Operating System Principle, OS

2025年秋. 操作系统原理

第8章 设备管理系统

课 程 组：邹德清, 李珍, 李志, 苏曙光

企业教师：华为认证专家（鸿蒙方向）

第8章 设备管理(输入/输出管理)

- 内容
  - 设备管理概述
  - 设备驱动程序
  - 缓冲技术
  - 设备分配
  - SPOOL技术
- 重点/难点
  - 理解“设备是文件”概念
  - 掌握Linux设备驱动程序开发技术
  - 掌握缓冲的作用和Linux缓冲实现机制

8.1 设备管理概念

设备管理概念

- 设备/外部设备/外设/I/O设备
  - 除CPU和内存之外的部件。通过接口连接到CPU的三总线。












设备类型和特征

- 按交互对象分类
  - 与人交互：显示设备、键盘、鼠标、打印机
  - 与CPU交互：磁盘、磁带、传感器、控制器
  - 计算机间交互：网卡、调制解调器
- 按交互方向分类
  - 输入设备：鼠标、键盘、扫描仪、传感器
  - 输出设备：显示设备、打印机、控制器
  - 双向设备：硬盘、软盘、网卡、调制解调器
- 按数据传输速率
  - 低速(KB)：键盘、鼠标、MODEM、传感器
  - 中速(MB)：打印机、串口、软盘
  - 高速(GB)：网卡、硬盘、显卡

设备类型和特征

- 按信息组织特征分类
  - 字符设备
    - 传输的基本单位是字符。例：键盘、串口
  - 块设备
    - 传输的基本单位是块(Block)。例：硬盘
    - 一般支持文件系统
  - 网络设备
    - 采用socket套接字接口访问
    - 在全局空间有唯一名字，如eth0、eth1
  - 对应用影响：驱动程序的结构和接口有不同

设备管理功能

- 设备管理的目标
  - （1）提高设备读写效率
    - 设备缓冲机制
  - （2）提高设备的利用率
    - 设备分配（设备调度）
  - （3）为用户提供统一接口
    - 实现设备对用户透明

设备管理功能

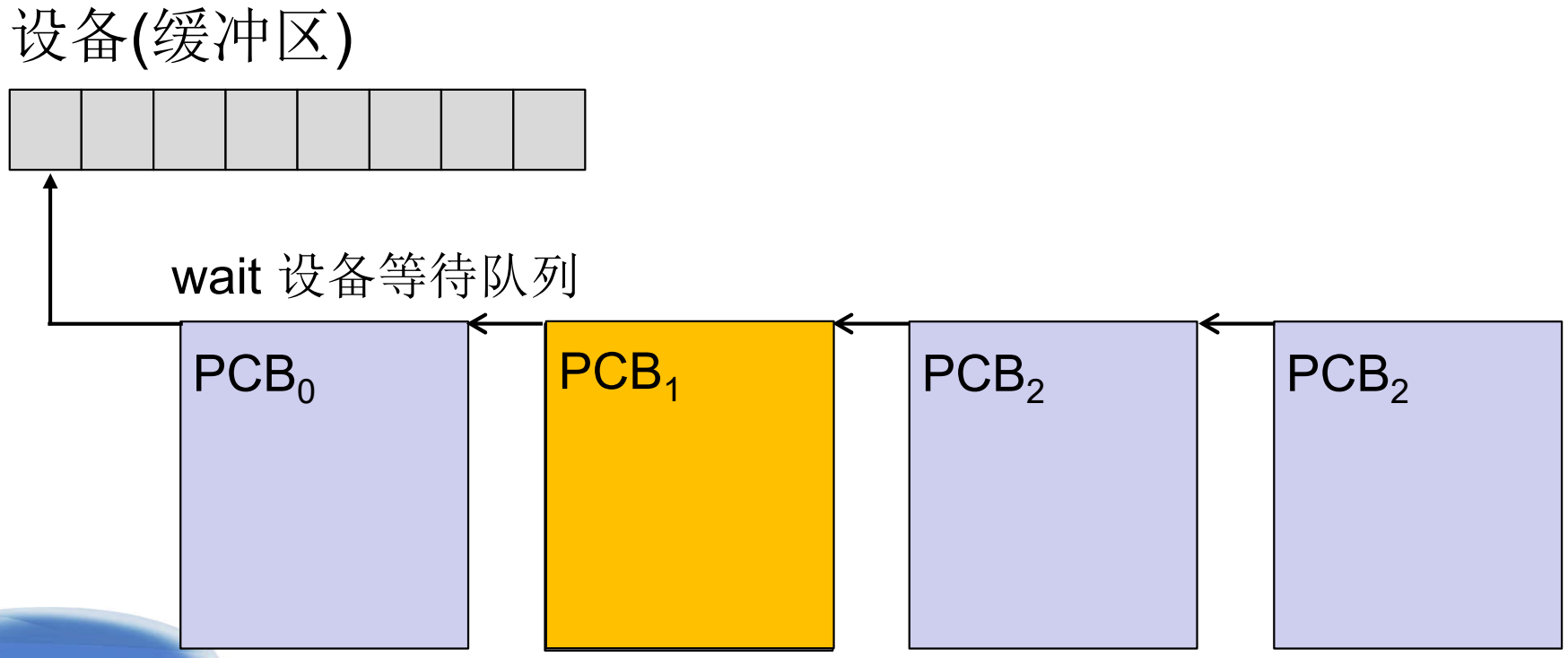
- 设备管理的功能
  - 1）状态跟踪
  - 2）设备分配
  - 3）设备映射
  - 4）设备控制/设备驱动
  - 5）缓冲区管理

设备管理功能>状态跟踪

- 状态跟踪
  - 记录设备的属性、状态、接口及进程访问信息
    - 设备控制块（Device Control Block，DCB)
      - 设备名
      - 设备地址
      - 设备状态
      - 命令转换表[write→，read→，…]
      - 正访问进程指针
      - 请求进程队列指针
      - ……

设备管理功能>设备分配

- 功能
  - 按一定策略为进程安全地分配和管理各种设备。
    - 按相应算法把设备分配给请求该设备的进程，并把未分到设备的进程放入设备等待队列。



设备管理功能>设备映射

- 设备映射
  - ■把应用程序中的逻辑设备转换到真实的物理设备
    - ◆设备的逻辑名到物理名的转换
- 设备物理名
  - ■Linux：实际安装设备的主/次设备号
    - ◆端口，内存地址，操作接口
  - ■Windows：内核空间的设备名或GUID
    - ◆端口，内存地址，操作接口

设备管理功能>设备映射

- 设备逻辑名/友好名(Friendly Name)
  - 用户编程时使用的名字(文件名/设备文件名)
  - 例：Linux： /dev/test

```
int testdev = open("/dev/test",O_RDWR);
if ( testdev == -1 )          ↑↳主/次设备号
{
    printf("Cann't open file ");
    exit(0);
}
Read(testdev,  lpBuffer, ……..);
Write(testdev,  lpBuffer, ……..);
```

设备管理功能>设备映射

- 设备逻辑名/友好名(Friendly Name)
  - 用户编程时使用的名字(文件名/设备文件名)
  - 例：Windows：\\.\MyDevice

```
hDevice = CreateFile("\\\\.\\MyDevice",   →内核名：??\Device\设备名
                     GENERIC_WRITE|GENERIC_READ,
                     FILE_SHARE_WRITE | FILE_SHARE_READ,
                     NULL,
                     OPEN_EXISTING,
                     0,
                     NULL);
ReadFile( hDevice,  lpBuffer, ……..);
WriteFile(hDevice,  lpBuffer, ……..);
……..
```

设备管理功能>设备映射

- 设备映射
  - 把应用程序中的逻辑设备转换到真实的物理设备
    - 设备的逻辑名到物理名的转换
- 设备物理名
  - Linux：实际安装设备的主/次设备号
    - 端口，内存地址，操作接口
  - Windows：内核空间的设备名或GUID
    - 端口，内存地址，操作接口
- 设备独立性/设备无关性

设备管理功能>设备映射

- 设备独立性/设备无关性
  - 用户程序中使用统一接口访问设备，而不用考虑物理设备的特殊结构或操作方式。

```
int testdev = open("/dev/test",O_RDWR);
Read(testdev,  lpBuffer, …… );
Write(testdev,  lpBuffer, …… );
```

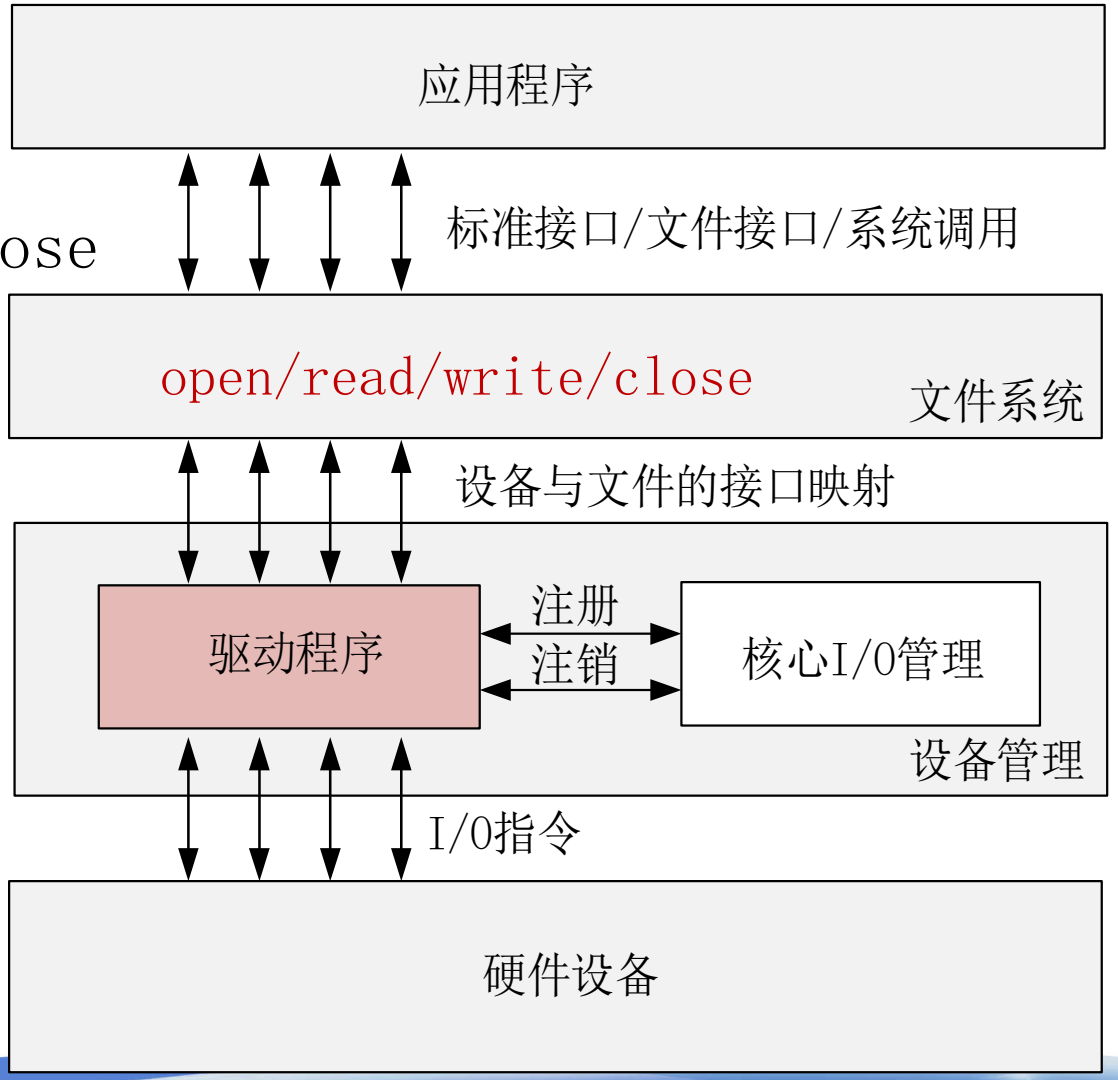
```
hDevice = OpenFile("\\\\.\\MyDevice",
                   GENERIC_WRITE|GENERIC_READ,
                   FILE_SHARE_WRITE | FILE_SHARE_READ,
                   NULL,  OPEN_EXISTING, 0, NULL);
ReadFile( hDevice,  lpBuffer, …… );
WriteFile(hDevice,  lpBuffer, …… );
```

设备管理功能> I/O缓冲区管理

- I/O缓冲区管理
  - 开辟和管理I/O缓冲区
    - 隔离CPU与外设
  - 提高读/写效率

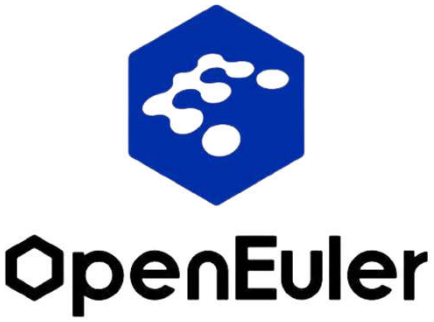
设备管理功能>设备驱动

- 设备驱动
  - 为应用程序提供设备操作接口，把来自应用的操作请求转为对设备的I/O操作指令
    - 应用采用文件接口操作
      - open/read/write/close
    - “文件是设备的抽象”



设备管理功能>设备驱动

- 设备驱动程序的特点
  - 设备驱动程序与硬件密切相关。
  - 设备必须要配置驱动程序
  - 驱动程序一般由设备厂商根据操作系统要求编写。



8.2 设备驱动

Linux字符设备驱动示例

- 例：应用程序(功能：读/写设备RWDevState的状态)

```
1  main()
2  {
3      int fd, nDevState;
4      //打开设备，设备文件名是：/dev/RWDevState
5      fd = open("/dev/RWDevState", O_RDWR);
6      //读设备的状态DevState
7      read(fd, &nDevState, sizeof(int));
8      printf("The Steate of Device is %d\n", nDevState);
9      //更新设备的状态DevState
10     nDevState = 100;
11     write(fd, &nDevState, sizeof(int));
12     //读设备的状态DevState
13     read(fd, &nDevState, sizeof(int));
14     printf("The Steate of Device is %d\n", nDevState);
15     //关闭设备
16     close(fd);
17 }
```

Linux字符设备驱动示例

- 例：驱动程序(功能：读/写设备RWDevState的状态)

```
13  static int chr_open(struct indoe * pinode, struct file *pfile)
14  {  //应用：打开设备 fd = open("/dev/RWDevState ")
15      MOD_INC_USE_COUNT;
16      return 0;
17  }
18  static int chr_read(struct file *pfile, char *buf, int len, int *off)
19  {  //应用：读设备 read( fd, &nDevState )
20      //将 nDevState 从内核空间复制到用户空间
21      copy_to_user(buf, &nDevState, sizeof(int));
22      return 0;
23  }
24  static int chr_write(struct file *pfile, const char *buf, int len, int *off)
25  {  //应用：写设备 write( fd , &nDevState )
26      //将 nDevState 从用户空间复制到内核空间
27      copy_from_user(&nDevState, buf, sizeof(int));
28      return 0;
29  }
30  static int chr_release(struct indoe * pinode, struct file *pfile)
31  {  //应用：关闭设备 close( fd ) → release( fd ) //libc
32      MOD_DEC_USE_COUNT;
33      return 0;
34  }
```

Linux字符设备驱动示例

- 例：驱动程序(功能：读/写设备RWDevState的状态)
  - 定义设备操作接口与文件操作接口之间的映射

```
1  static const  struct file_operations  MyFops  =
2  {
3      .read              = chr_read,
4      .write             = chr_write,
5      .release close()   = chr_release,
6      .open              = chr_open,
7  };
```

```
1  struct file_operations {  // 文件操作结构体（POSIX）
2      struct module *owner;
3      int (*llseek) (struct file *, int, int);
4      int (*read) (struct file *, char __user *, int, loff_t *);
5      int (*write) (struct file *, char __user *, int, loff_t *);
6      int (*poll) (struct file *, struct poll_table_struct *);
7      int (*ioctl) (struct inode *, struct file *,  int,  long);
8      int (*mmap) (struct file *, struct vm_area_struct *);
9      int (*open) (struct inode *, struct file *);
10     int (*flush) (struct file *, fl_owner_t id);
11     int (*release) (struct inode *, struct file *);
12     ......
13 };
```

Linux字符设备驱动示例

- 例：驱动程序(功能：读/写设备RWDevState的状态)
  - 实现设备的注册函数

```
35  //定义注册函数
36  static int DevInit(void)
37  {
38      int ret;
39      ret = register_chrdev(MAJOR_NUM,"RWDevState",&MyFops);
40      printk("RWDevState register success\n");
41      return ret;
42  }

50  module_init(DevInit);
```

  - 安装驱动程序

```
# insmod  RWDevState.ko
```

Linux字符设备驱动示例

- 编译驱动程序

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

- 创建设备文件

mknod /dev/RWDevState c 252 0

```
4  //打开设备，设备文件名是：/dev/RWDevState
5  fd = open("/dev/RWDevState", O_RDWR);
6  //读设备的状态DevState
7  read(fd, &nDevState, sizeof(int));
```

Linux字符设备驱动示例

- 例：应用程序（读/写设备的状态）

#mknod /dev/RWDevState c 252 0

```
1  main()
2  {
3      int fd, nDevState;
4      //打开设备，设备文件名是：/dev/RWDevState
5      fd = open("/dev/RWDevState", O_RDWR);
6      //读设备的状态DevState
7      read(fd, &nDevState, sizeof(int));
8      printf("The Steate of Device is %d\n", nDevState);
9      //更新设备的状态DevState
10     nDevState = 100;
11     write(fd, &nDevState, sizeof(int));
12     //读设备的状态DevState
13     read(fd, &nDevState, sizeof(int));
14     printf("The Steate of Device is %d\n", nDevState);
15     //关闭设备
16     close(fd);
17 }
```

Linux字符设备驱动示例

- 例：驱动程序(功能：读/写设备RWDevState的状态)
  - 实现设备的注销函数

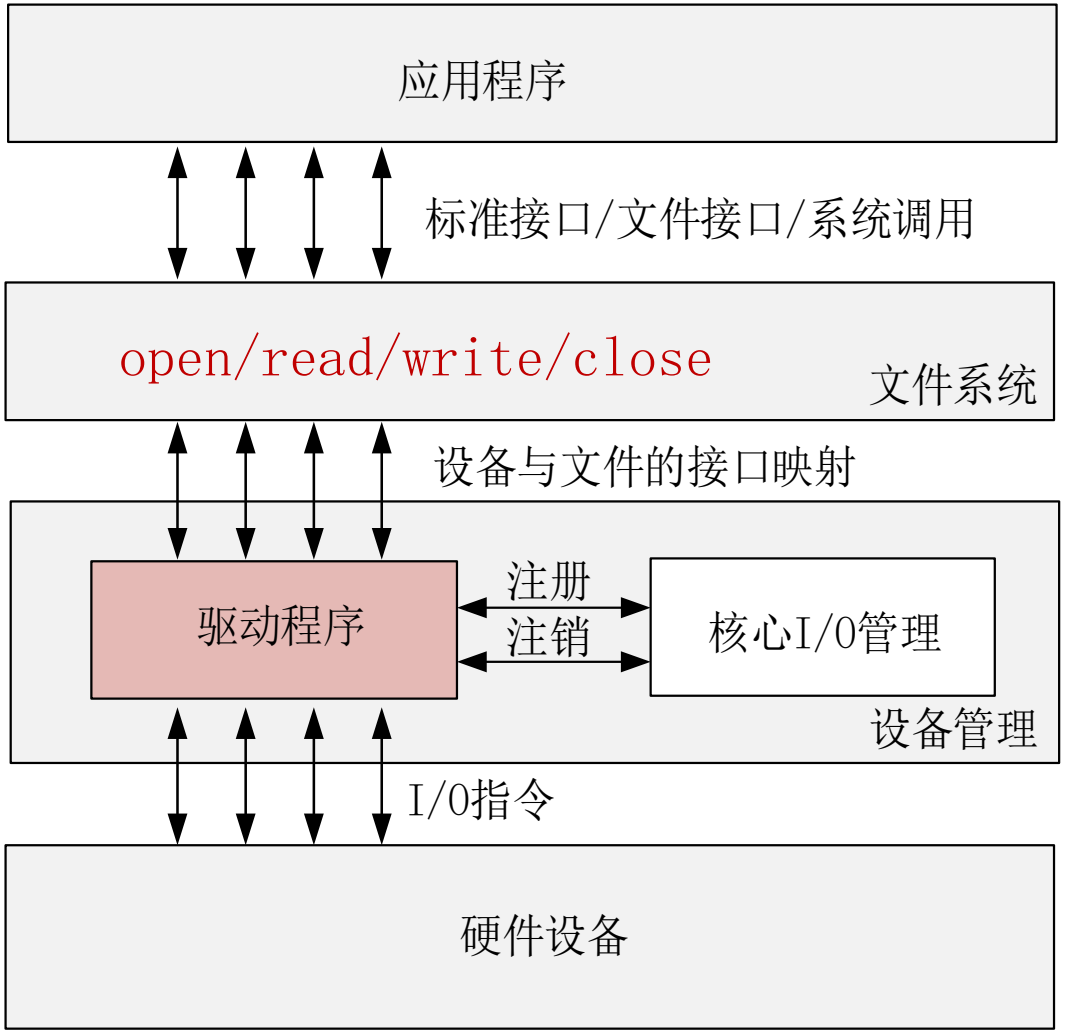
```
43  //定义注销函数
44  static void DevExit(void)
45  {
46      int ret;
47      ret = unregister_chrdev(MAJOR_NUM, "RWDevState");
48      printk("RWDevState unregister success\n");
49  }

51  module_exit(DevExit);
```

- 删除驱动程序

  # rmmod RWDevStat

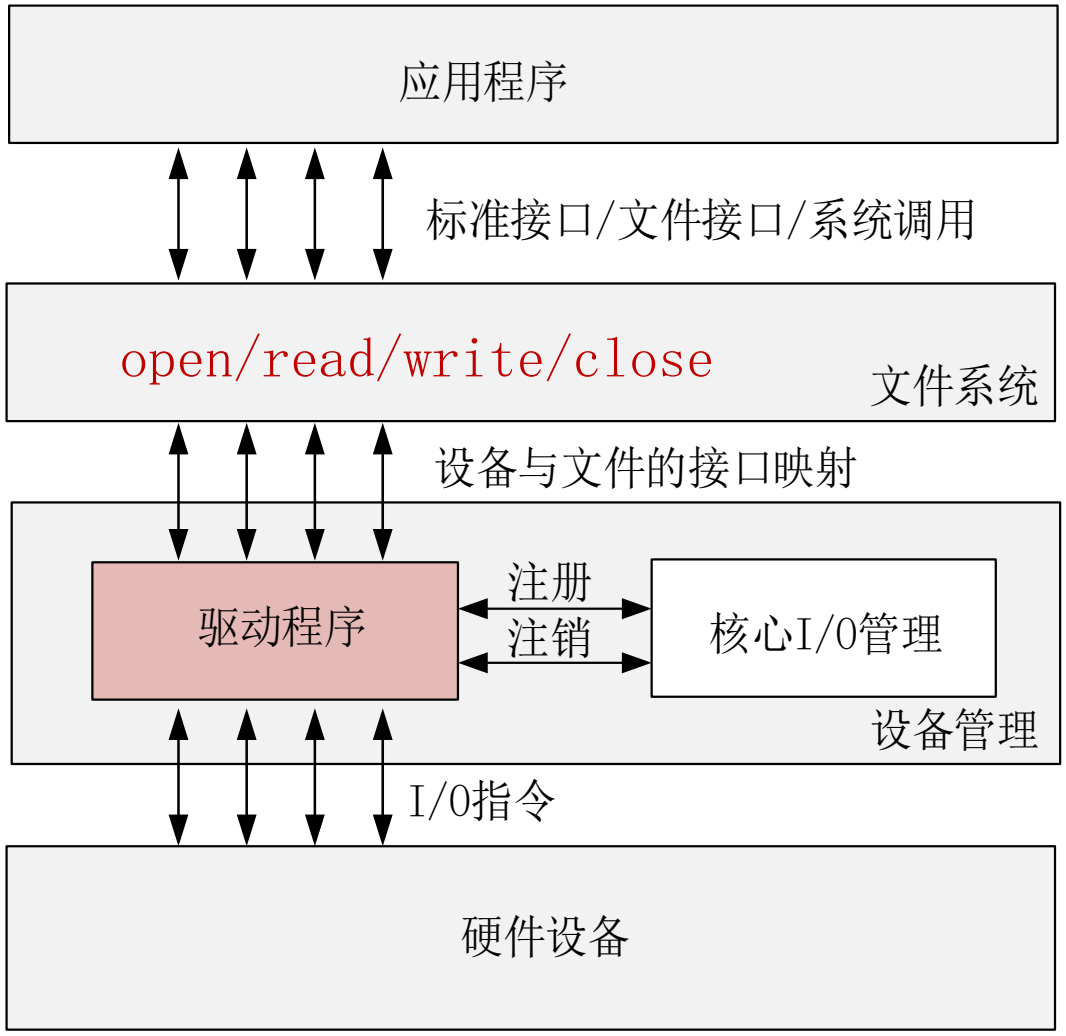
驱动程序在内核中的地位和接口

- 驱动程序的接口
  - 面向用户程序的接口
  - 面向I/O管理器的接口
  - 面向设备的接口



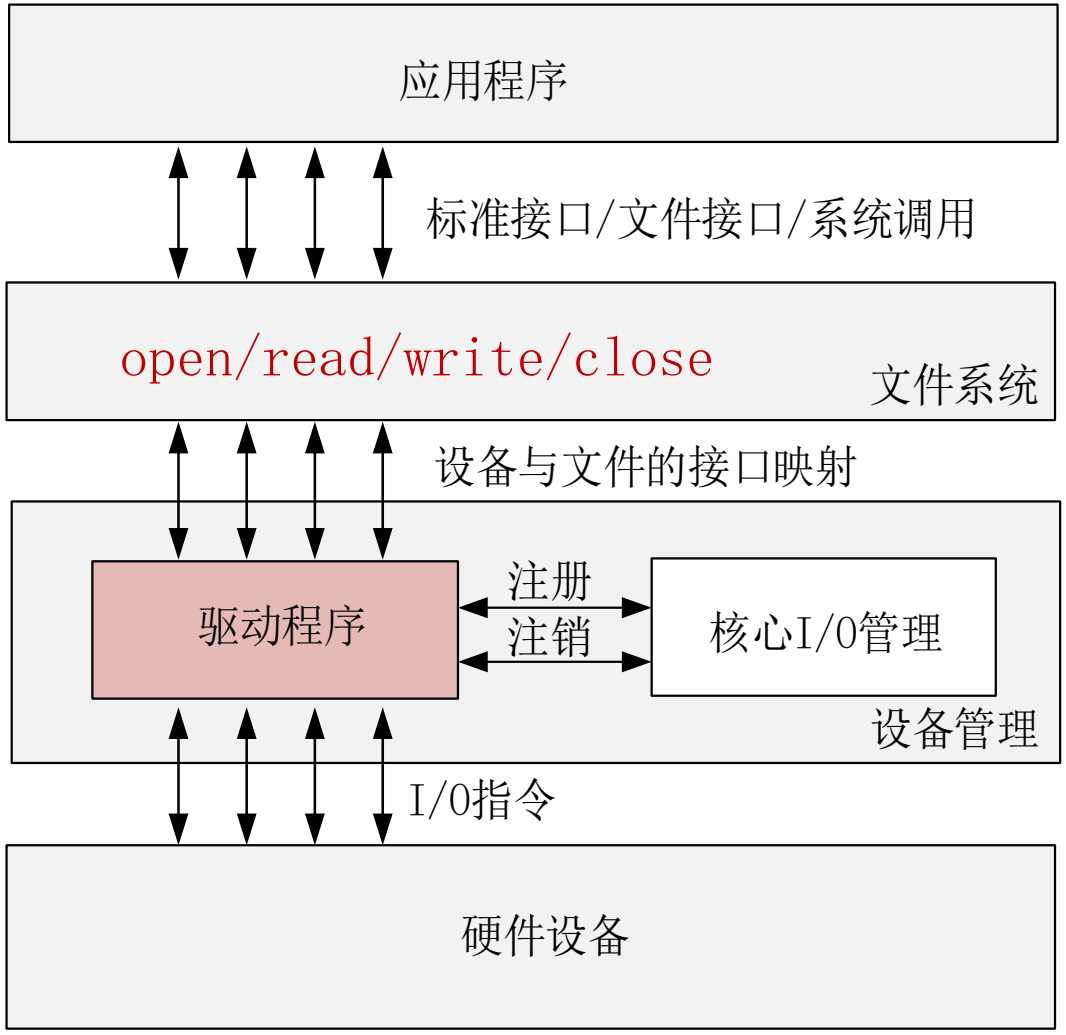
驱动程序在内核中的地位和接口

- 面向用户程序的接口
  - 设备的打开与释放
  - 设备的读写操作
  - 设备的控制操作
  - 设备的中断处理
  - 设备的轮询处理



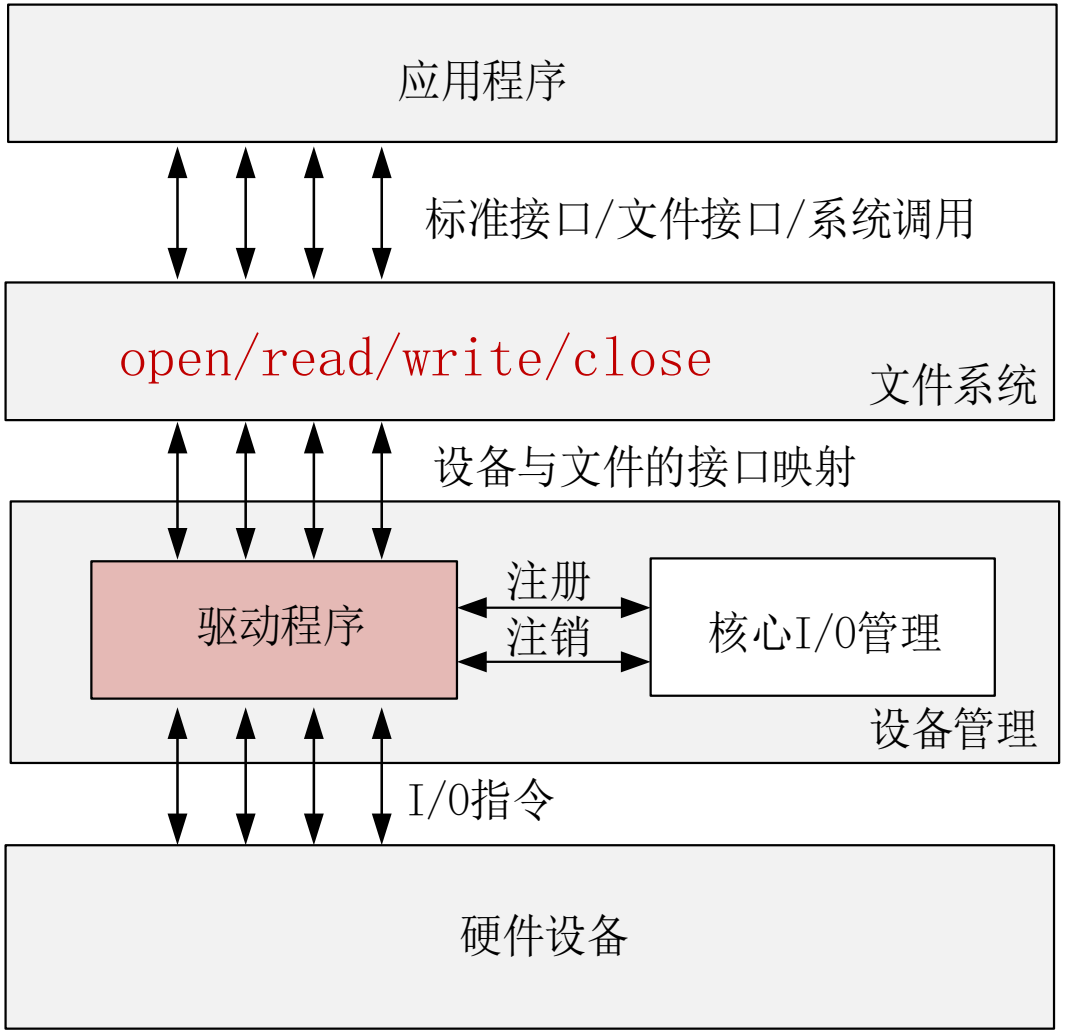
驱动程序在内核中的地位和接口

- 面向I/O管理器的接口
  - 注册函数
    - module_init(注册函数)
      - insmod(命令)
  - 注销函数
    - module_exit(注销函数)
      - rmmod (命令)
  - 必需的数据结构
    - file_operations
    - 设备数组



驱动程序在内核中的地位和接口

- 面向设备的接口
  - 实现设备的I/O操作
    - 无条件传送
    - 查询传送
    - 中断传送
    - DMA传送
  - 例：
    - 打开设备 chr_open( )
    - 读设备 chr_read( )
    - 写设备 chr_write( )
    - 关闭设备chr_release( )



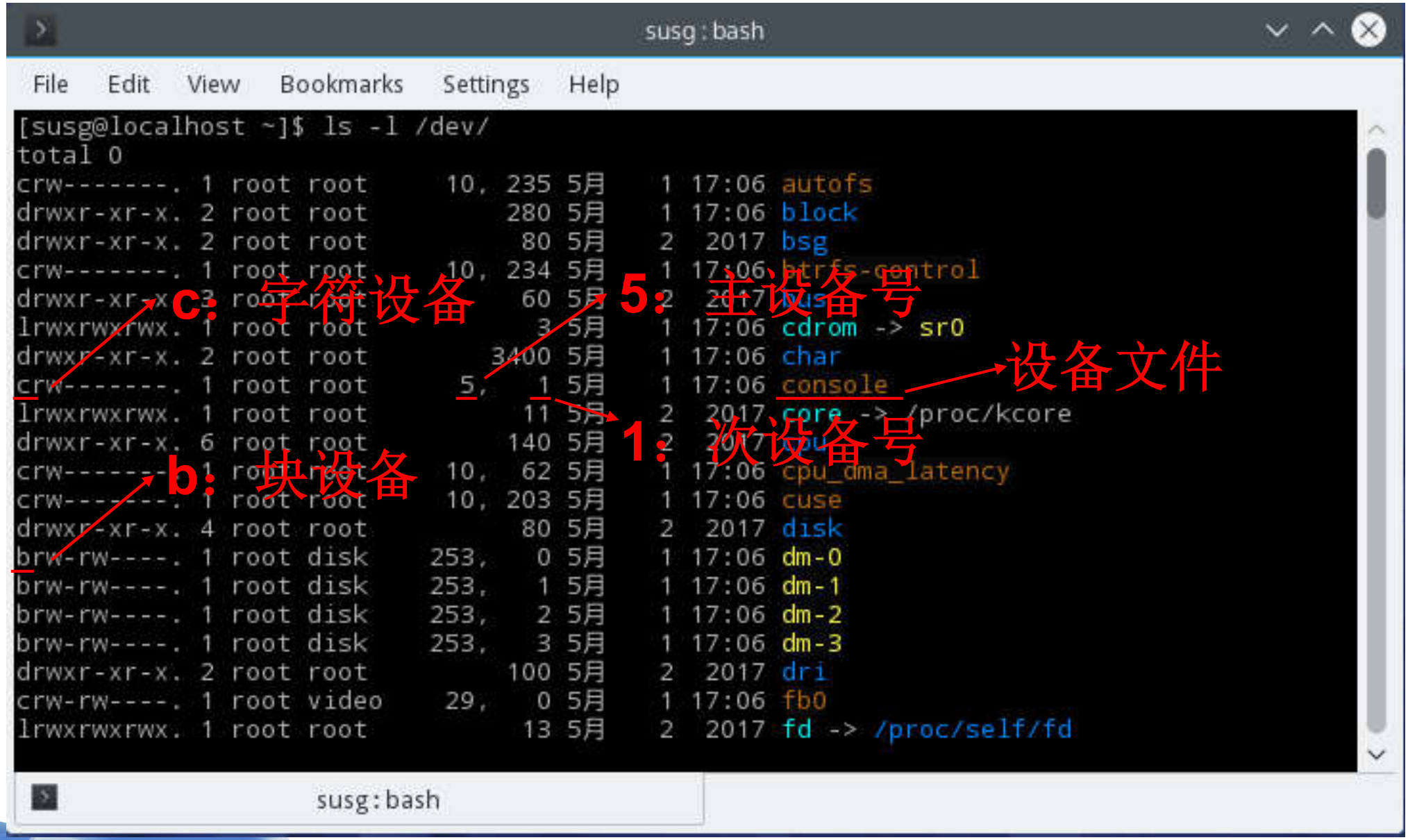
驱动程序在内核中的地位和接口

- 用户态与内核态
  - 驱动程序工作在内核态
  - 应用程序和驱动程序之间传送数据
    - get_user( )：用户态 → 内核态
    - put_user( )：内核态 →用户态
    - copy_from_user ( ) ：用户态 → 内核态
    - copy_to_user ( )：内核态 →用户态

“文件是设备的抽象”的理解(小结)

- 驱动程序和应用程序的开发过程
  - 应用程序使用文件操作接口完成设备的控制
  - 驱动程序实现文件操作接口
  - 设备(字符设备和块设备)创建有相应的设备文件
    - mknod命令

“文件是设备的抽象”的理解(小结)



“文件是设备的抽象”的理解(小结)

- 主设备号和次设备号
  - 主设备号
    - 标识该设备种类，标识驱动程序
    - 主设备号的范围：1-255
    - Linux内核支持动态分配主设备号
  - 次设备号
    - 标识同一设备驱动程序的不同硬件设备

Linux 2.6之后的内核

- 驱动程序注册过程的变化

```
//V2.4 字符设备注册
Led_Major = register_chrdev(0, DEVICE_NAME, &my_fops);
//V2.6字符设备注册
cdev_add(&cdev, MKDEV(major_No,0), DEV_NR);
```

- 预告：第11周实验
  - Linux字符设备驱动程序开发(Linux 5.5内核)

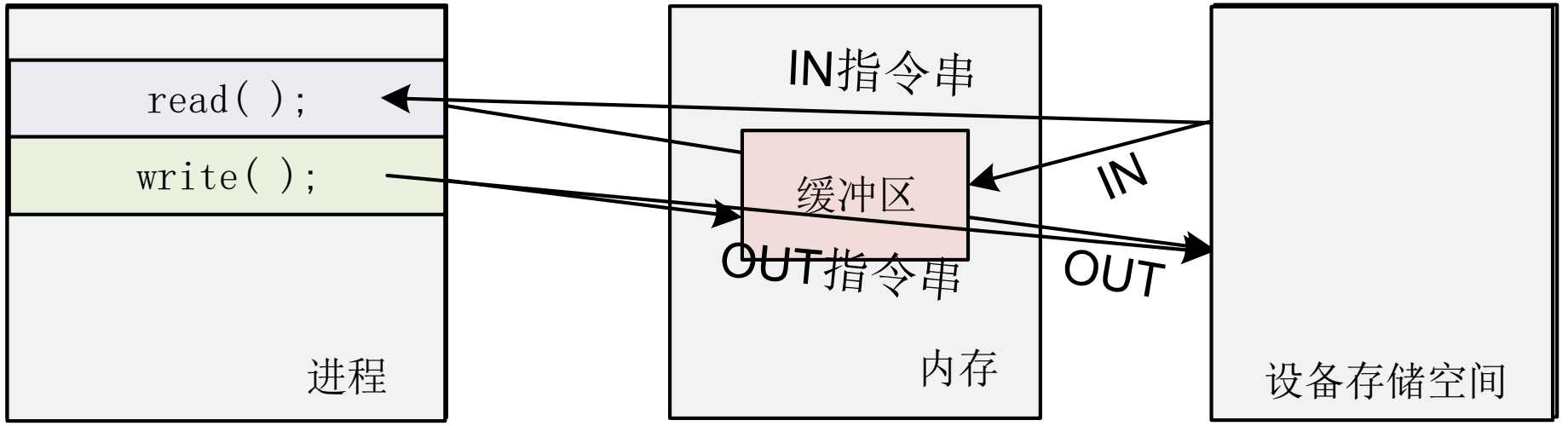
8.2 缓冲技术

缓冲技术

- 缓冲作用
  - 1）连接不同数据传输速度的设备
  - 2）协调数据记录大小的不一致
  - 3）正确执行应用程序的语义拷贝

缓冲作用

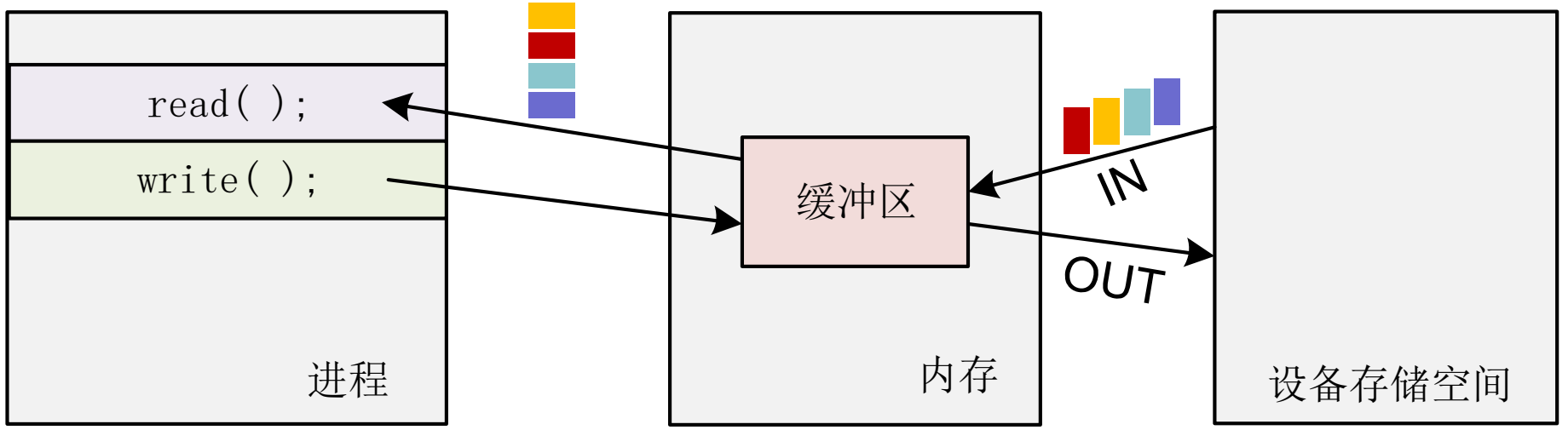
- 1）连接不同数据传输速度的设备
  - 例子：CPU（设备驱动）与设备（控制器）之间传输数据
  - 改进：内存中增加缓冲区



CPU（进程/驱动程序）　　外设（控制器）

缓冲作用

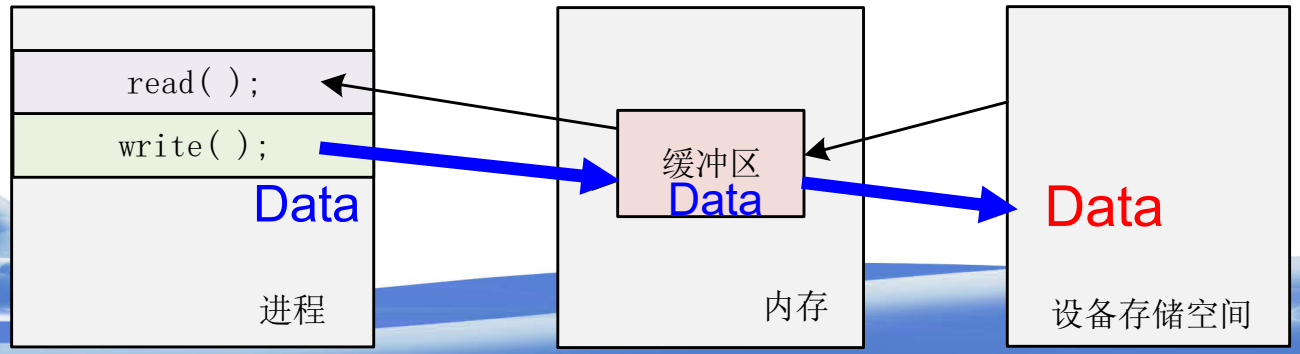
- 2）协调数据记录大小的不一致
  - 进程之间或CPU与设备之间的数据记录大小不一致
  - 例：进程（结构化数据）：设备（字节流/非结构）



CPU（进程/驱动程序）　　外设（控制器）

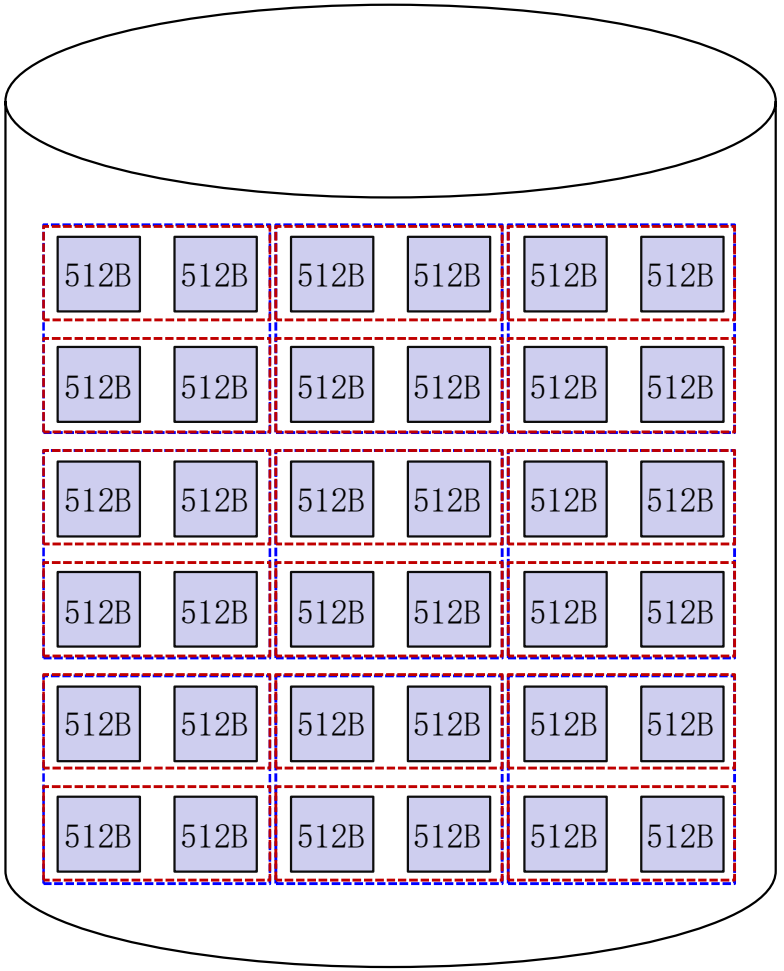
缓冲作用

- 3）正确执行应用程序的语义拷贝
  - 例子：利用write(Data, Len)向磁盘写入数据Data
    - 确保写入磁盘的Data是write调用时刻的Data
  - 方法：
    - 方法1
      - 应用等待内核写完磁盘再返回。（实时性差）
    - 方法2
      - 应用仅等内核写完内存即返回
        - 事后由内核把缓冲区写到磁盘。（实时性好）
      - 语义拷贝：确保事后拷贝的数据是正确版本



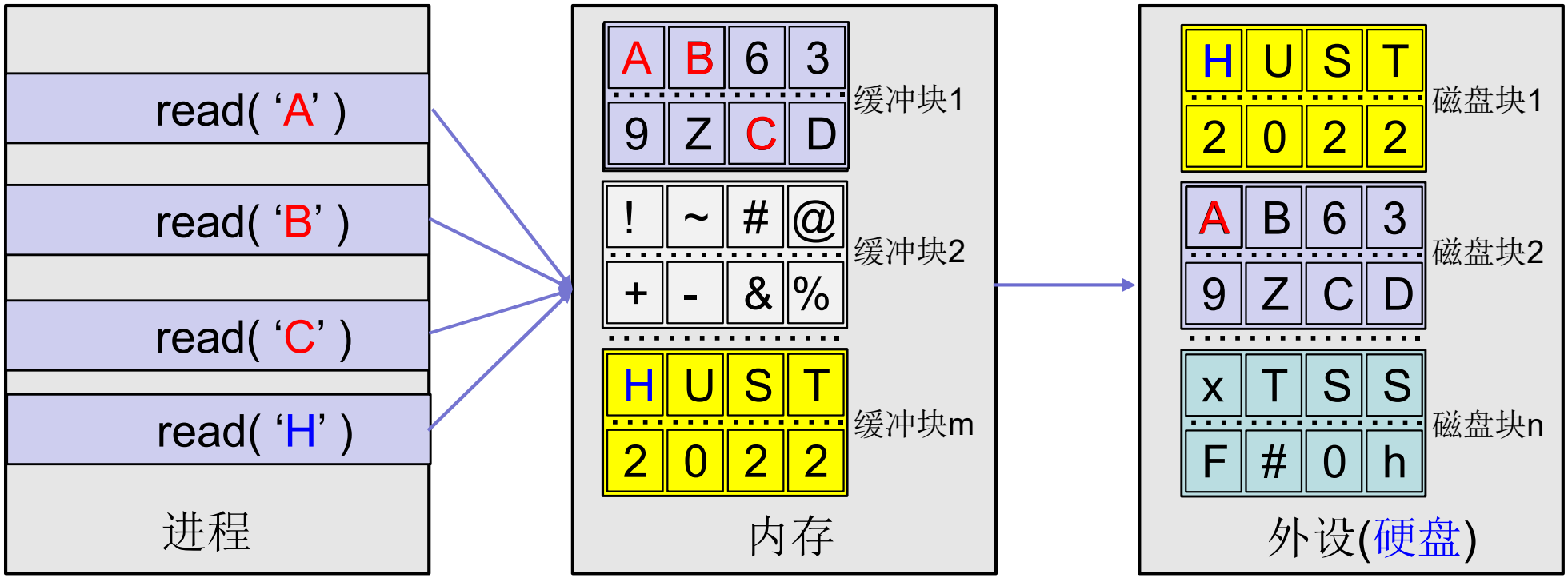
Linux缓冲机制应用（块设备）

- 典型的块设备
  - 硬盘、软盘、RAM DISK等
  - 块(block)和扇区
    - 硬盘读/写/寻址：扇区
    - 文件读/写/寻址：块
      - 块 = 2^n ×扇区
      - Linux块 = 1KB (n=1)
- Linux缓冲机制
  - 内存开辟高速缓冲区
  - 提前读/延后写



Linux缓冲机制应用（块设备）

- 提前读
  - 进程读时，其所需数据已被提前读到了缓冲区中，不需要启动外设去执行读操作。



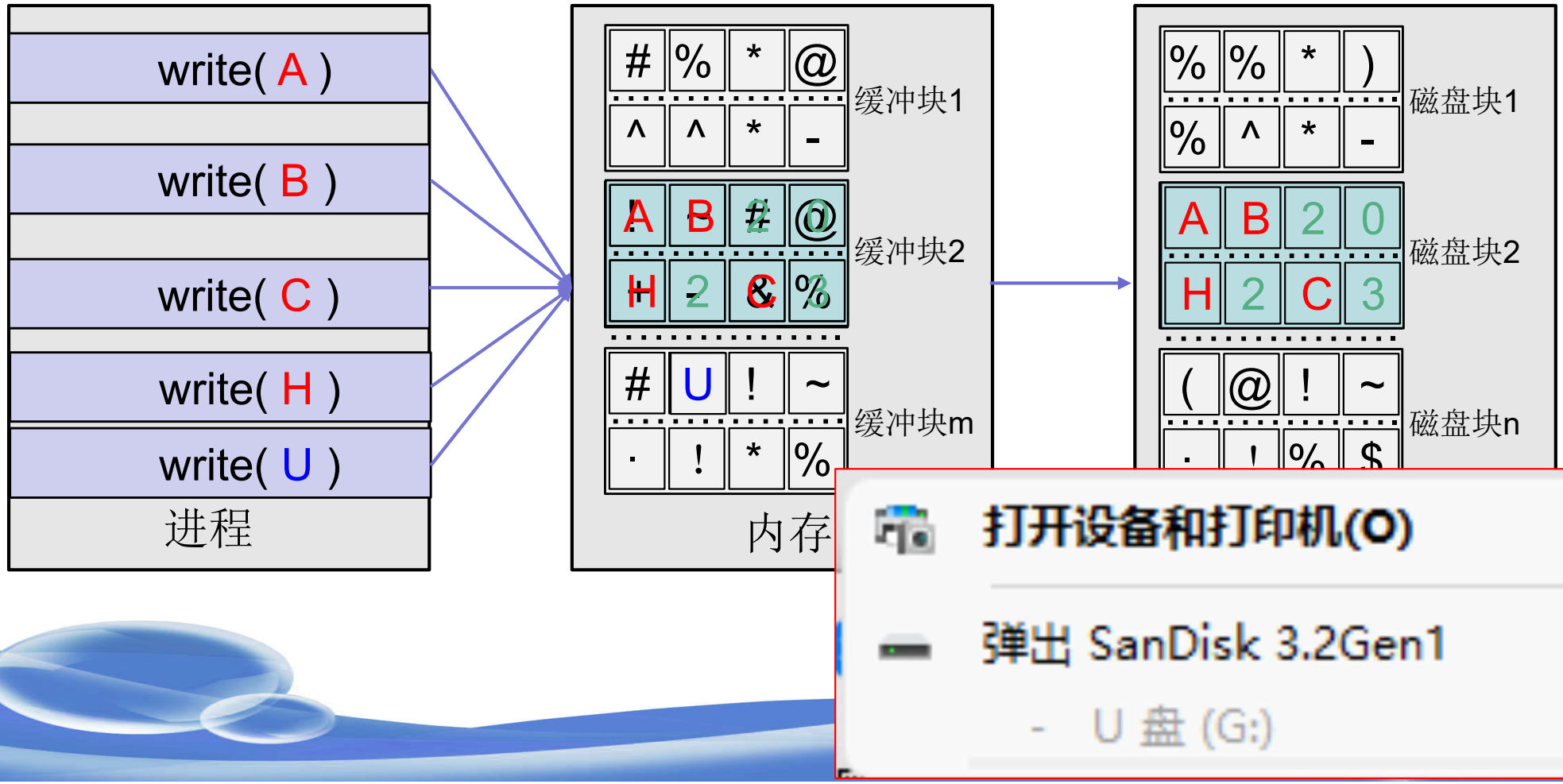
m << n

Linux缓冲机制应用（块设备）

- 延后写
  - 进程写时，数据先存在缓冲区，等到特定事件发生或足够时间后（已延迟），再启动外设完成写入。

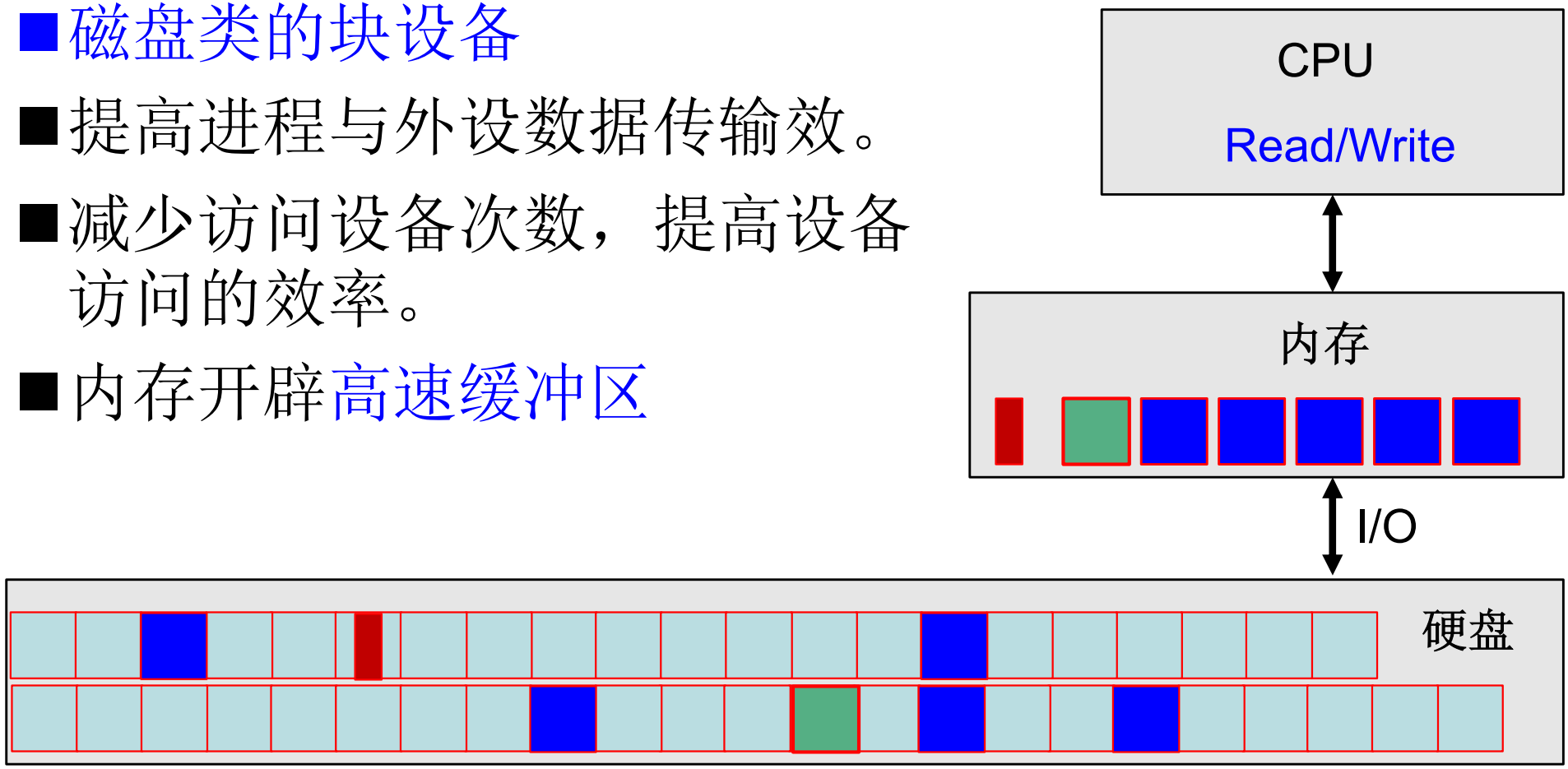
Linux缓冲机制应用（块设备）

- 延后写
  - 进程写时，数据先存在缓冲区，等到特定事件发生或足够时间后（已延迟），再启动外设完成写入。



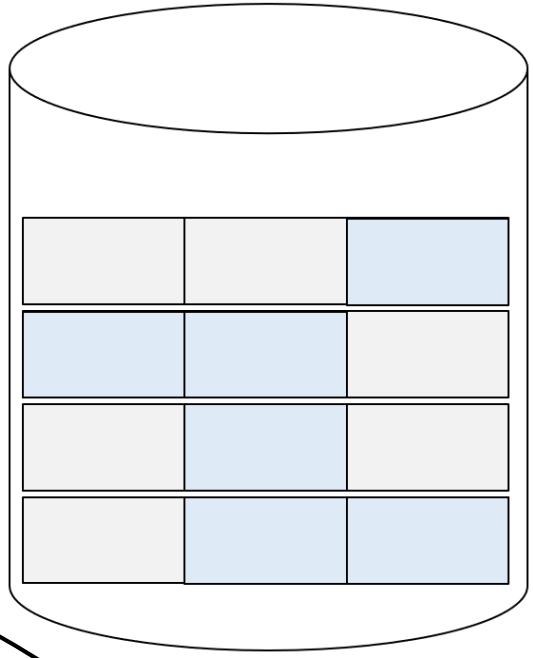
Linux缓冲机制应用（块设备）

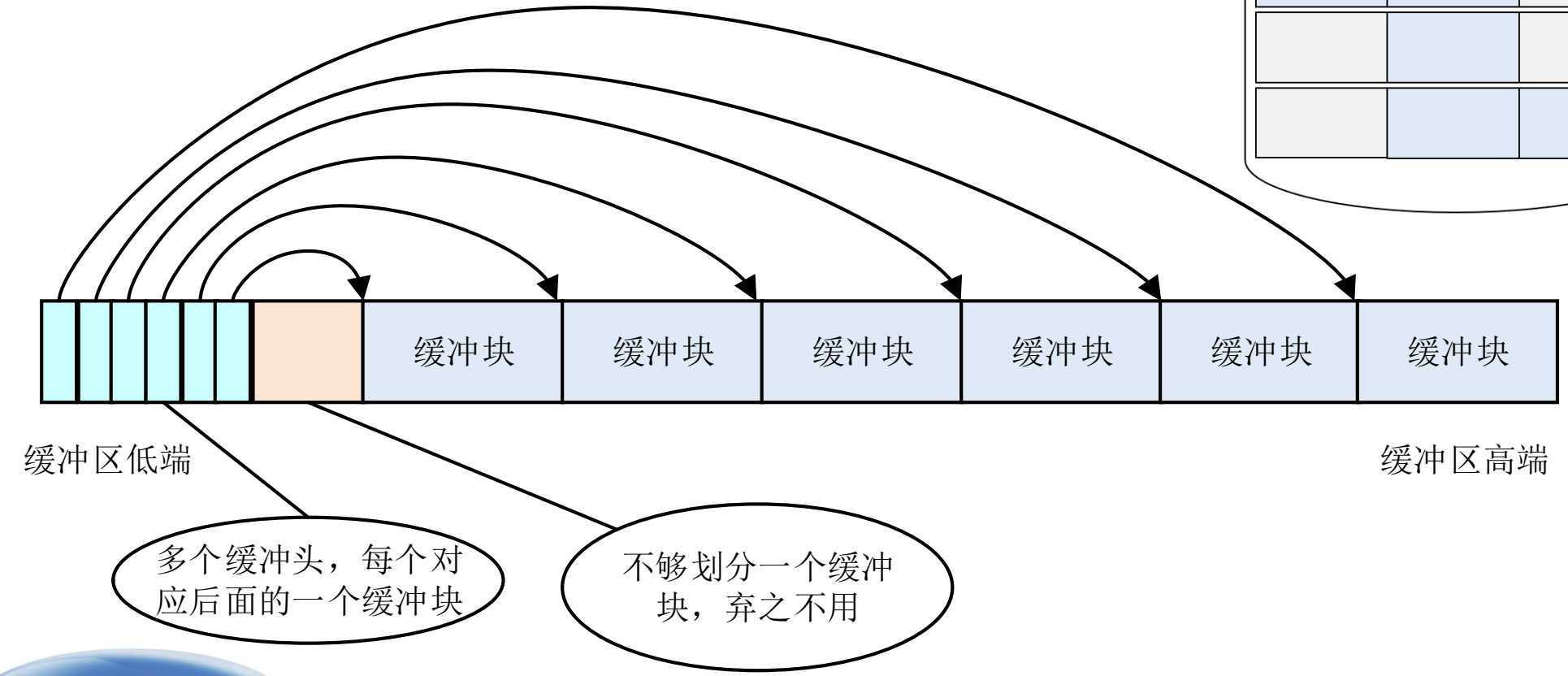
- 提前读与延后写
  - 磁盘类的块设备
  - 提高进程与外设数据传输效。
  - 减少访问设备次数，提高设备访问的效率。
  - 内存开辟高速缓冲区



Linux缓冲机制应用（块设备）

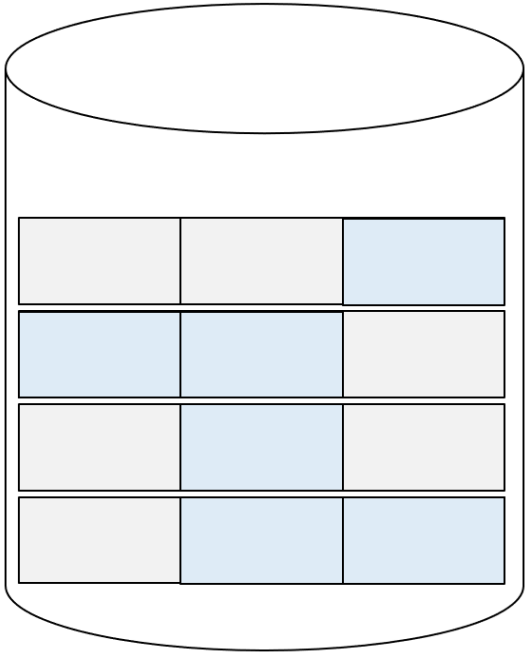
- 高速缓冲区(内存区)
  - 按块分为缓冲块(数据块)，与磁盘块对应
  - 缓冲头（buffer_head）：描述缓冲块





Linux缓冲机制应用（块设备）

- 高速缓冲区(内存区)
  - 按块分为缓冲块(数据块)，与磁盘块对应
  - 缓冲头（buffer_head）：描述缓冲块



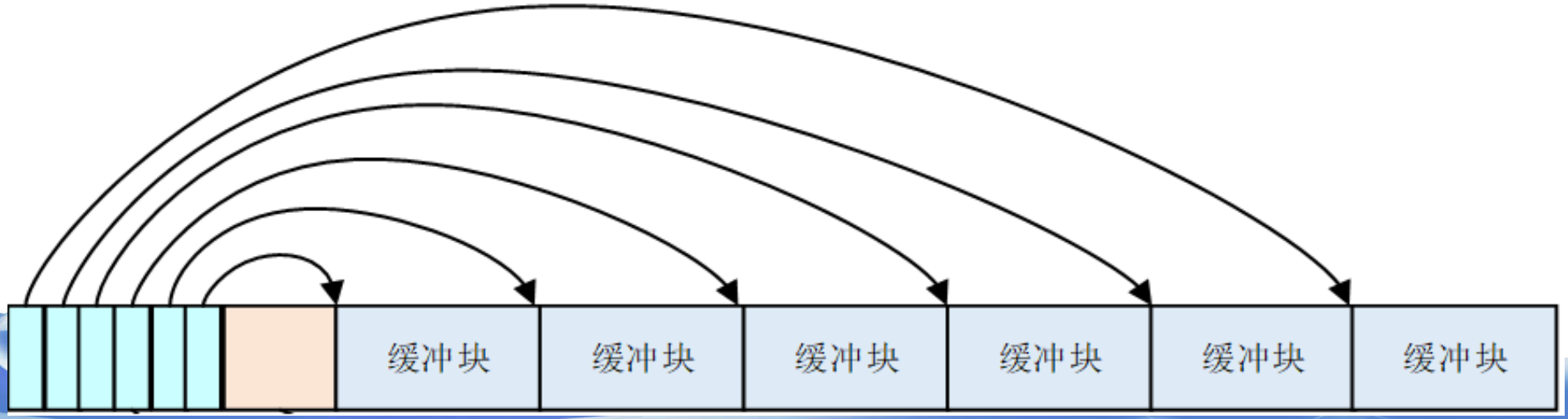
```
1  struct buffer_head {
2      char * b_data;              /* pointer to data bloc
3      unsigned long b_blocknr;    /* block number */
4      unsigned short b_dev;       /* device (0 = free) */
5      unsigned char b_uptodate;
6      unsigned char b_dirt;       /* 0-clean,1-dirty */
7      unsigned char b_count;      /* users using this block */
8      unsigned char b_lock;       /* 0 - ok, 1 -locked */
9      struct task_struct * b_wait;
10     struct buffer_head * b_prev;
11     struct buffer_head * b_next;
12     struct buffer_head * b_prev_free;
13     struct buffer_head * b_next_free;
14 };
```

Linux缓冲机制应用（块设备）

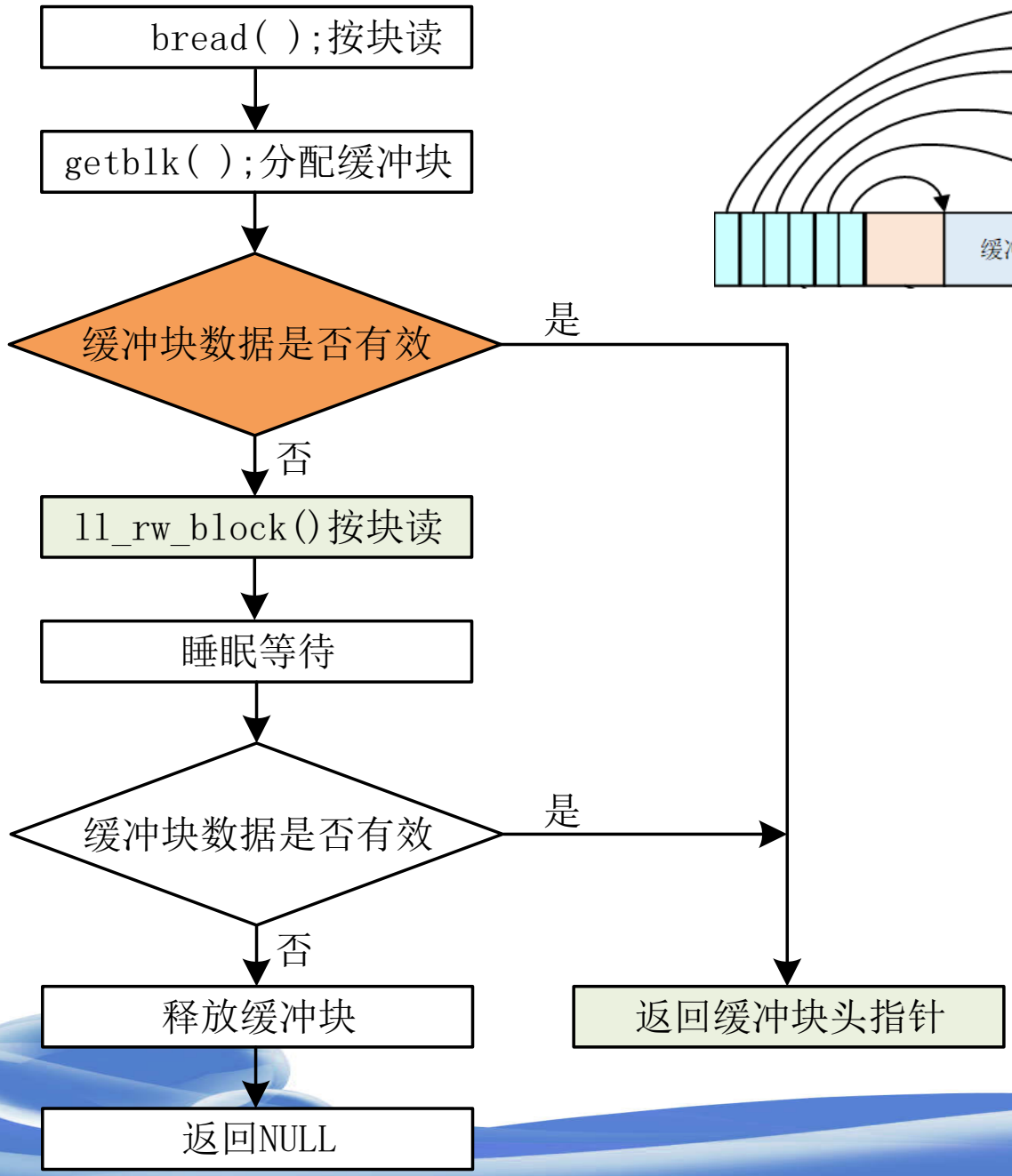
- 高速缓冲区(内存区)
  - 缓冲头buffer_head
    - b_data：指向数据区的指针
    - b_blocknr：对应的设备块的块号
    - b_dev：设备号
    - b_lock：缓冲块是否已被锁定
    - b_count：引用进程的数量
    - b_dirt：脏数据标记(延迟写标记)
    - b_uptodate：数据有效标记
    - b_wait：等待访问缓冲块的进程队列

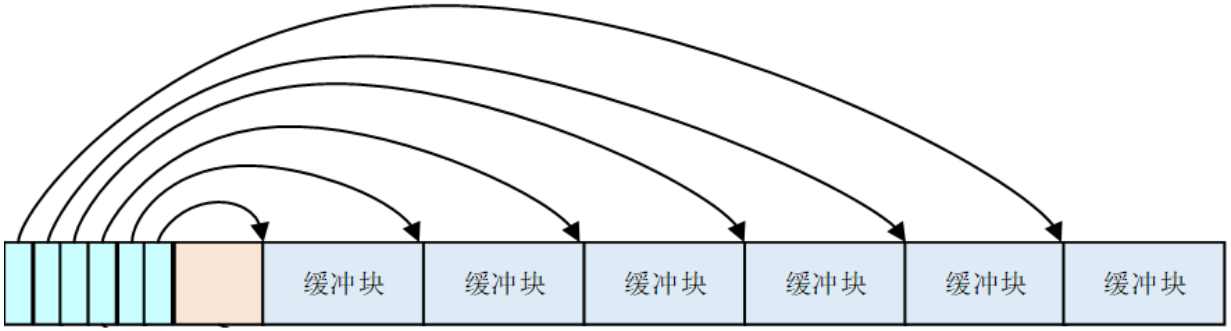
Linux缓冲机制应用（块设备）

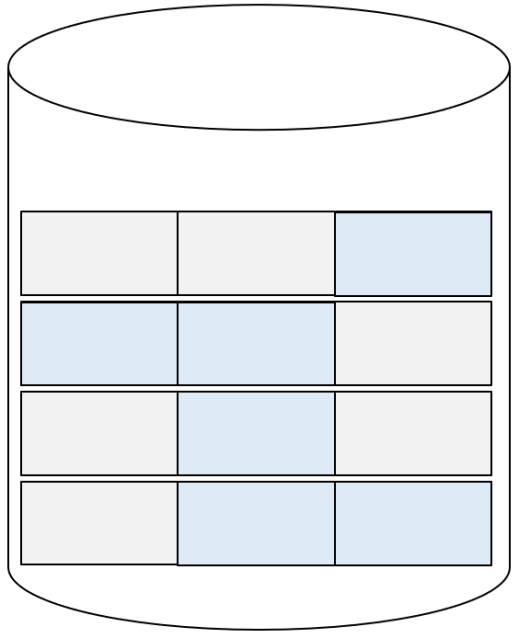
- 进程读设备数据
  - 进程read →文件访问请求 → 读取磁盘块(函数bread())
- 读取磁盘块**bread(设备号,块号)**
  - 以(设备号, 块号)为索引搜索高速缓冲区，查找对应的缓冲块
    - ◆若找到(即存在)，直接读回其中数据
    - ◆若没有找到(即还不存在)，分配一个新的缓冲块
      - ◘调用ll_rw_block把磁盘块(设备号, 块号)读入新的缓冲块



bread(设备号,块号)函数

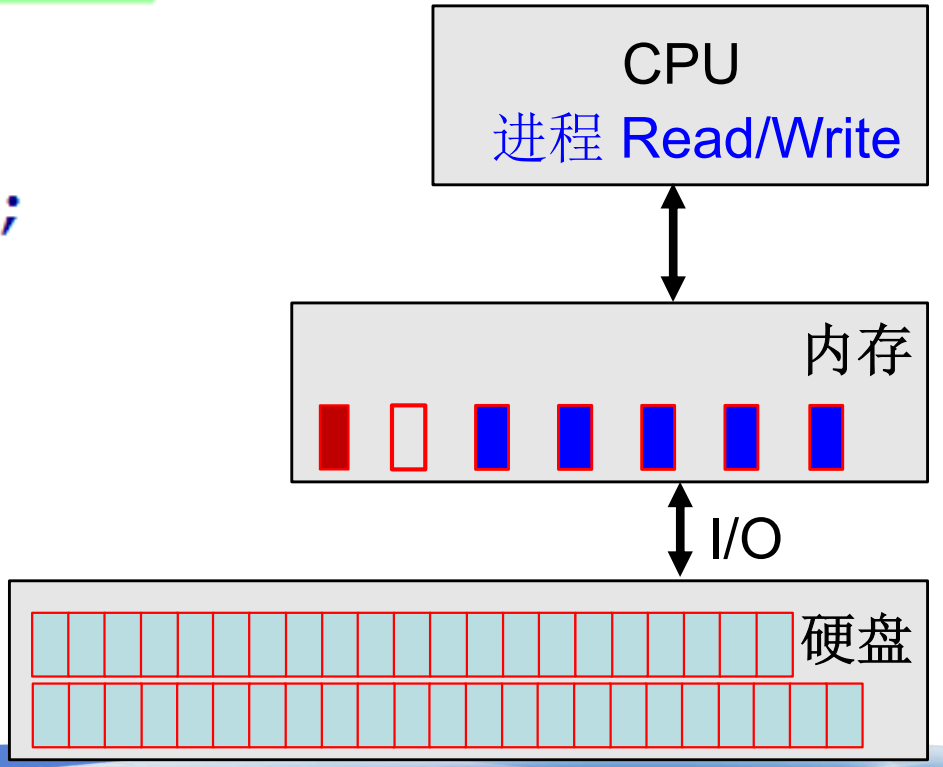






bread(设备号,块号)函数

```
1  struct buffer_head * bread(int dev,int block)
2  {
3      struct buffer_head * bh;
4
5      if (!(bh=getblk(dev,block)))
6          panic("bread: getblk returned NULL\n");
7      if (bh->b_uptodate)
8          return bh;
9      ll_rw_block(READ,bh);
10     wait_on_buffer(bh);
11     if (bh->b_uptodate)
12         return bh;
13     brelse(bh);
14     return NULL;
15 }
```



缓冲的组成

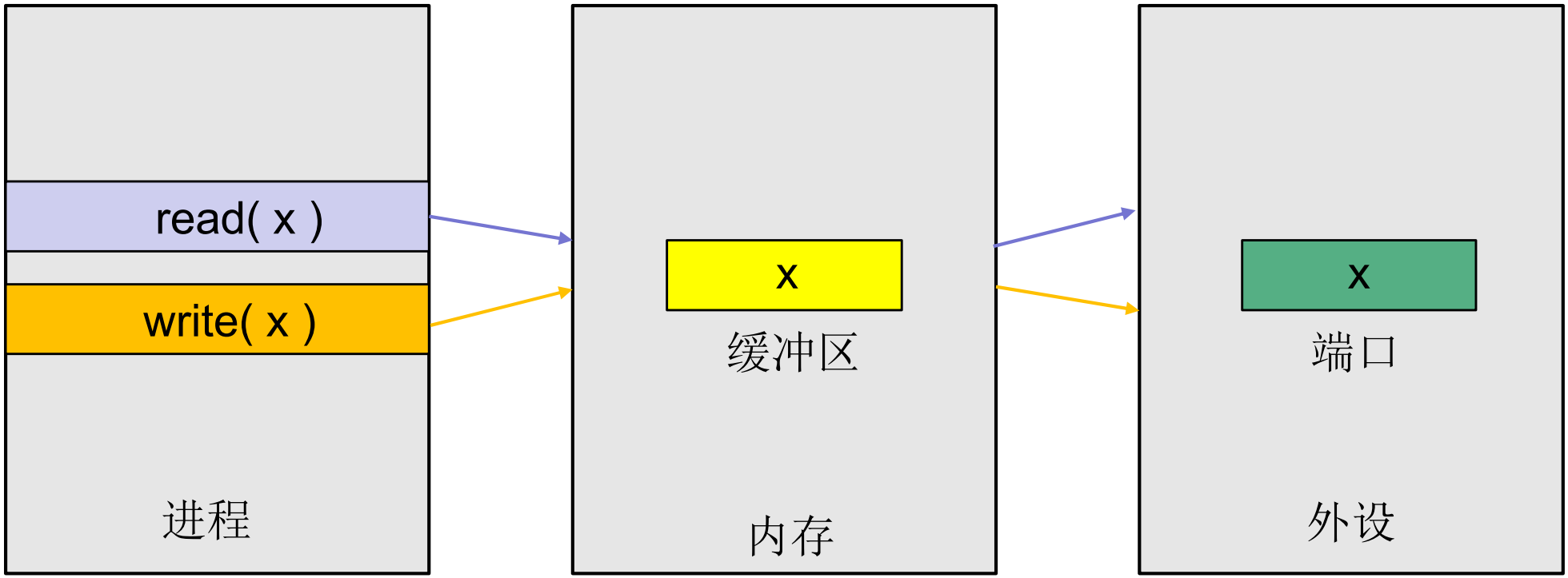
- 缓冲的组成形式
  - Cache
    - 高速缓冲寄存器【CPU ⇔ 内存】
  - 设备内部缓冲区
    - 外设或I/O接口的内部缓冲区【端口】
  - 内存缓冲区
    - 应用广泛，使用灵活【CPU ⇔ 接口/外设】
    - 应用开辟 | 内核开辟
  - 辅存缓冲区
    - 开辟在辅存上【 暂存内存数据，SWAP】

缓冲的实现

- 单缓冲
- 双缓冲
- 环形缓冲
- 缓冲池

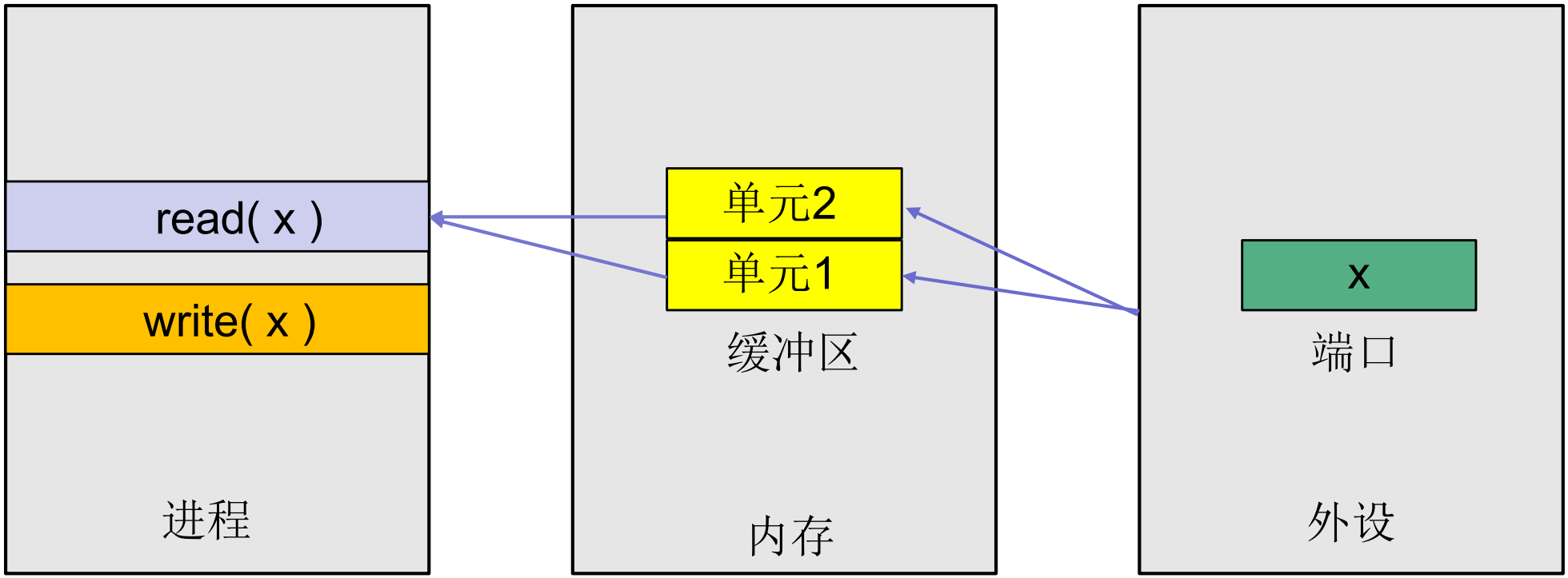
缓冲的实现>单缓冲

- 单缓冲
  - 缓冲区仅有1个单元



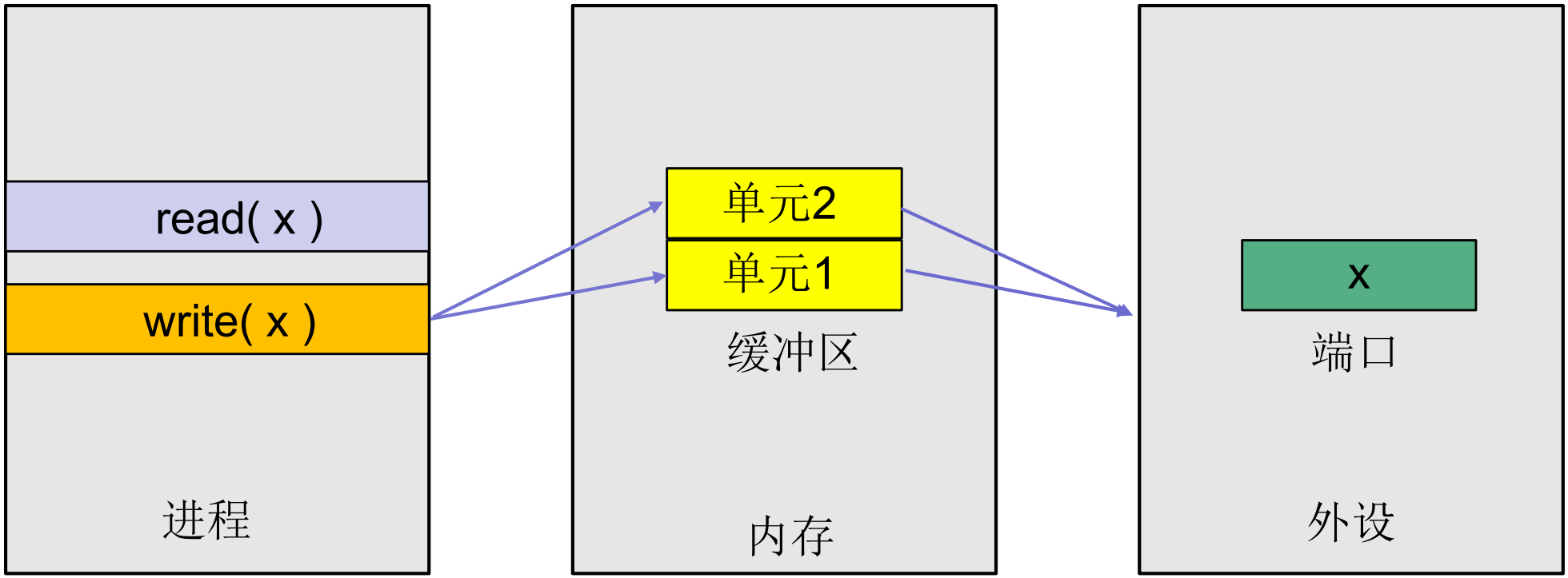
缓冲的实现>双缓冲

- 双缓冲
  - 缓冲区有2个单元



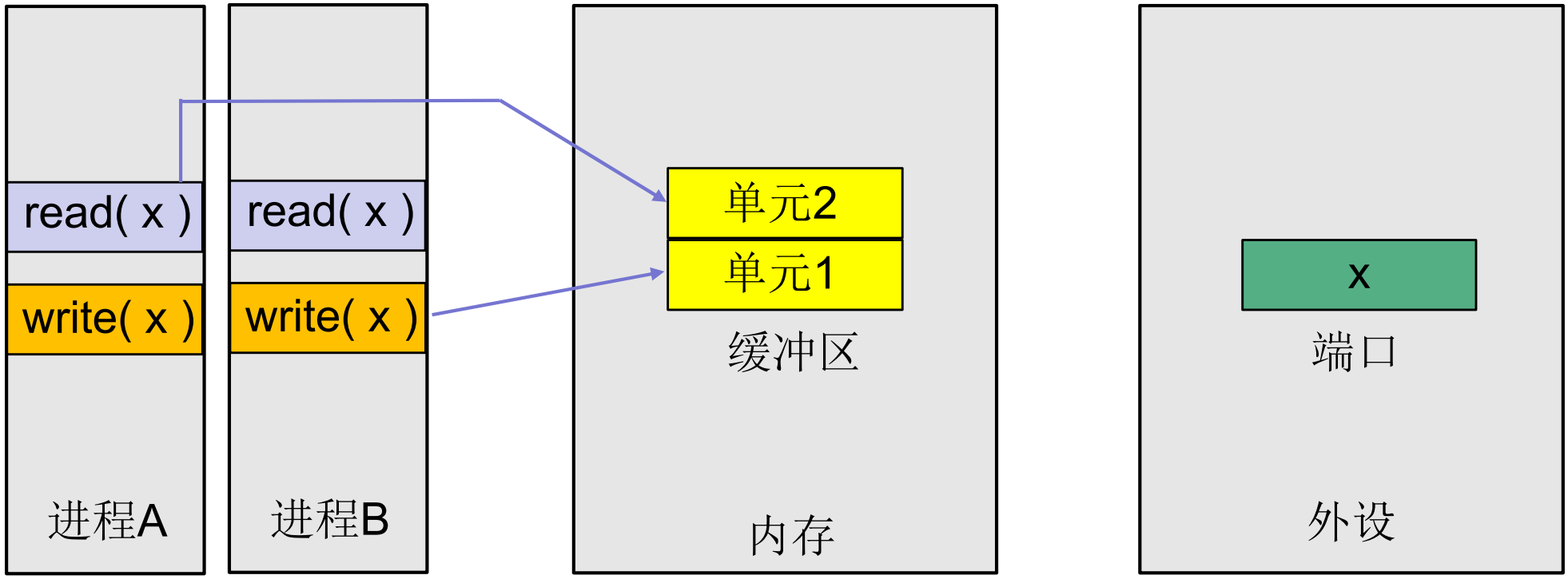
缓冲的实现>双缓冲

- 双缓冲
  - 缓冲区有2个单元



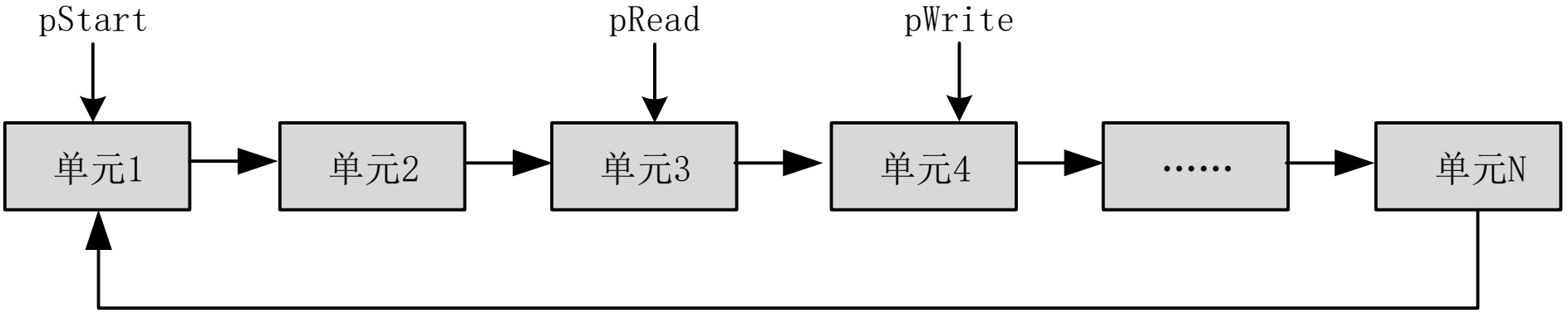
缓冲的实现>双缓冲

- 双缓冲
  - 缓冲区有2个单元



缓冲的实现>环形缓冲

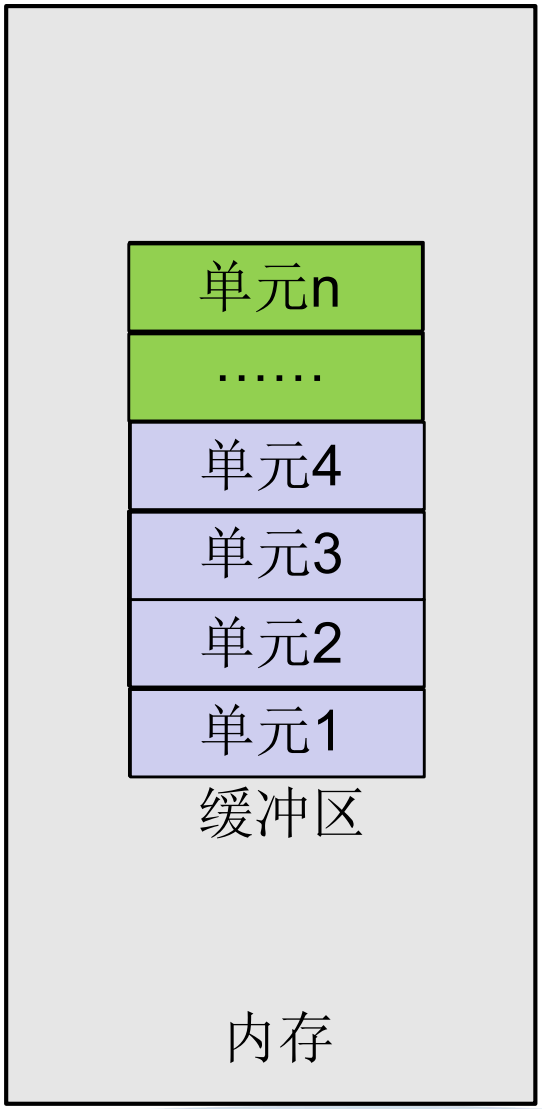
- 环形缓冲
  - 在双缓冲的基础上增加了更多的单元，并让首尾两个单元在逻辑上相连。



  - 起始指针pStart
  - 输入指针pWrite
  - 输出指针pRead

缓冲的实现>缓冲池

- 缓冲池
  - 多个缓冲区
  - 可供若干个进程共享
  - 可以支持输入，也可以支持输出
  - 提高缓冲区利用率，减少内存浪费



8.4 I/O控制（自学）

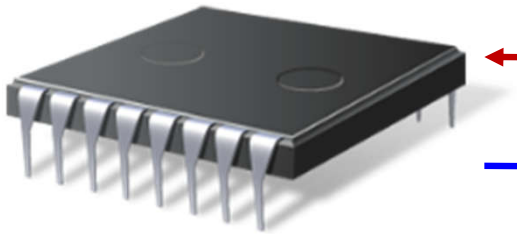
I/O控制

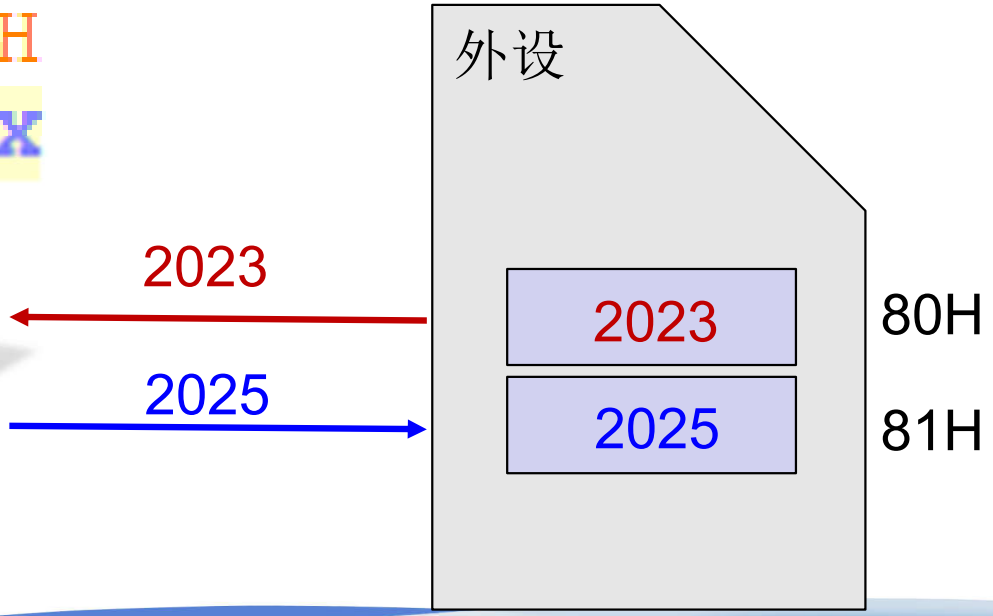
- **I/O数据控制方式**
  - 无条件传送方式（同步传送）
  - 查询方式（异步传送，循环测试I/O）
  - 中断方式
  - 通道方式
  - DMA方式

无条件传送（同步传送）

- 工作过程
  - 进行I/O时无需查询外设状态，直接进行。
  - 主要用于外设时序固定且已知的场合。
  - 当程序执行I/O指令【IN/OUT/MOV】时，外设必定已为传送数据做好了准备。

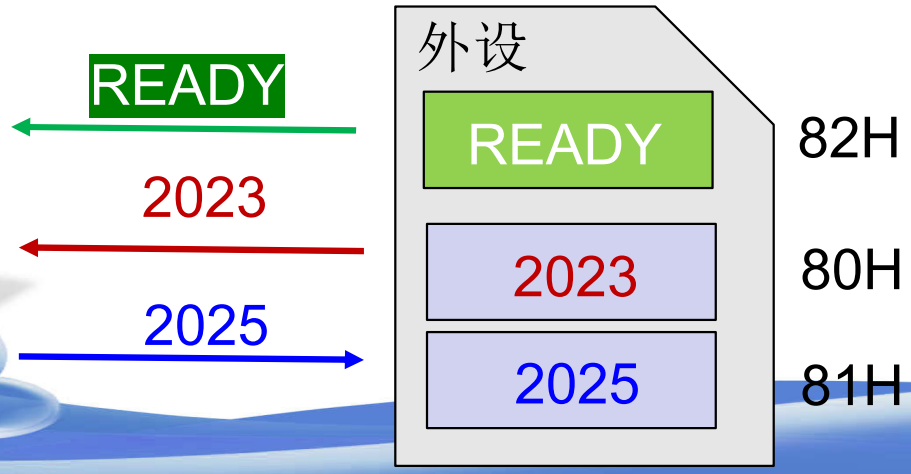
```
1    IN    AL,   80H
2    OUT   81H,   AX
```

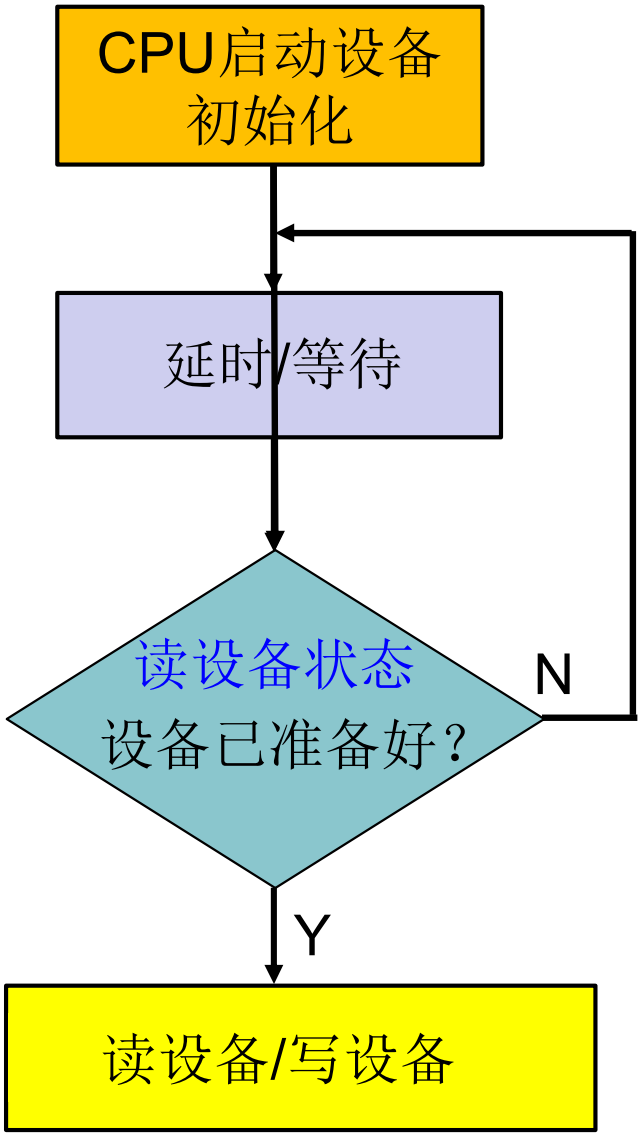




查询方式（异步传送）

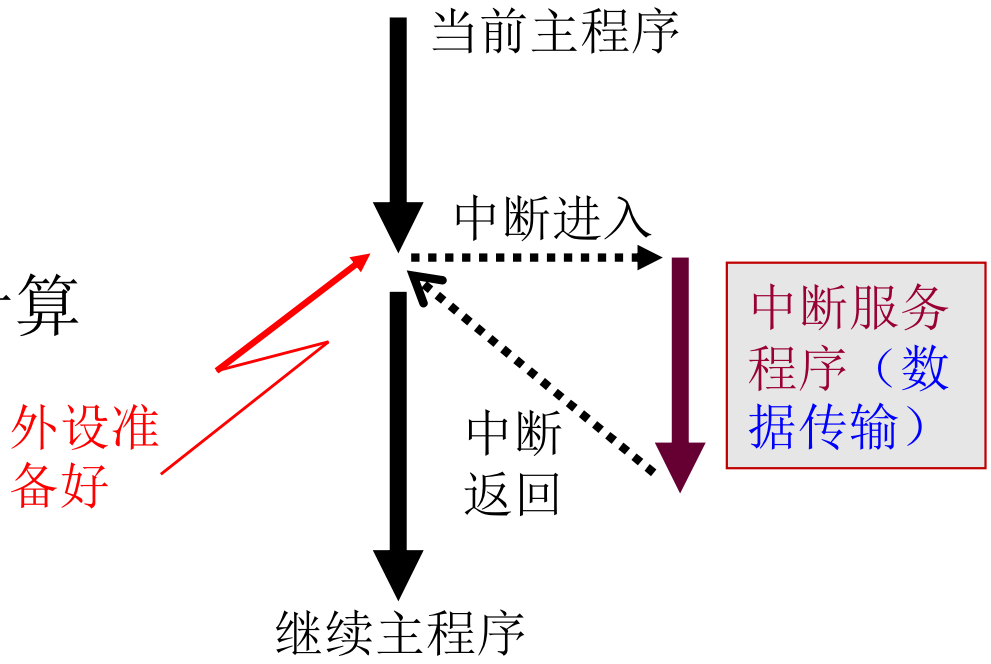
- 基本原理
  - 传送数据前，先检测外设状态，直到外设准备好才开始传送。
    - ◆输入时：外设数据“准备好”；
    - ◆输出时：外设 “准备好”接收。
- 特点
  - I/O操作由程序/ CPU发起并等待完成
    - ◆IN / OUT
  - 每次读写操作通过CPU





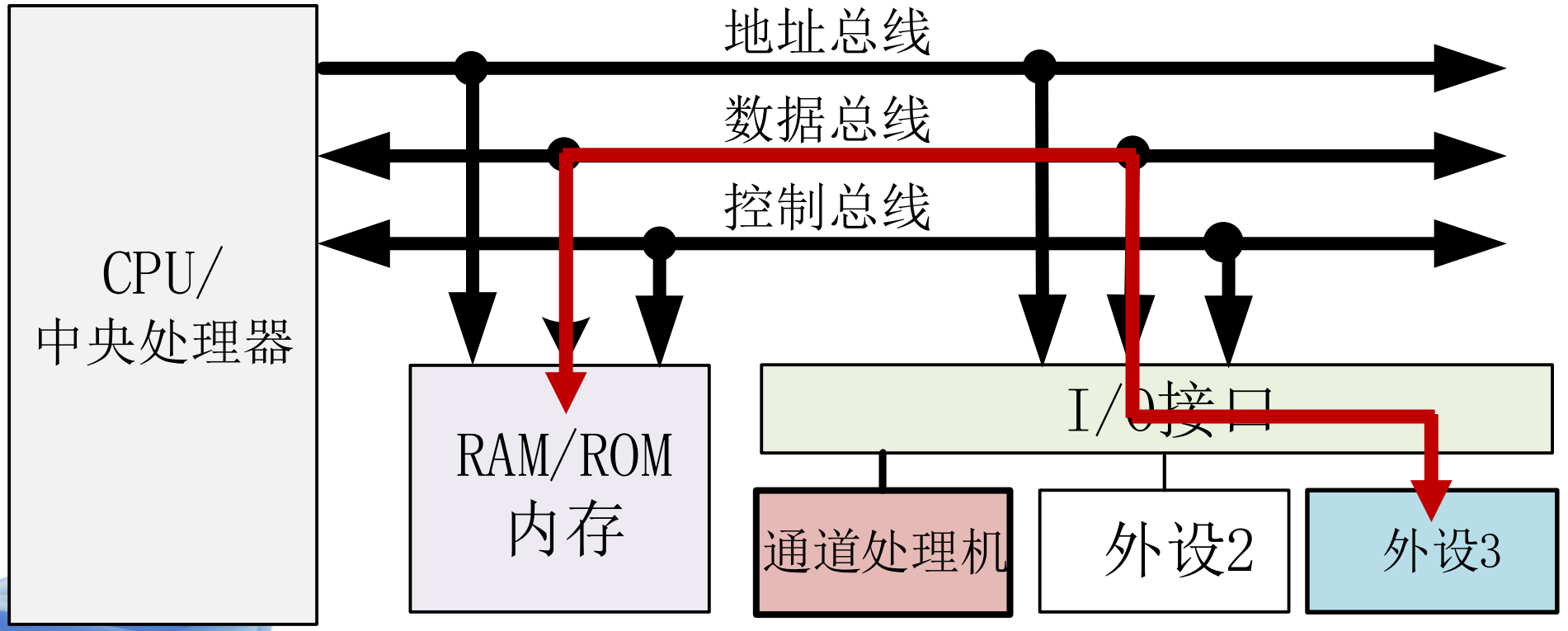
中断方式

- 工作原理
  - 外设数据准备好或准备好接收时，产生中断信号
  - CPU收到中断信号后，停止当前工作，执行数据传输。
  - CPU完成数据传输后继续原来工作。
- 特点
  - CPU和外设并行，CPU效率高
  - 中断服务程序中完成数据传送
- 缺点
  - 若设备频繁中断影响CPU有效计算
  - 数据吞吐率低
    - ◆仅适合少量数据低速传输。



通道方式

- 概念
  - 控制外设与内存之间数据传输的专门部件。
  - 有独立的指令系统（通道处理机，I/O处理机）
  - 既能受控于CPU，又能独立于CPU。

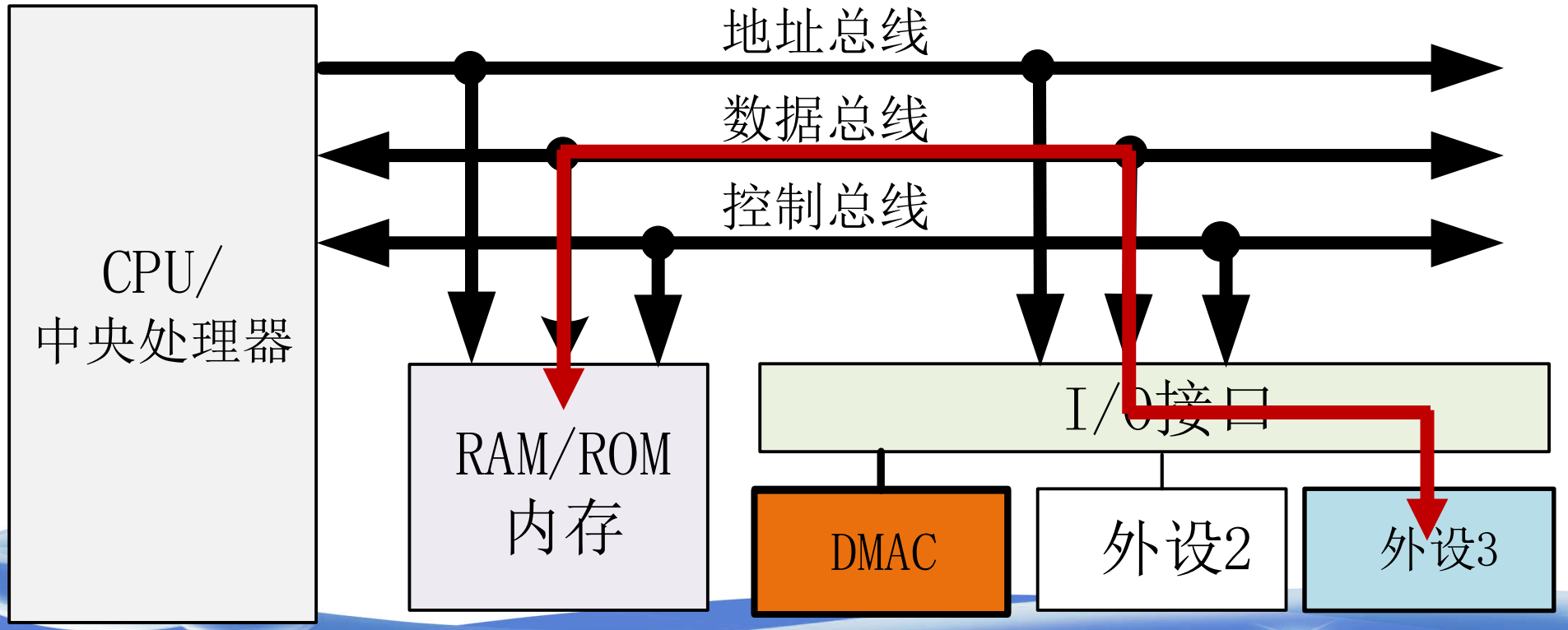


通道方式

- 概念
  - 控制外设与内存之间数据传输的专门部件。
  - 有独立的指令系统（ 通道处理机，I/O处理机）
  - 既能受控于CPU，又能独立于CPU。
- 特点
  - 传输过程无需CPU参与（除传输初始化和结束工作）
  - 以内存为中心，实现内存与外设直接数据交互。
  - 提高CPU与外设的并行程度

DMA(直接内存访问，Direct Memory Access)方式

- 概念
  - ■以内存为中心，实现内存与外设直接数据交互。
    - ◆仅传送初始化和传送结束工作需要CPU参与
  - ■DMA控制器/DMA Controller(DMAC)
  - ■微机广泛采用



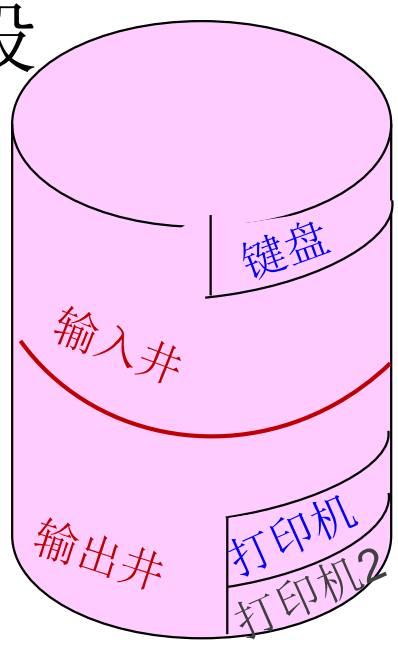
8.5 设备分配

设备分类

- 独占设备
  - 不可抢占设备（普通外设或资源）
    - 使用时独占，释放后才能被其它进程申请到。
    - 先申请，后使用（主动）
- 共享设备
  - 可抢占设备(CPU[分时]，内存[分区]，硬盘[分区])
    - 允许多个作业或进程同时使用。
    - 不申请，直接用（被动 + 主动）
- 虚拟设备
  - 借助虚拟技术，在共享设备上模拟的独占设备。

设备分类

- 虚拟设备
  - 借助虚拟技术，在共享设备上模拟的独占设备。具有独占设备的逻辑特点。
- 虚拟技术
  - 在一类物理设备上模拟另一类物理设备的技术
  - 借助辅存的部分存储区域模拟独占设备
    - 输入井：模拟输入设备的辅存区域
    - 输出井：模拟输出设备的辅存区域



设备分配方法

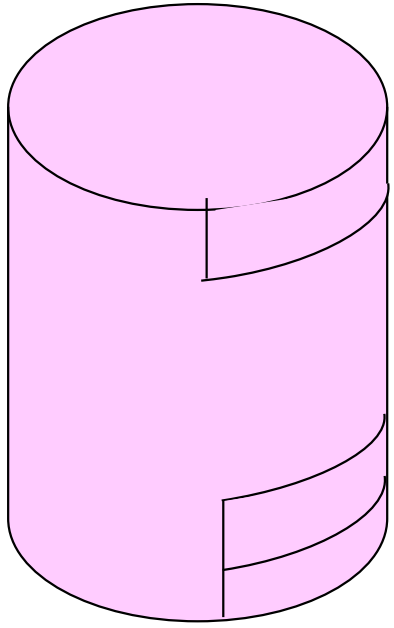
- 独享分配
- 共享分配
- 虚拟分配

设备分配方法

- 独享分配
  - 针对独占设备
  - 流程：申请→占用→释放
    - 指进程使用设备之前先申请，申请成功开始使用，直到使用完再释放。
  - 若设备已经被占用，则进程会被阻塞，被挂入设备对应的等待队列等待设备可用之时被唤醒。

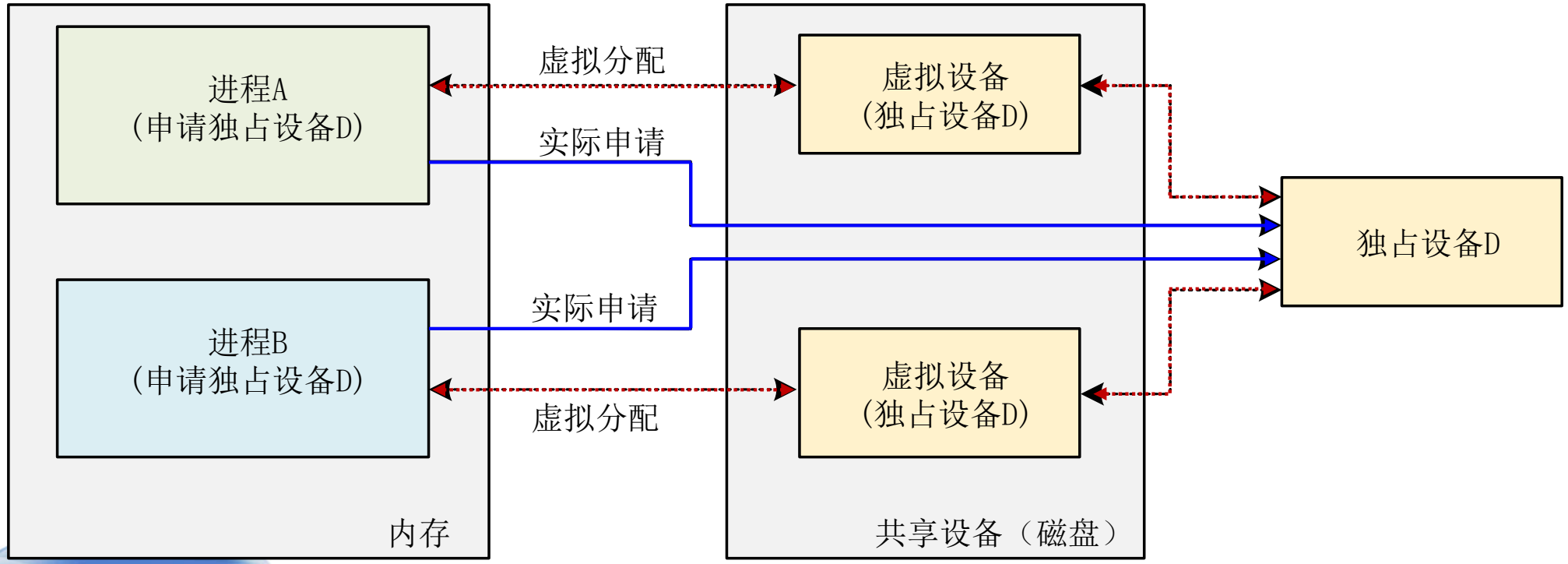
设备分配方法

- 共享分配
  - 针对共享设备
    - 典型共享设备：硬盘
  - 当进程申请使用共享设备时，操作系统能立即为其分配共享设备的一块空间（空分方式），不让进程产生阻塞。
  - 共享分配随时申请，随时可得。



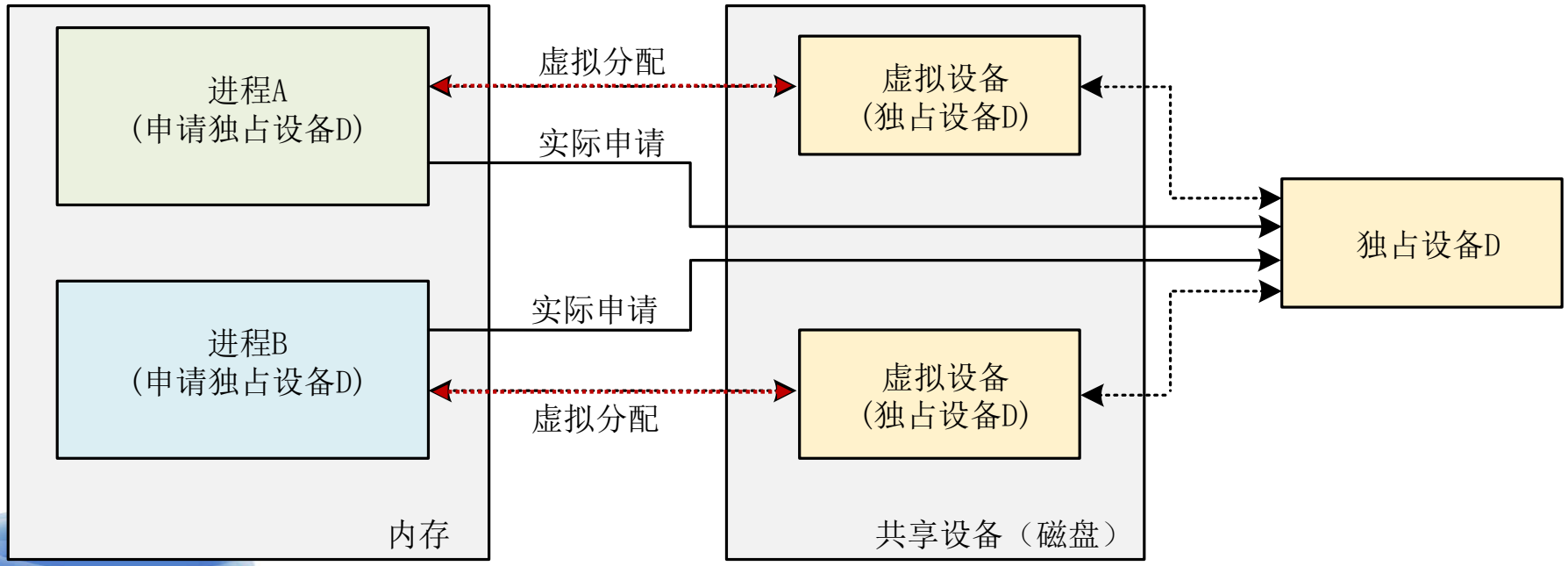
设备分配方法

- 虚拟分配
  - 当进程申请独占设备时将对应虚拟设备分配给它。
    - 首先，采用共享分配为进程分配虚拟设备；
    - 其次，将虚拟设备与对应的独占设备关联。



设备分配方法

- 虚拟分配
  - 当进程申请独占设备时将对应虚拟设备分配给它。
    - 首先，采用共享分配为进程分配虚拟设备；
    - 其次，将虚拟设备与对应的独占设备关联。
  - 进程运行中仅与虚拟设备交互，提高了运行效率

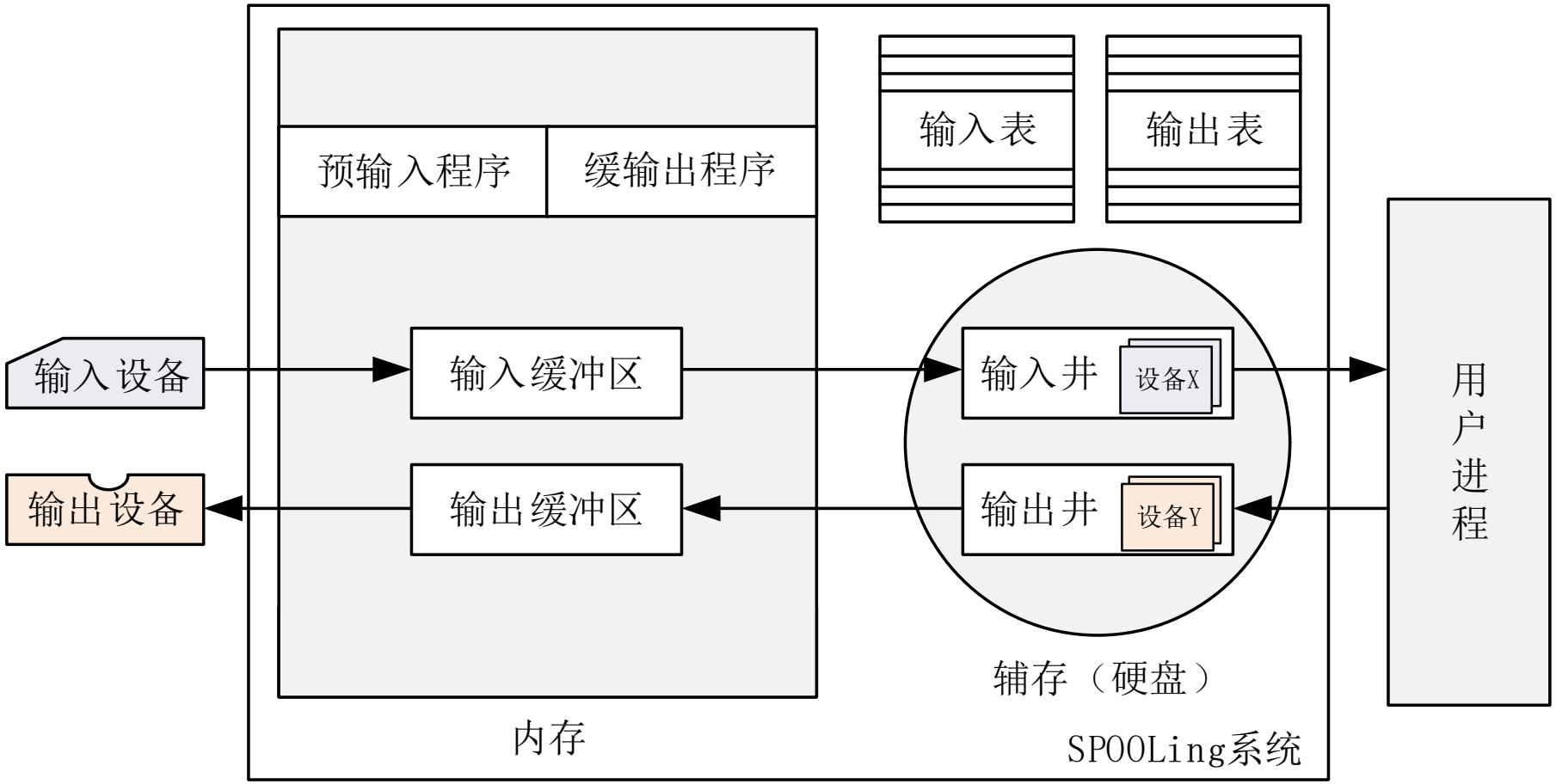


设备分配方法

- 虚拟分配
  - 当进程申请独占设备时将对应虚拟设备分配给它。
  - 例：SPOOLing系统
    - Simultaneaus Periphernal Operations OnLine
    - SPOOLing是虚拟技术和虚拟分配的实现
    - 外部设备同时联机操作 | 假脱机输入/输出

虚拟分配

- SPOOLing系统的结构



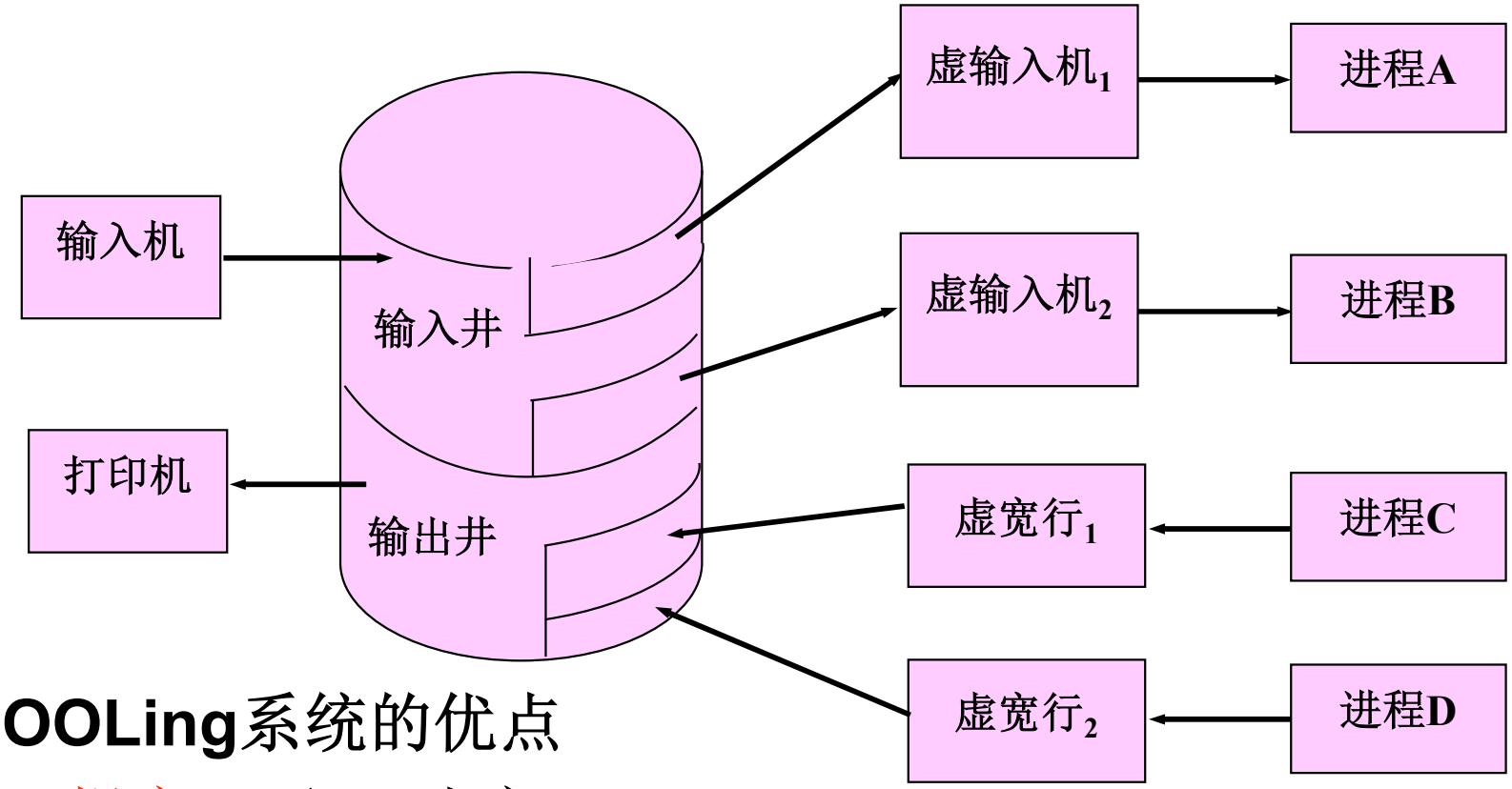
SPOOLing系统的结构（硬件）

- 输入井和输出井
  - 磁盘上开辟的两个存储区域
    - ◆输入井模拟脱机输入时的磁盘
    - ◆输出井模拟脱机输出时的磁盘
- 输入缓冲区和输出缓冲区
  - 内存中开辟的存储区域
    - ◆输入缓冲区：暂存输入数据，以后再传送到输入井。
    - ◆输出缓冲区：暂存输出数据，以后再传送到输出设备。

SPOOLing系统的结构（软件）

- 预输入程序
  - 控制信息从独占设备输入到辅存，模拟脱机输入的卫星机；
- 输入表
  - 独占设备⇔虚拟设备
- 缓输出程序
  - 控制信息从辅存输出到独占设备，模拟脱机输出的卫星机；
- 输出表
  - 独占设备⇔虚拟设备
- 井管理程序
  - 控制用户程序和辅存之间的信息交换

SPOOLing的例子



- SPOOLing系统的优点
  - “提高”了I/O速度
  - 将独占设备改造为“共享”设备
    - 实现了虚拟设备功能

8.5.3 Windows设备驱动

最简单的windows驱动——Hello world

```
01. #include <ntddk.h>
02.
03. NTSTATUS helloUnload(IN PDRIVER_OBJECT DriverObject) {
04.
05.     DbgPrint("good bye!\n");
06.     return STATUS_SUCCESS;
07. }
08.
09. NTSTATUS DriverEntry(
10.                     IN PDRIVER_OBJECT DriverObject,
11.                     IN PUNICODE_STRING RegistryPath) {
12.
13.     DbgPrint("hello world!\n");
14.    pDriverObject->DriverUnload = helloUnload;
15.     return STATUS_SUCCESS;
16. }
```

8.6. 设备阻塞工作模式

设备的阻塞工作模式和非阻塞工作模式

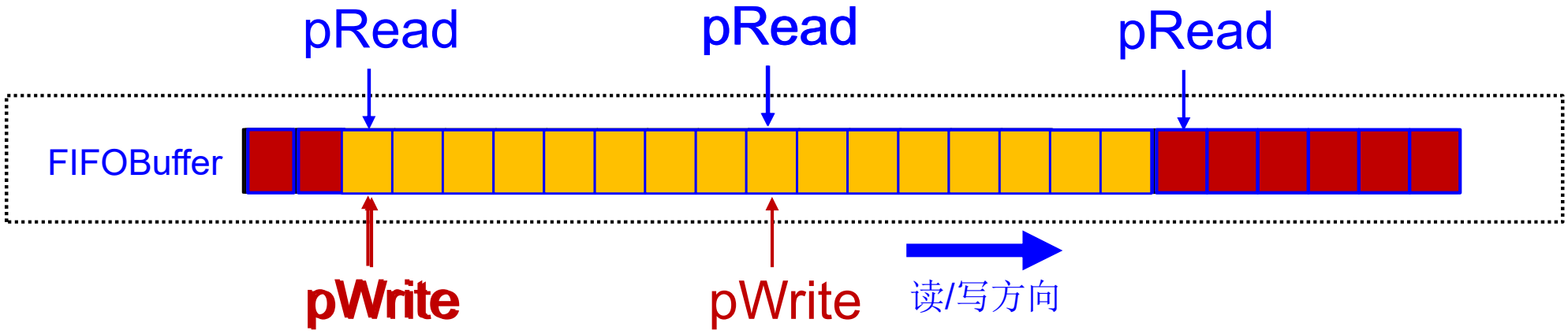
- 阻塞工作模式
  - ■若不能提供服务，则挂起应用进程直到条件满足
    - ◆应用进程进入休眠状态（不占用CPU ）
      - □转移到设备的等待队列。
- 非阻塞工作模式
  - ■若不能提供服务，返回应用进程错误代码，不挂起
    - ◆应用进程
      - □措施1：做错误处理（如结束正常工作）
      - □措施2：重新申请服务。

应用的阻塞模式和非阻塞模式

- 设备不能服务时，应用进程是否愿被阻塞？
  - 愿被阻塞
    - open(fd,BLOCK);
    - 一般在中断中被唤醒
  - 不愿被阻塞
    - open(fd, NON_BLOCK);
    - 愿意接收错误码
      - 应用可尝试重复申请服务、结束或其他处理。

支持阻塞工作模式的设备驱动程序的示例

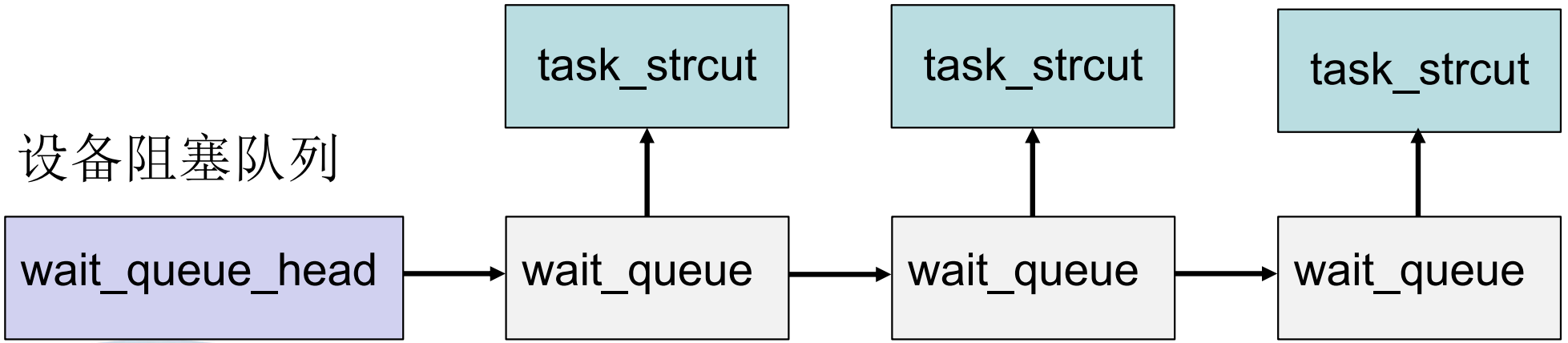
- 支持阻塞工作模式的设备驱动程序的示例



  - FIFOBuffer：环形缓冲区
  - 若干应用程序随机读写FIFOBuffer缓冲区。
    - 读写同步要求：不重复，不遗漏
  - 读队列：请求读的进程
  - 写队列：请求写的进程

设备的阻塞队列

- 设备的阻塞进程队列
  - 队列：wait_queue_head
    - 元素：wait_queue
  - 服务可用时被唤醒
    - 进程设为就绪并从阻塞队列中删除
    - 疯狂群兽



支持阻塞工作模式的设备驱动程序的示例

```
1  #define BUFFER_SIZE 32
2  DEFINE_KFIFO(FIFOBuffer, char, BUFFER_SIZE);
3  struct _BlockDevice
4  {
8      wait_queue_head_t ReadQueue;
9      wait_queue_head_t WriteQueue;
10 };
11 static ssize_t
12 DevRead(struct file *file, char *buf, size_t count, loff_t *ppos)
13 {
16     int actual_readed;
19     if (kfifo_is_empty(&FIFOBuffer))
20     {
21         if (file->f_flags & O_NONBLOCK)
22             return -EAGAIN;
25         ret = wait_event_interruptible(BlockDevice->ReadQueue,
26                                        !kfifo_is_empty(&FIFOBuffer));
27         if (ret)
28             return ret;
29     }
31     ret = kfifo_to_user(&FIFOBuffer, buf, count, &actual_readed);
35     if (!kfifo_is_full(&FIFOBuffer))
36         wake_up_interruptible(&BlockDevice->WriteQueue);
40     return actual_readed;
41 }
```