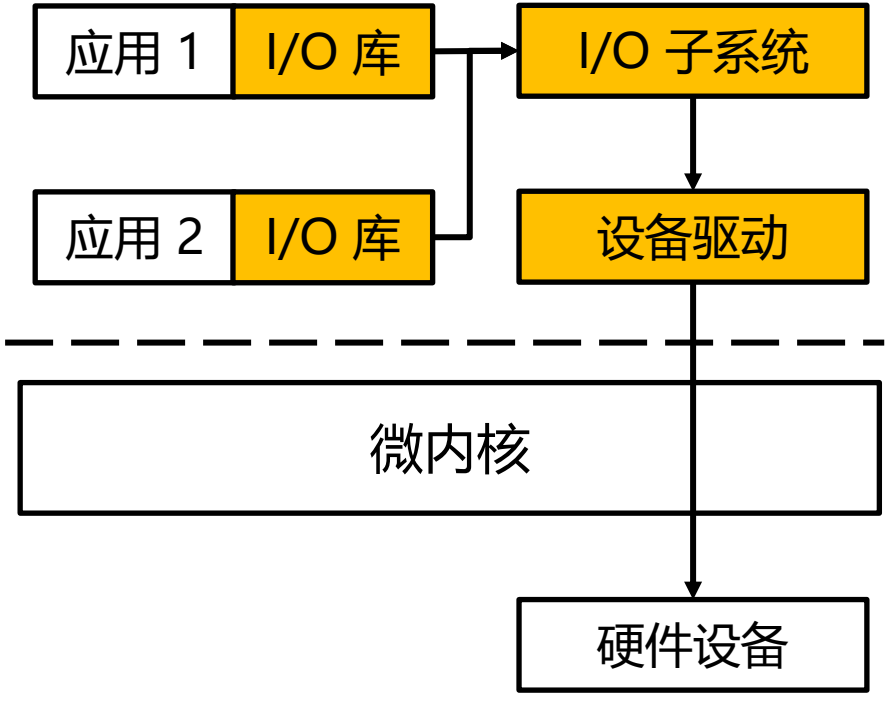
退出时执行，无此函数，则模块无法卸载

模块实例

```
% make
make[1]: Entering directory `/usr/src/linux-2.6.10'
  CC [M]  /home/ldd3/src/misc-modules/hello.o
  Building modules, stage 2.
  MODPOST
  CC      /home/ldd3/src/misc-modules/hello.mod.o
  LD [M]  /home/ldd3/src/misc-modules/hello.ko
make[1]: Leaving directory `/usr/src/linux-2.6.10'
% su
root# insmod ./hello.ko
Hello, world
root# rmmod hello
Goodbye cruel world
root#
```

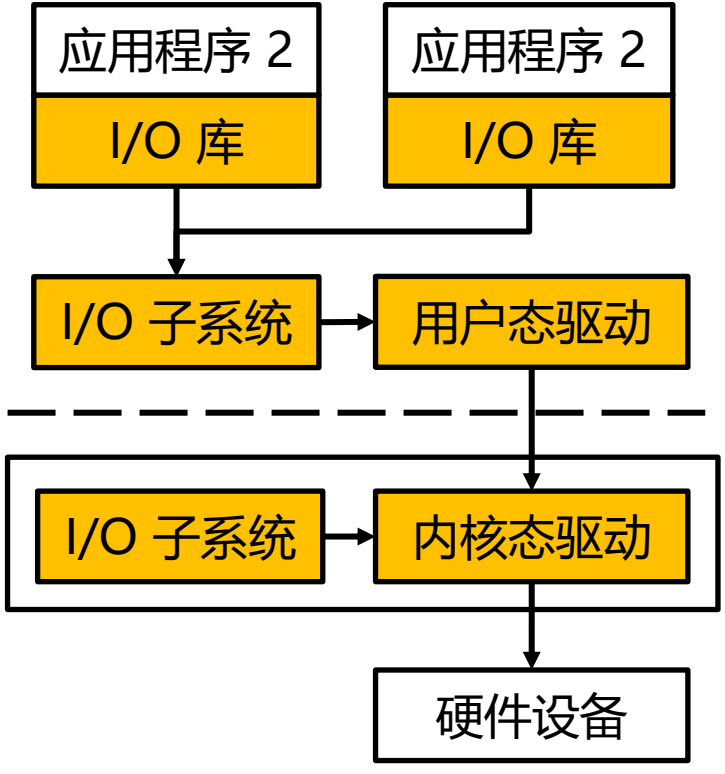
微内核I/O架构

- **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

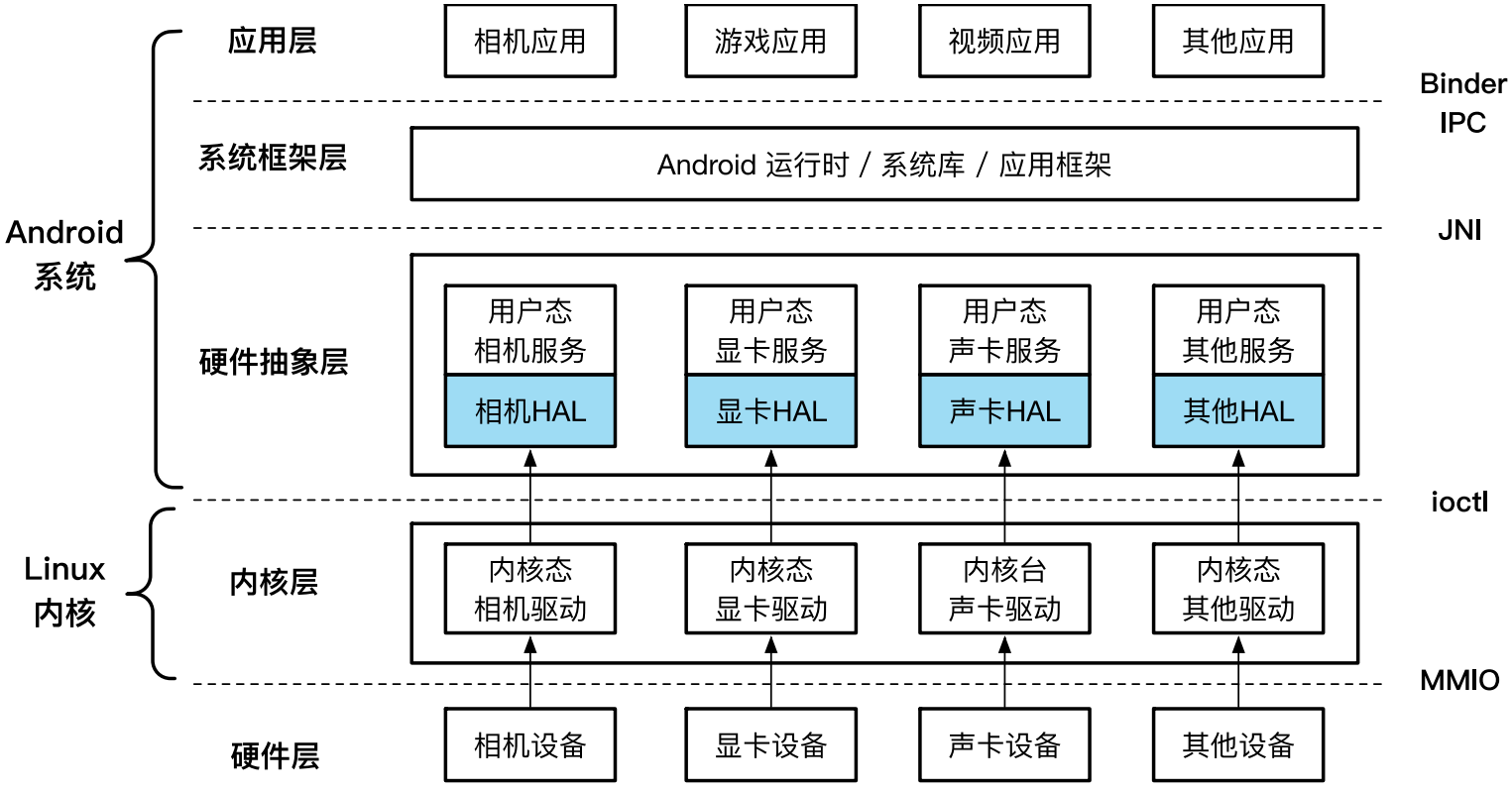


混合I/O架构

- **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


Android
系统
Linux
内核
应用层
相机应用
游戏应用
视频应用
其他应用
Binder
IPC
系统框架层
Android 运行时 / 系统库 / 应用框架
JNI
硬件抽象层
用户态
相机服务
相机HAL
用户态
显卡服务
显卡HAL
用户态
声卡服务
声卡HAL
用户态
其他服务
其他HAL
ioctl
内核层
内核态
相机驱动
内核态
显卡驱动
内核台
声卡驱动
内核态
其他驱动
MMIO
硬件层
相机设备
显卡设备
声卡设备
其他设备

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

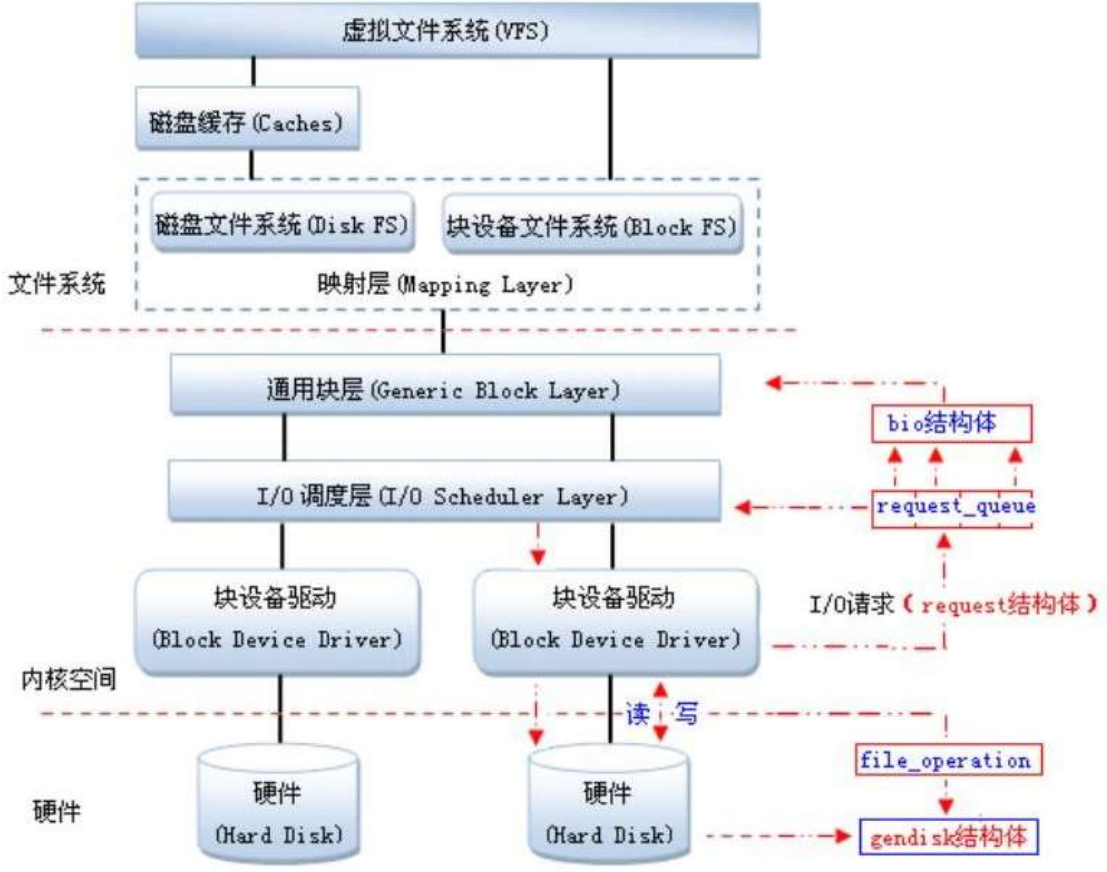
- **电源管理：**
  - 描述设备在系统中的拓扑结构（树形结构）
  - 保证能正确控制设备的电源，先关闭设备和再关闭总线
- **驱动开发者：**
  - 允许同系列设备的驱动代码之间的复用
  - 将设备和驱动联系起来，方便相互索引
- **系统管理员：**
  - 帮助用户枚举系统设备，观察设备间拓扑和设备的工作状态

案例：Linux驱动模型

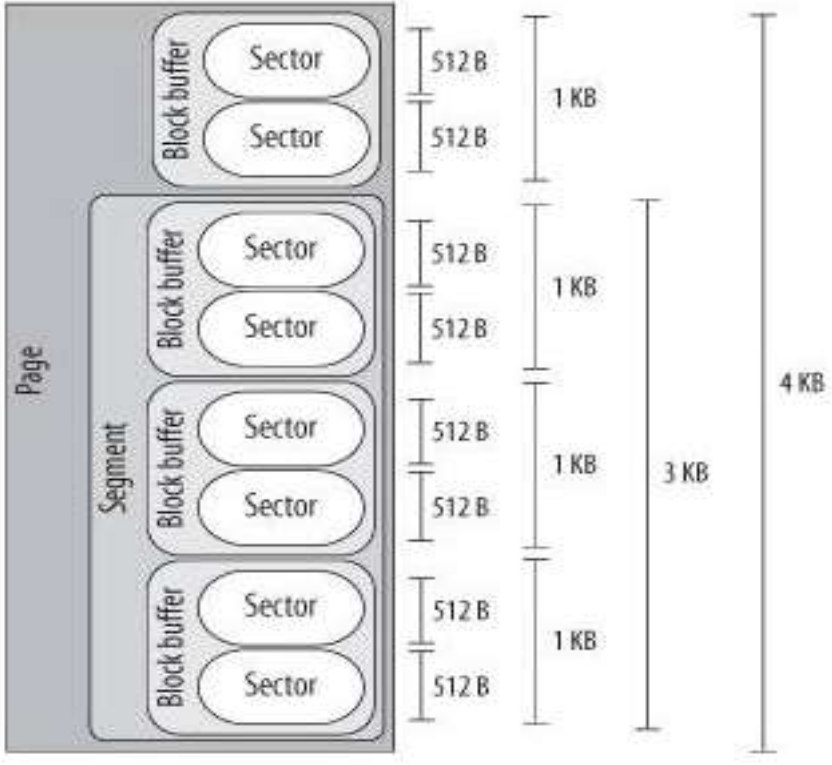
Linux Device Driver Model (LDDM)

- **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

linux 块设备驱动架构图



sector, block, segment, page的关系



sector

- 存放数据的连续区域单位
- 512字节

block

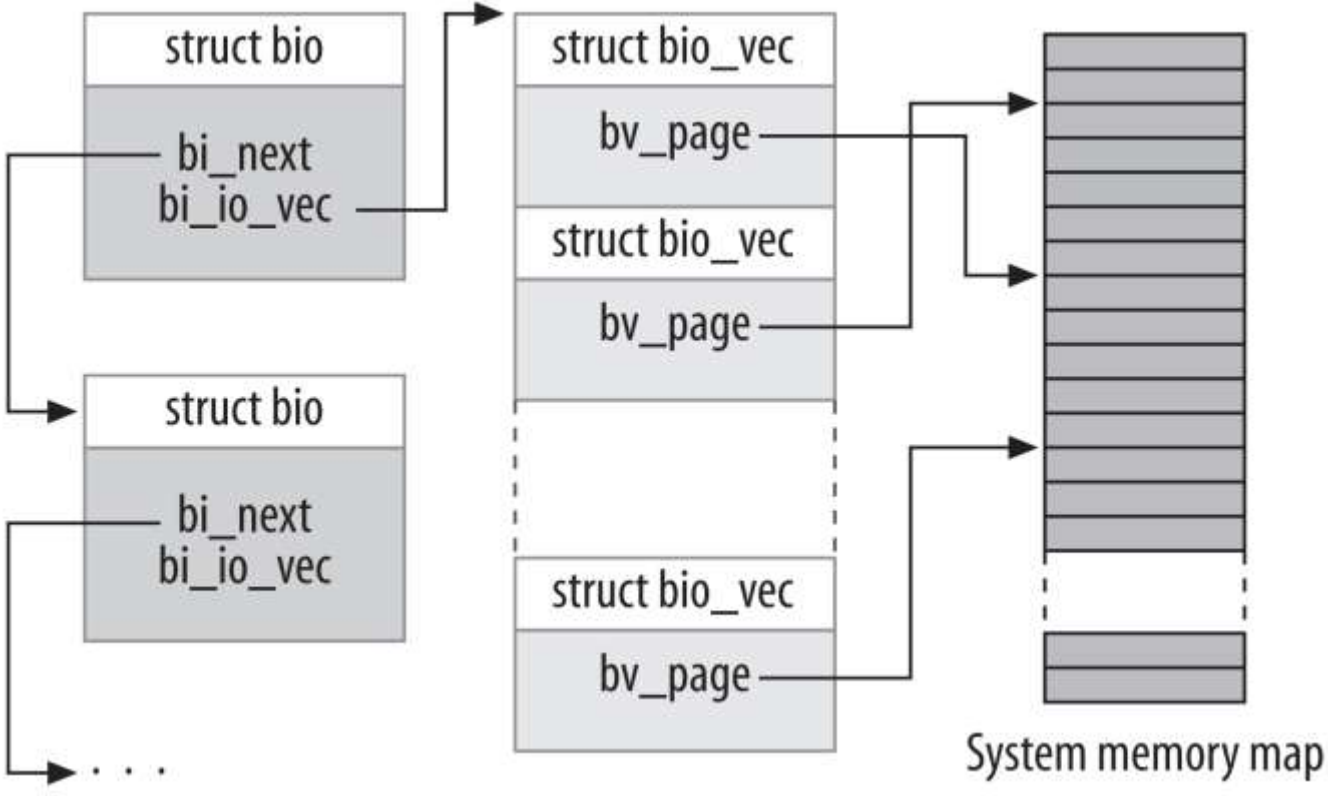
- 文件系统管理数据的单位
- sector的整数倍
- 不超过页的大小

BIO Structure

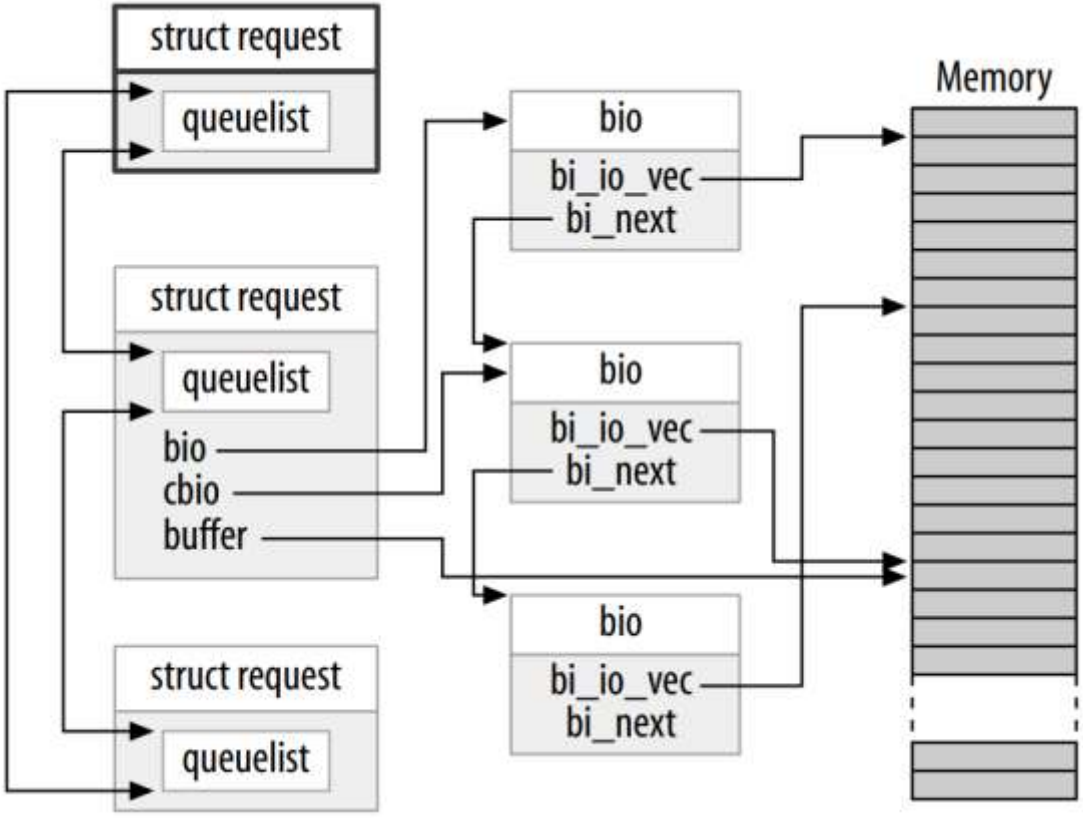
- sector_t bi_sector; /* 要传输的第一个扇区 */
- unsigned int bi_size;
- unsigned long bi_flags; /* 状态、命令等 */
- unsigned short bi_phys_segments;
- unsigned short bi_hw_segments;

```
struct bio_vec {
        struct page     *bv_page;
        unsigned int    bv_len;
        unsigned int    bv_offset;
};
```

BIO Structure



Request

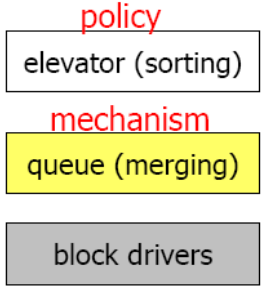

struct request
queuelist
struct request
queuelist
bio
cbio
buffer
struct request
queuelist
bio
bi_io_vec
bi_next
bio
bi_io_vec
bi_next
bio
bi_io_vec
bi_next
Memory

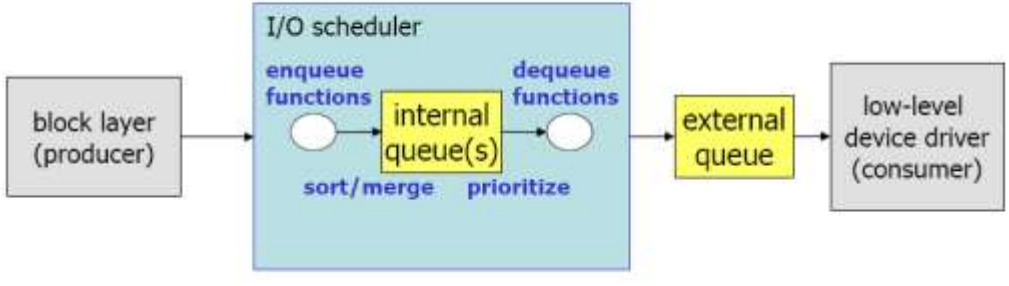
为什么需要I/O调度?

- **磁盘寻道在计算机系统中是最慢的操作**
  - 没有合适的I/O调度器，对系统性能影响非常大
- **I/O scheduler 可以安排磁头在一个方向上移动，减少 seek 次数**
  - 像电梯（操作系统中一般称这样的算法为电梯算法)
  - 在全局范围内获得高吞吐量
- **为了提高整个磁盘的吞吐量，则：**
  - 对请求重新排序，从而减少寻道时间
  - 合并请求而降低请求数量
- **针对在不同进程，提供公平性读写机会**

Linux I/O 调度框架

- **Linux elevator 是能附加的不同I/O调度器的抽象层，提供了一些函数供block layer使用**
- **机制：系统提供了一些函数可以对队列进行merge**
- **策略：sorting method, elevator提供**
  - 把请求放到队列中
  - 从队列中分发请求





Linux I/O 调度框架

- I/O调度的4种算法：
  - NOOP(硬件调度)
  - CFQ(完全公平排队I/O调度程序)
  - Deadline(截止时间调度程序，数据库友好)
  - AS(预料I/O调度程序)

```
cat /sys/block/sda/queue/scheduler
```

块设备驱动

- **和字符设备驱动程序完全不同**
- **文件系统以块为单位进行随机存取**
  - 不同的体系结构，对block的大小定义可能不同，比如有些系统定义一个 block 大小是 4096 字节
- **Block 设备一般和磁盘相关**
- **int register_blkdev(unsigned int major, const char *name);**
  - 在/proc/devices 中创建一入口
  - 如果第一个参数是 0，那么内核动态分配主设备号
- **int unregister_blkdev(unsigned int major, const char *name);**

设备的Major和Minor 数

- ls -l /dev/*

```
crw-rw-rw-   1 root    root      1,   3 Apr 11  2020 null
crw-rw-rw-   1 root    root      1,   5 Apr 11  2020 zero
crw-------   1 root    root     10,   1 Apr 11  2020 psaux
crw-------   1 root    root      4,   1 Oct 28 03:04 tty1
crw--w----   1 vcsa    tty       7,   1 Apr 11  2020 vcs1
```

- **不同的设备可以使用同一个驱动程序**
  - *如/dev/null* 和 */dev/zero* 主设备号都为1
- **主版本 → 哪类设备**
- **次版本 → 同类设备中哪个具体设备**
- **长度 32-bit：12 bits → major number； 20 bits → minor number.**
- **提取设备号：MAJOR(dev_t dev); MINOR(dev_t dev);**
- **构造dev_t： MKDEV(int major, int minor);**

struct block_device_operations

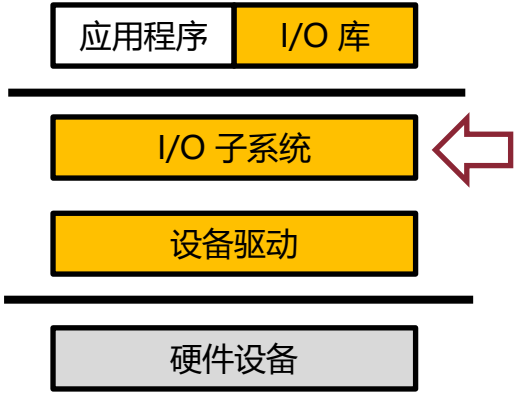
```
struct block_device_operations {
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*media_change) (struct gendisk *);
    int (*revalidate_disk) (struct gendisk *);
    int (*getgeo)(struct        block_device *,
                        struct  d_geometry *);
    struct module *owner;
};
```

- getgeo:

  得到块设备相关参数，如：柱面数、扇区数、开始扇区等

设备无关的I/O软件

I/O子系统



为什么需要I/O子系统?

- **根据不同需求和场景，出现了大量设备：**
  - 通信、存储、智能加速器、人机交互等
  - 数以千计的设备类型，个性千差万别
- **每种设备有自己的协议、规范：**
  - 如何标准化设备接口?
- **设备的异步不可预测性和慢速性：**
  - 如何提高应用程序的设备读写效率?
- **设备的不可靠性（介质失效或传输错误）：**
  - 如何得知设备的状态并修复错误?

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：

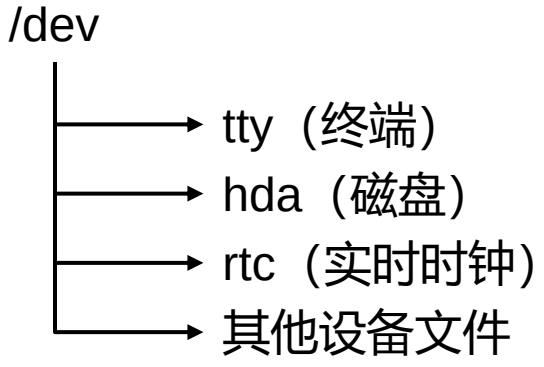
```
FILE fd = fopen(“/dev/something”,“rw”);
for (int i = 0; i < 10; i++) {
        fprintf(fd,“Count %d\n”,i);
}
close(fd);
```

- **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);
```



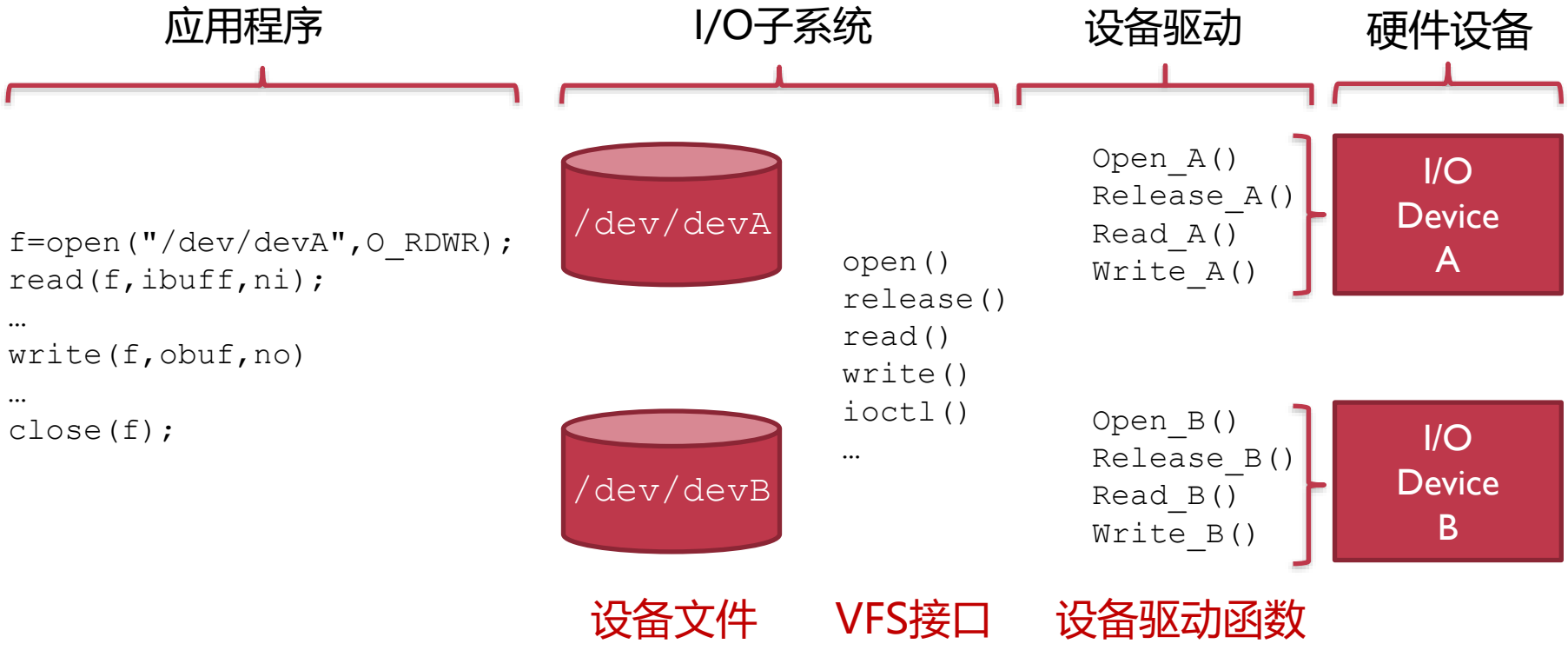
设备操作的专用接口：ioctl

- 应用程序和驱动程序事先协商好“操作码”和对应语义

```c
#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;
}
```

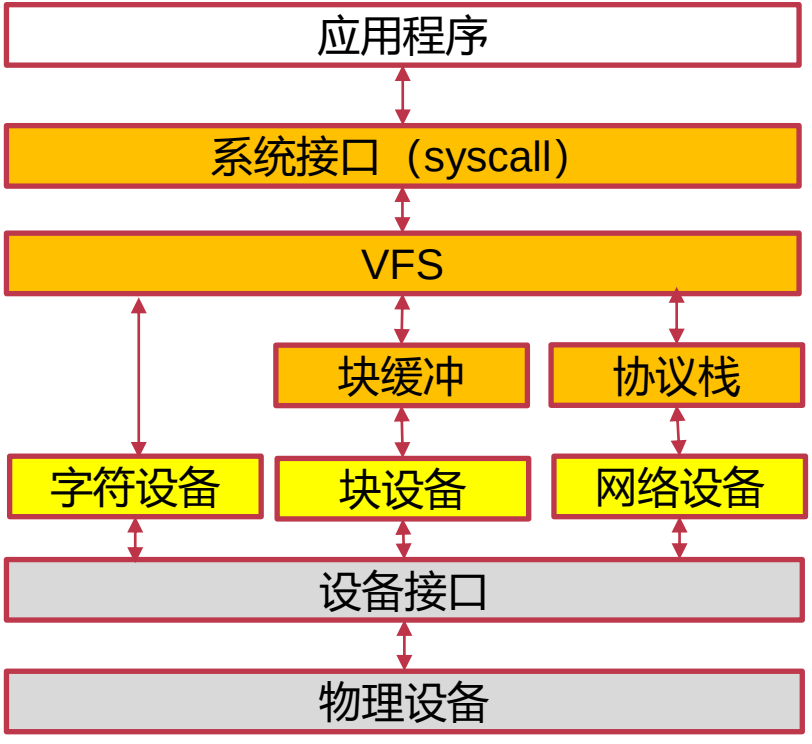
```c
#define LED_ALL_ON     _IO('L',0x1234)
#define LED_ALL_OFF    _IO('L',0x5678)

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,
};
```

设备文件操作与设备驱动函数的对接



设备的逻辑分类

- **Linux设备分类**
  - 字符设备
  - 块设备
  - 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）

内存映射文件：mmap访问磁盘

- 可减少系统调用次数
- 可减少数据的拷贝次数

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
    const char *message = "memory mapped file";

    int fd = open("mmap.c", O_RDWR);

    // 标准I/O：用lseek()寻址，再用write()写入数据
    lseek(fd, 0x10, SEEK_SET);
    write(fd, message, strlen(message));

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>

int main()
{
    struct stat sbuf;
    const char *message = "memory mapped file";

    int fd = open("mmap.c", O_RDWR);
    fstat(fd, &sbuf); // 获取文件长度

    // 内存映射文件：用指针直接寻址，将数据写入内存
    char *data = mmap((caddr_t)0, sbuf.st_size,
        PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    strncpy(data+0x10, message, strlen(message));

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

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

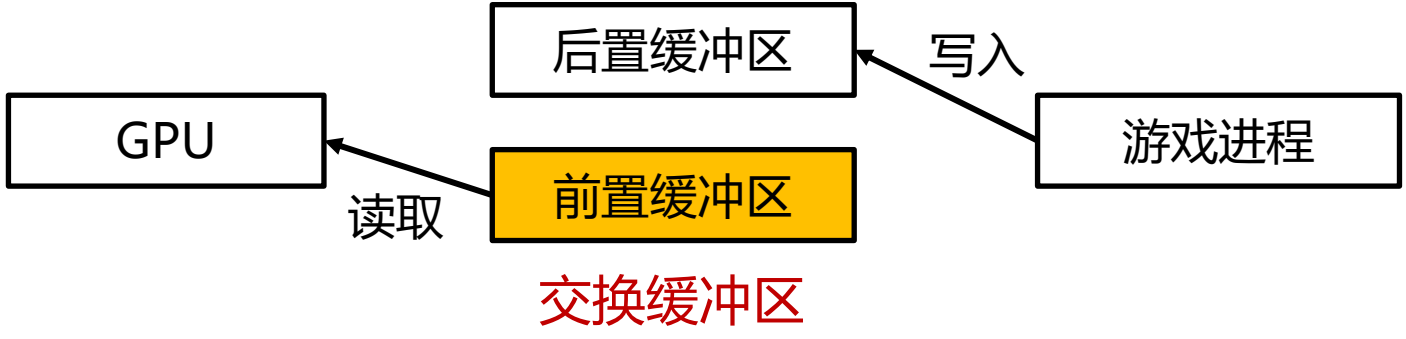
- **问题：**
  - 读写性能不匹配：慢速的存储设备 vs. 高速的CPU
  - 读写粒度不匹配：小数据的访问存在读写放大的问题
- **解决方法：**
  - 开辟内存缓冲区，避免频繁读写I/O
- **单缓冲区例子：Linux的page cache**

设备的缓冲管理：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**

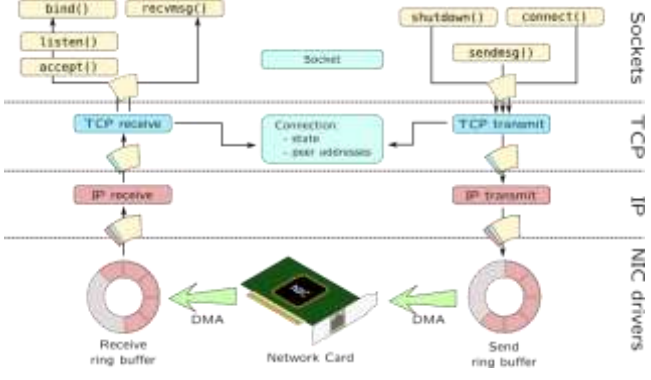
设备的缓冲管理：双缓冲区

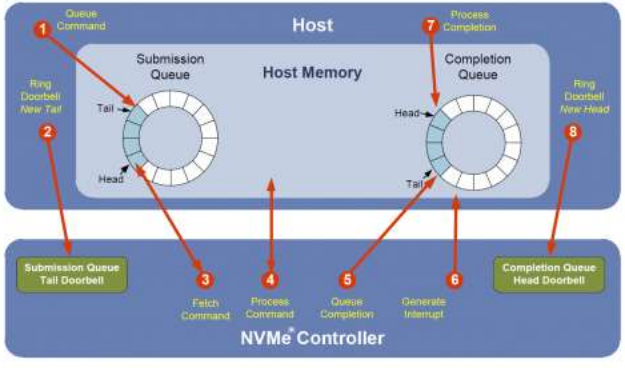
- **维护两个缓冲区，轮流使用**
- **第一个缓冲区被填满但没被读取前，使用第二个缓冲区填充数据**
- **双缓冲区例子：显存刷新，防止屏幕内容出现闪烁或撕裂**
  - 前置缓冲区被读取后，通过“交换” 将前置和后置身份互换
- **游戏中甚至启用三重缓冲**



设备的缓冲管理：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

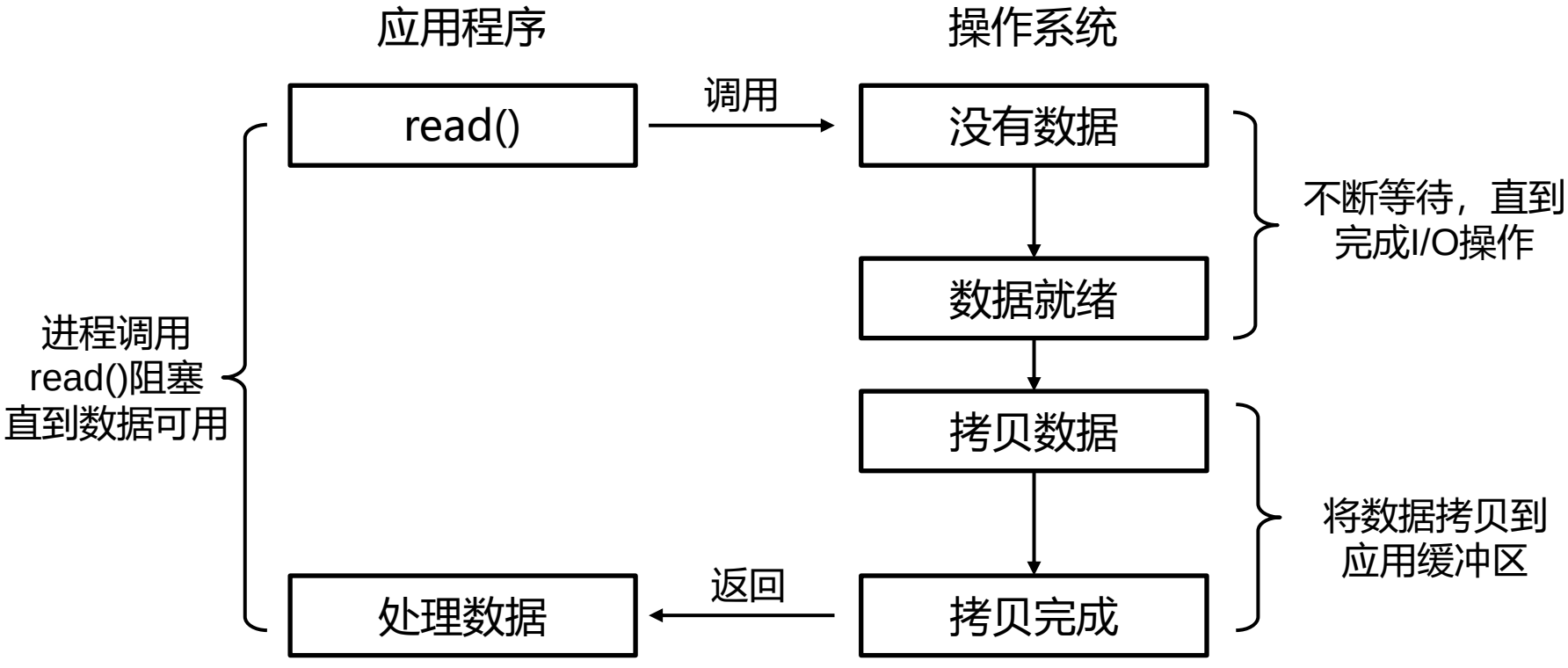
- **多种缓冲区实现方式**
  - 单缓冲区：块设备的缓冲区
  - 双缓冲区：常用于游戏或流媒体的显存同步
  - 环形缓冲区：网卡的数据交互和NVMe存储的命令交互

- **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

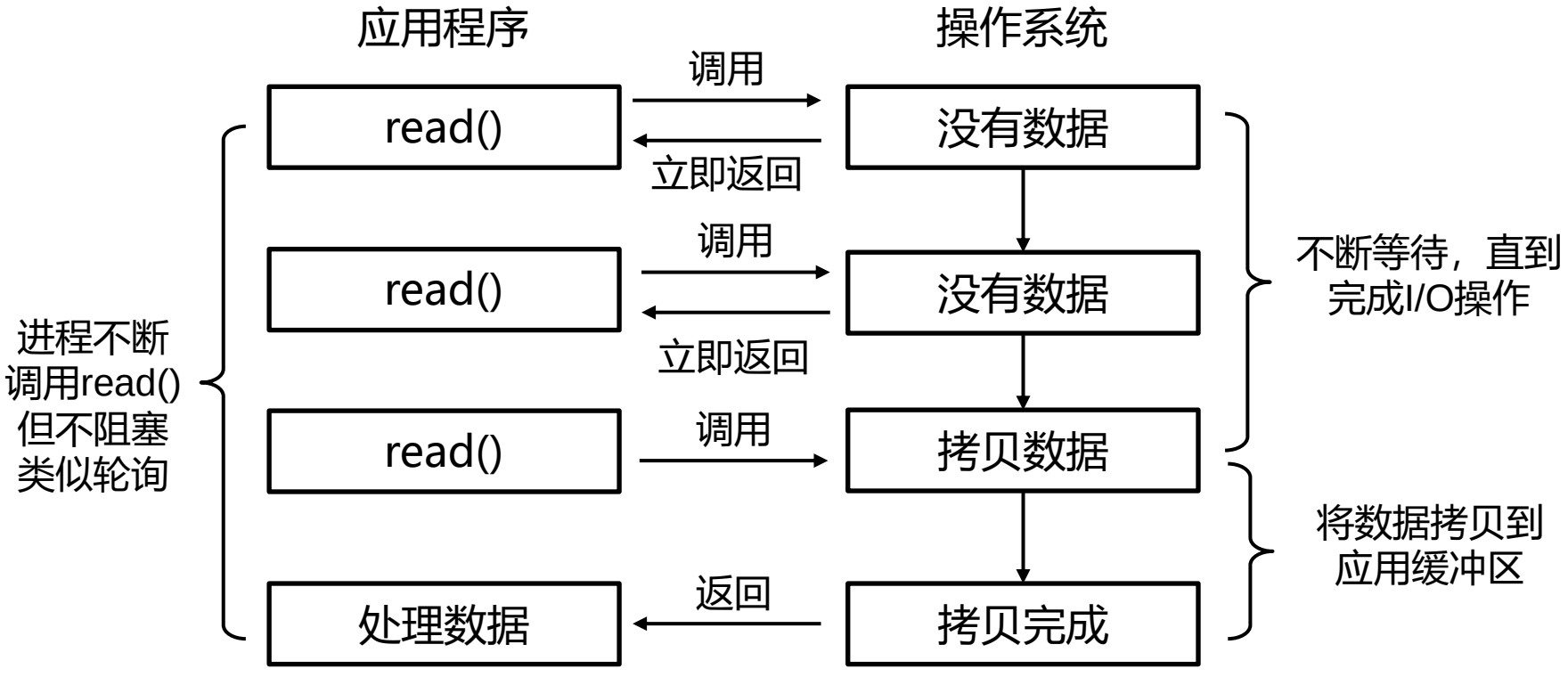
问题：如何同时监听多个设备？

- **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

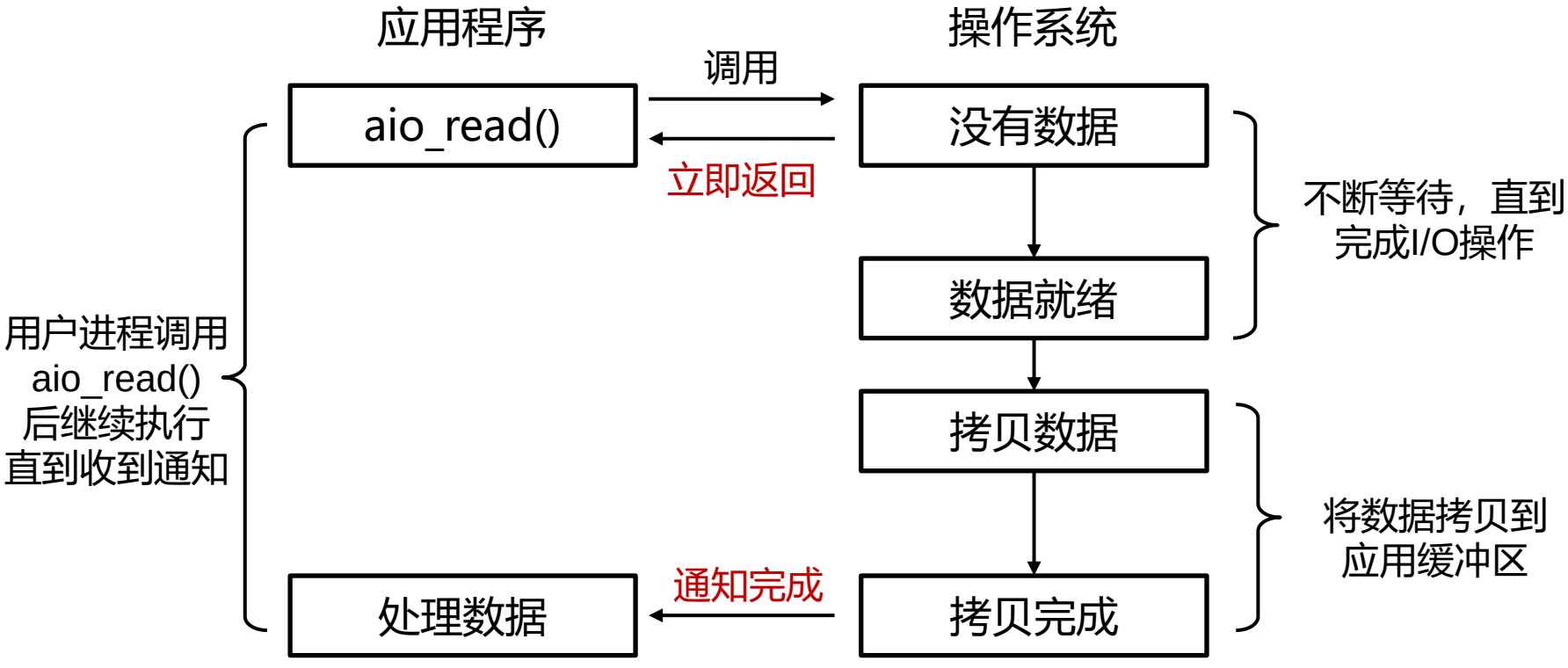
I/O模型：阻塞I/O模型


应用程序
操作系统
read()
调用
没有数据
数据就绪
拷贝数据
拷贝完成
返回
处理数据
不断等待，直到
完成I/O操作
将数据拷贝到
应用缓冲区
进程调用
read()阻塞
直到数据可用

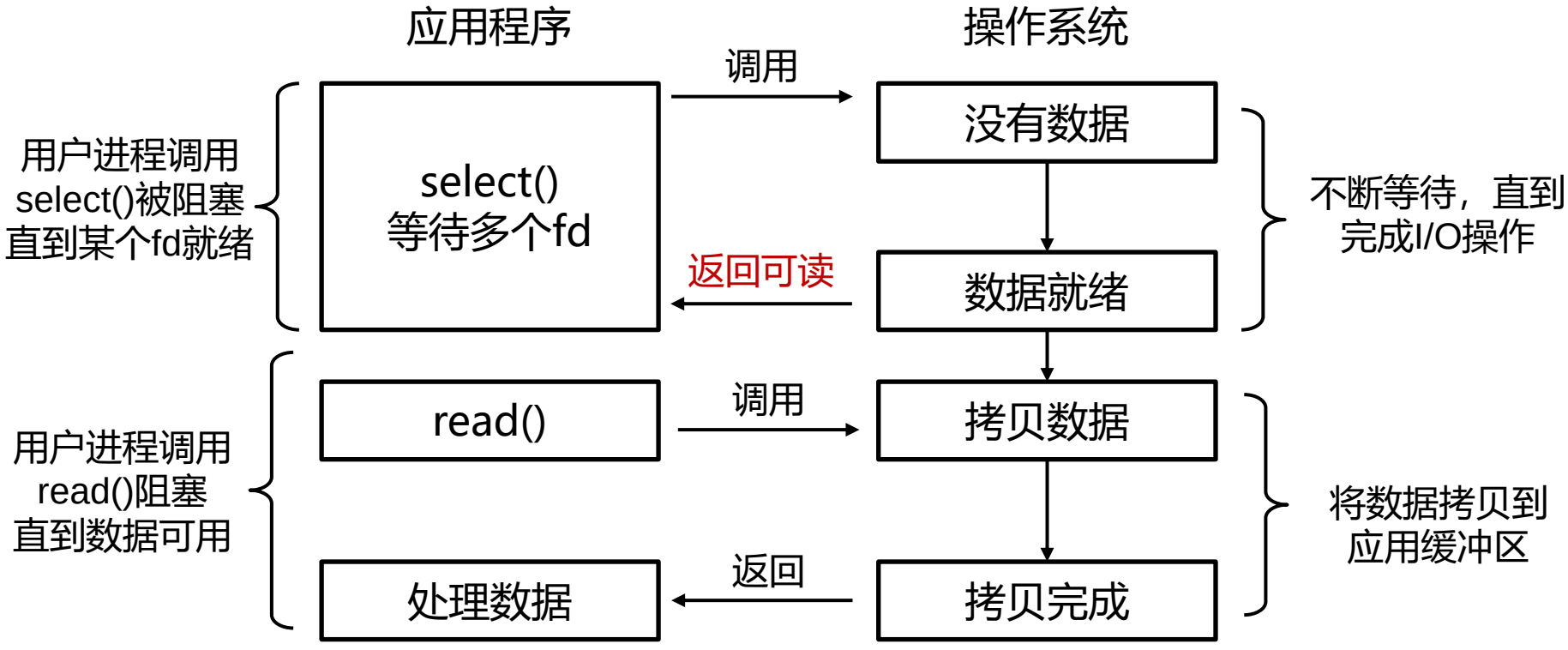
I/O模型：非阻塞I/O模型



I/O模型：异步I/O模型


应用程序
操作系统
调用
aio_read()
立即返回
没有数据
数据就绪
拷贝数据
拷贝完成
通知完成
处理数据
不断等待，直到
完成I/O操作
将数据拷贝到
应用缓冲区
用户进程调用
aio_read()
后继续执行
直到收到通知

I/O模型：I/O多路复用模型

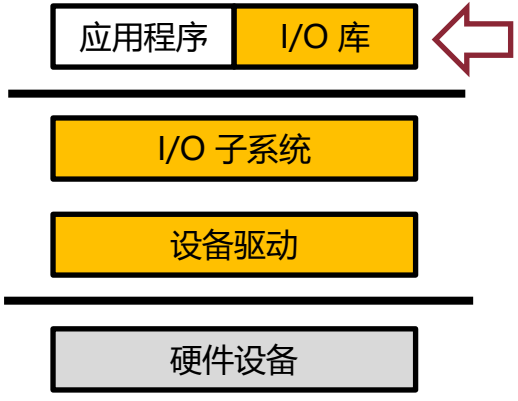


I/O模型小节

- **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**

应用程序员视角

I/O库



I/O库

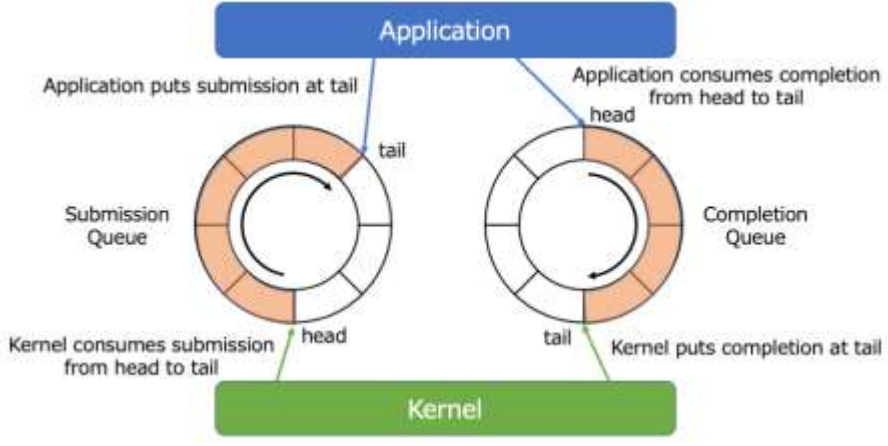
- **以共享库的形式，和应用程序直接链接**
- **简化应用程序I/O开发的复杂度**
- **提供更好的性能和更灵活的I/O管理能力**
- **例子：**
  - glibc：提供用户态I/O缓冲区管理
  - Linux AIO和io_uring：支持异步I/O

案例：glibc的buffer I/O

- **fread()/fwrite()和read()/write() 有何区别?**
  - 前者是I/O库接口（函数调用），后者是VFS接口（系统调用）
  - 使用用户态缓冲区，减少模式切换次数
- **缓冲模式配置：**
  - 全缓冲（_IOFBF）：缓冲区满了才flush缓冲区
  - 行缓冲（_IOLBF）：遇到换行符就flush缓冲区
  - 无缓冲（_IONBF）：和直接调用read()/write()效果一样

案例：io_uring (Linux 5.1)

- **问题：频繁的读写I/O请求导致高频模式切换开销**
- **提供“提交队列”和“完成队列”两个环形缓冲区**
  - 用户程序和内核共享队列，减少拷贝；允许批量处理多个I/O请求，减少切换
- **支持轮询模式，相比于异步通知机制有更低时延**



中断子系统

LINUX的上下半部

Linux上下半部

- **面临问题**
  - 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

思考：
为什么要共享IRQ?

- **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**

内核线程（Kernel Threads）

- **始终运行在内核态**
  - 没有用户空间上下文
  - 和用户进程一样被调度器管理
- **中断线程化*（*threaded interrupt handlers*）**
  - Linux 2.6.30引入
  - 每个中断线程都有自己的上下文

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

谢谢！