《操作系统原理》

20级考试复习版本，以本课件为准

第8章 设备管理(输入/输出管理)

教师：苏曙光

华中科技大学

2022年03月-2022年05月

- 内容
  - 设备管理概述
  - 缓冲技术
  - 设备分配
  - I/O设备控制
  - 设备驱动程序
- 重点
  - 理解缓冲的作用
  - 理解SPOOLING技术
  - 掌握设备驱动程序的开发过程

8.1 设备管理概念

设备类型和特征

- **1. 按交互对象分类**
  - 人机交互设备：显示设备、键盘、鼠标、打印机
  - 与CPU等交互的设备：磁盘、磁带、传感器、控制器
  - 计算机间的通信设备：网卡、调制解调器
- **2. 按交互方向分类**
  - 输入设备：键盘、扫描仪
  - 输出设备：显示设备、打印机
  - 双向设备：输入/输出：硬盘、软盘、网卡
- **3. 按外设特性分类**
  - 使用特征：存储、输入/输出
  - 数据传输率：低速(键盘)、中速(打印机)、高速(网卡、磁盘)
  - 信息组织特征：字符设备(如打印机), 块设备(如磁盘),网络设备

设备类型和特征

- **4. 按信息组织特征分类**
  - ■字符设备
    - ◆传输的基本单位是字符，像键盘、串口。
  - ■块设备
    - ◆块是设备存储和传输的基本单位。
  - ■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。

设备管理功能

- 设备管理的目标
  - 提高设备的利用率
  - 提高设备读写效率
  - 提高CPU与设备并行程度
  - 为用户提供统一接口
  - 实现设备对用户透明

设备管理功能

- 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
  - 设备控制块（Device Control Block，DCB)
    - 记录设备的基本属性、状态、操作接口以及进程与设备之间的交互信息等

设备管理功能>设备分配

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

设备管理功能>设备映射

- 设备逻辑名/友好名Friendly Name
  - 用户编程时使用的名字
  - 例：Windwos：\\.\MyDevice

```
hDevice = CreateFile("\\\\.\\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>设备映射

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

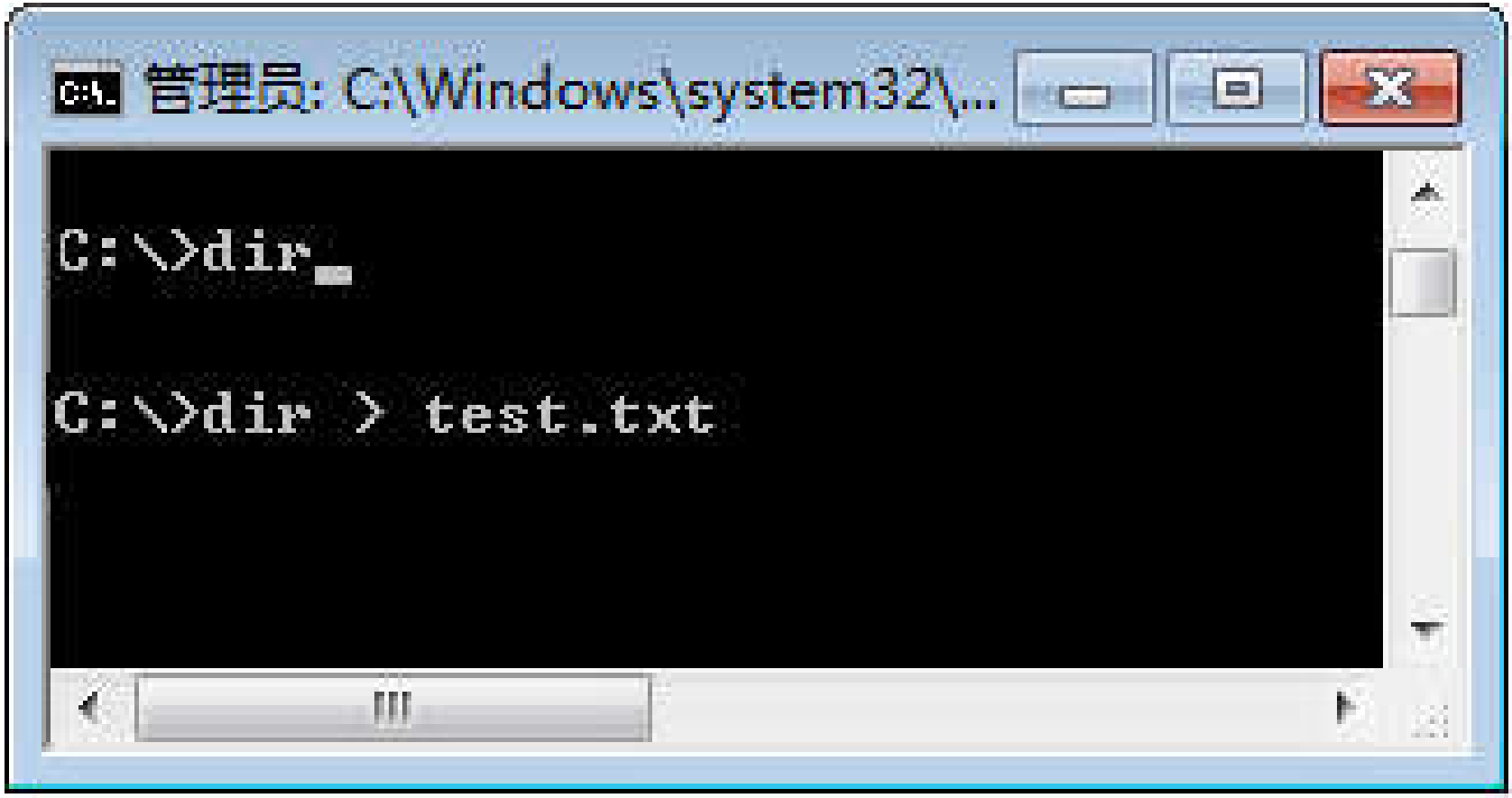
```
int testdev = open("/dev/test",O_RDWR);
if ( testdev == -1 )
{
    printf("Cann't open file ");
    exit(0);
}
```

设备管理功能>设备映射

- 设备物理名
  - I/O系统中实际安装的设备
  - 物理名：ID或字符串或主/次设备号
- 设备映射
  - 逻辑设备到物理设备的转换
    - 逻辑名到物理名的转换
- 设备独立性
  - 用户使用逻辑设备的统一接口去访问设备，而不用考虑物理设备复杂而特殊的物理操作方式和结果。
  - 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



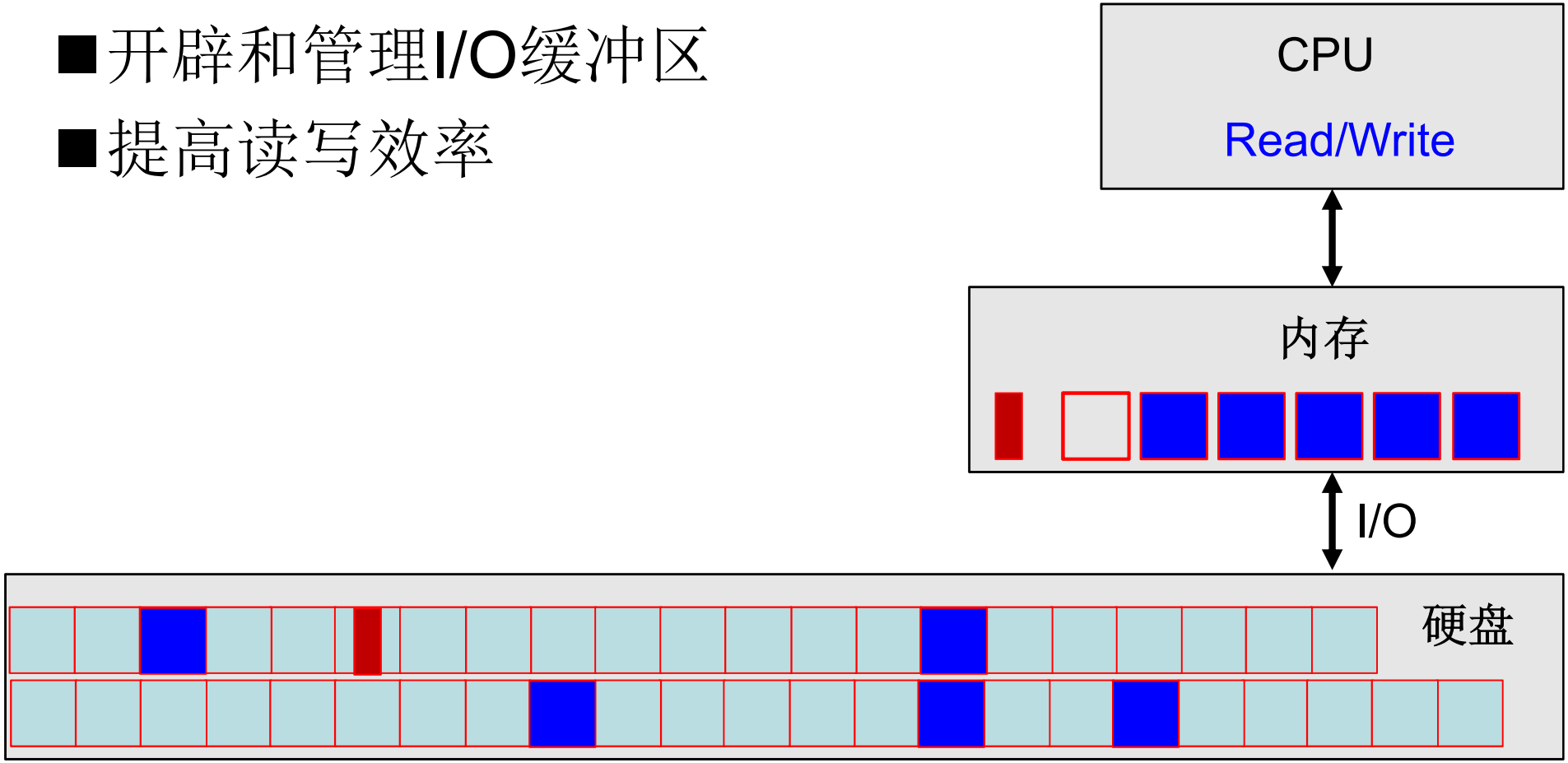
设备管理功能>设备驱动

- 设备驱动
  - 对物理设备进行控制，实现I/O操作。
  - 把应用服务请求（读/写命令）转换为I/O指令。
  - 向用户提供统一的设备使用接口
    - read/write
    - 把外设作为特别文件处理

- 设备驱动程序的特点
  - 设备驱动程序与硬件密切相关。
  - 每类设备都要配置特定的驱动程序
  - 驱动程序一般由设备厂商根据操作系统要求编写。

设备管理功能> I/O缓冲区管理

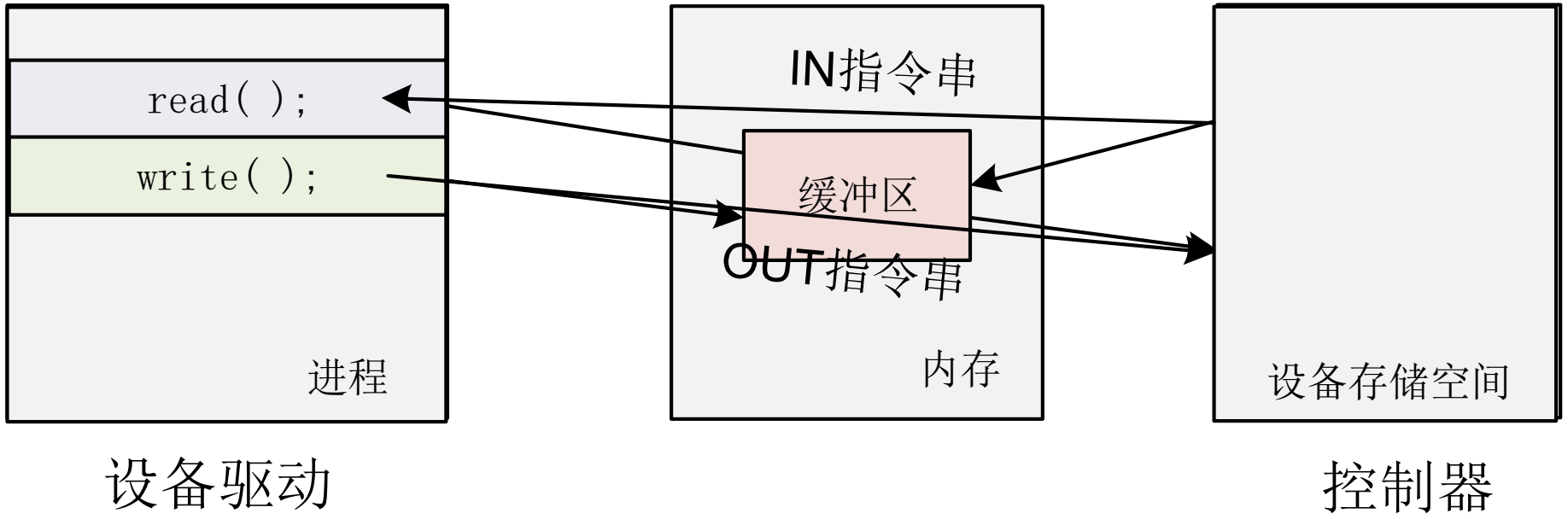
- I/O缓冲区管理
  - 开辟和管理I/O缓冲区
  - 提高读写效率



8.2 缓冲技术

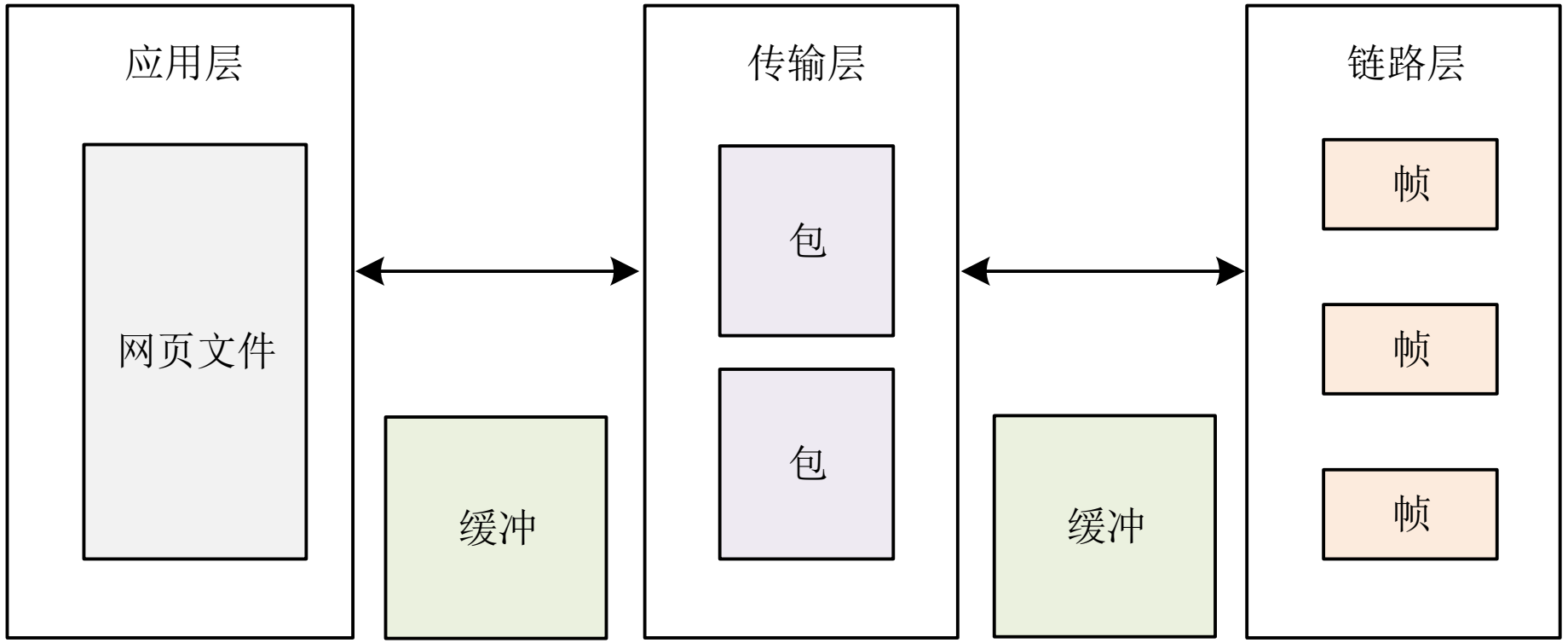
缓冲作用

- 1）连接不同数据传输速度的设备
  - 例子：设备驱动与控制器串行工作。
  - 改进：控制器增加缓冲



缓冲作用

- 2）协调数据记录大小的不一致
  - 两个设备或设备与CPU之间记录的大小不一致
  - 例：网络消息的包和帧



缓冲作用

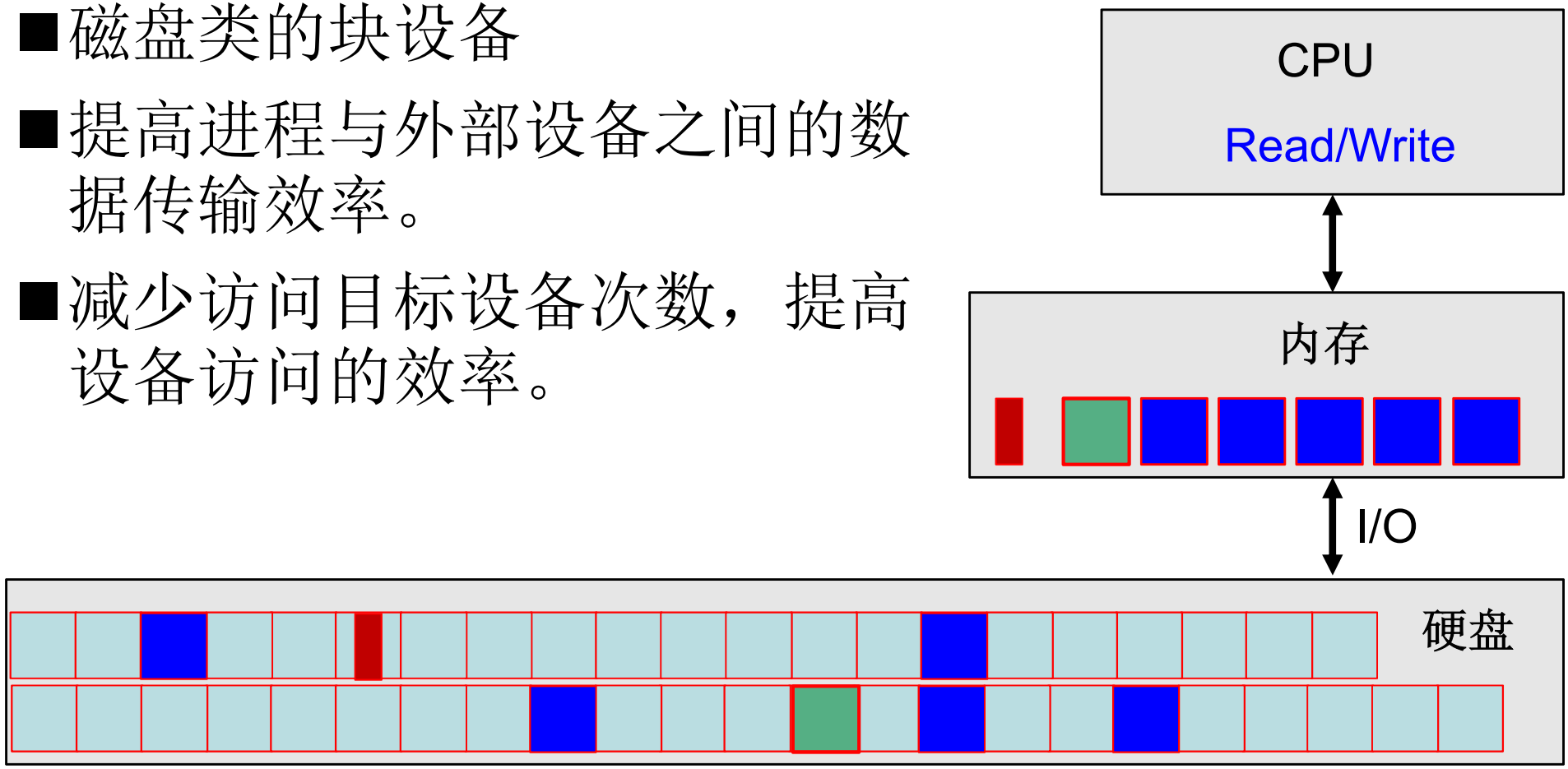
- 3）正确执行应用程序的语义拷贝
- 例子
  - 利用write(Data, Len)向磁盘写入数据Data
    - 确保写入磁盘的Data是调用时刻的Data
  - 方法：
    - 方法1
      - 应用待内核写完再返回。（实时性差）
    - 方法2
      - 内核设置缓冲区，完成内核复制即返回
      - 事后由内核把缓冲区写到磁盘。（实时性好）
      - 语义拷贝：确保事后拷贝的数据是正确版本

设备管理功能

- 设备管理的目标
  - 提高设备的利用率
  - 提高设备读写效率
  - 提高CPU与设备并行程度
  - 为用户提供统一接口
  - 实现设备对用户透明

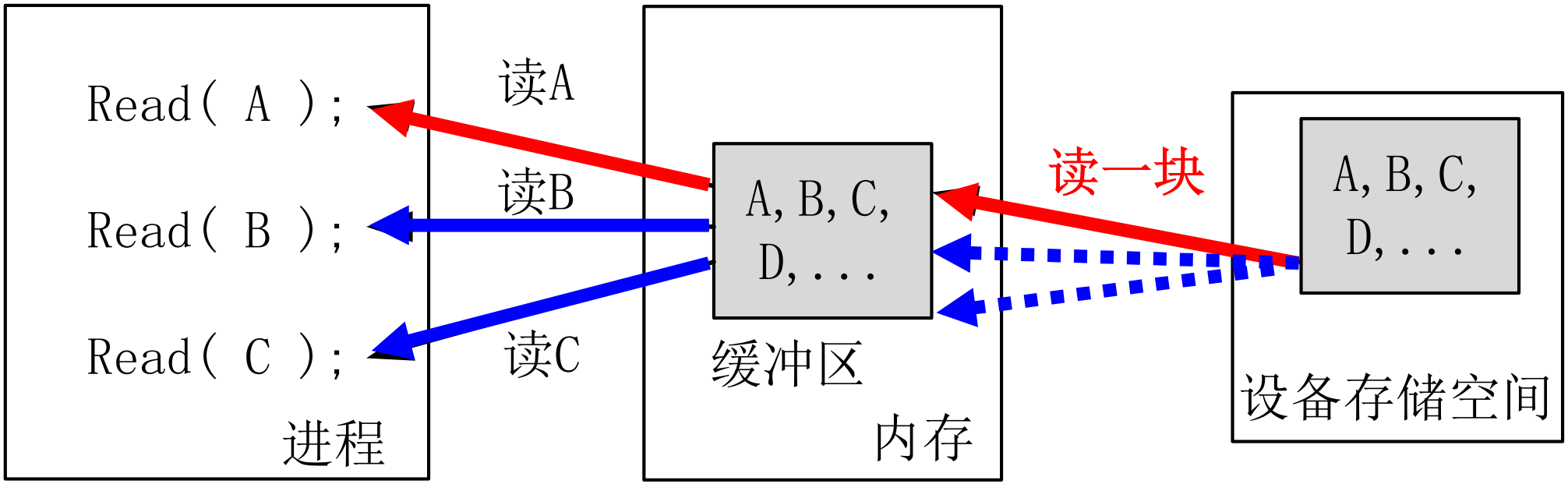
缓冲区的存取

- 提前读与延后写
  - 磁盘类的块设备
  - 提高进程与外部设备之间的数据传输效率。
  - 减少访问目标设备次数，提高设备访问的效率。



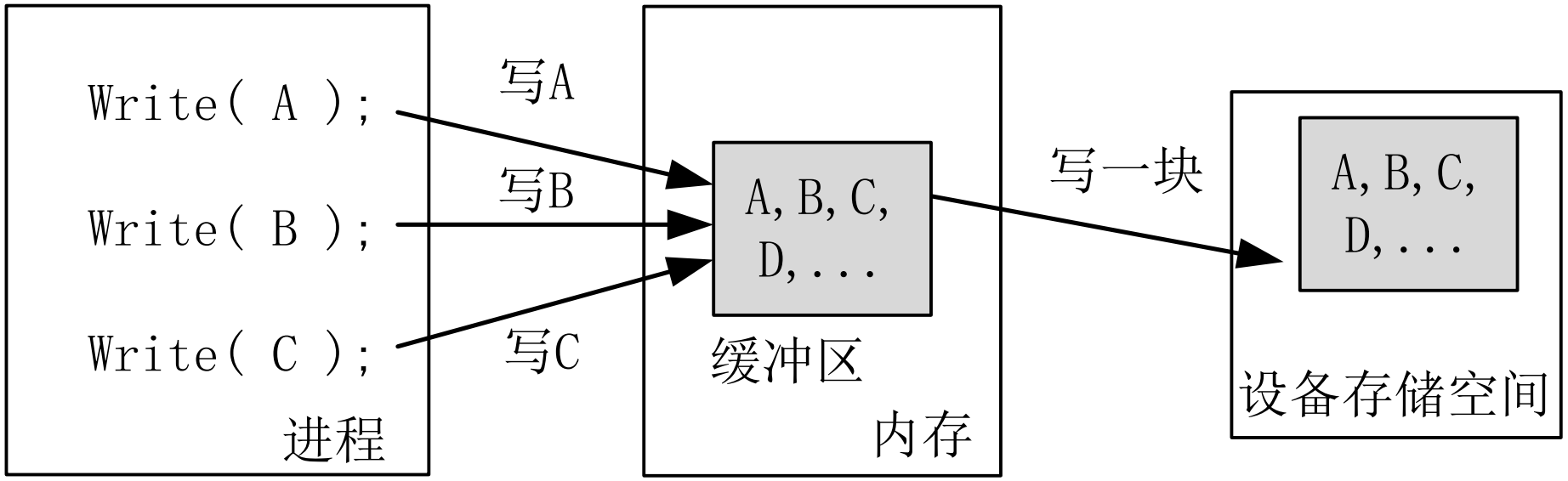
缓冲区的存取

- 提前读
  - 进程需要从外设读取的数据事先已被提前读取到了缓冲区中，不需要继续启动外设执行读取操作。



缓冲区的存取

- 延后写
  - 进程向外设写入的数据先缓存起来，延迟到特定事件发生或足够时间后，再启动外设，完成数据真正写入。

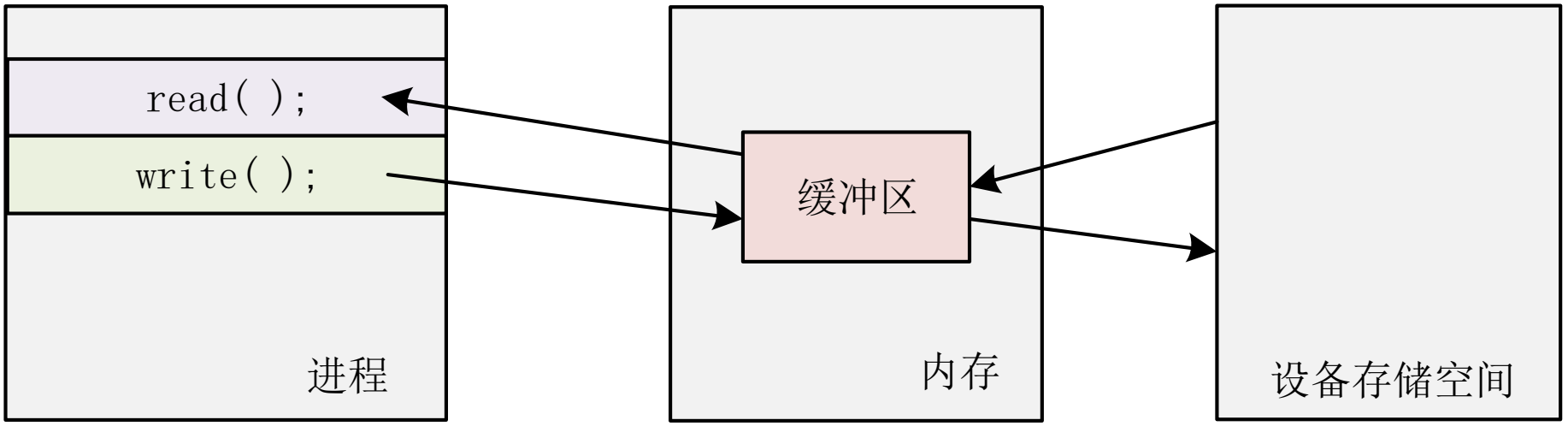


缓冲的组成

- 4种缓冲形式
  - Cache
    - 高速缓冲寄存器。
  - 设备内部缓冲区
    - 外部设备或I/O接口内部的缓冲区
  - 内存缓冲区
    - 内存开辟，应用广泛，使用灵活
  - 辅存缓冲区
    - 开辟在辅存上

缓冲的组成

- 内存缓冲区
  - 内存开辟，应用广泛，使用灵活
  - 提前读/延后写

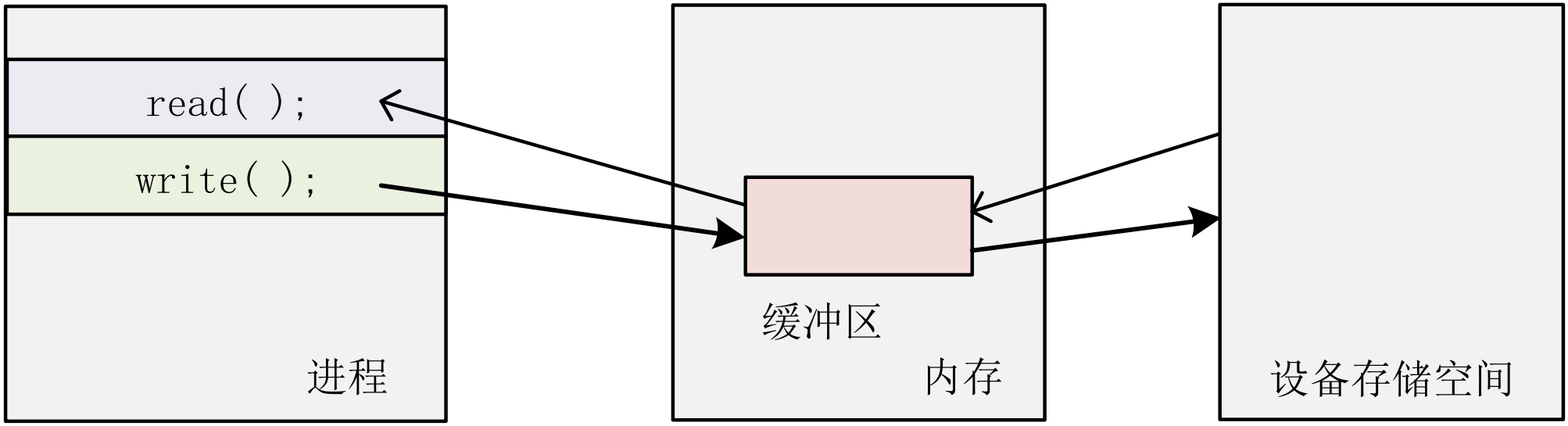


常用的缓冲技术

- 单缓冲
- 双缓冲
- 环形缓冲
- 缓冲池

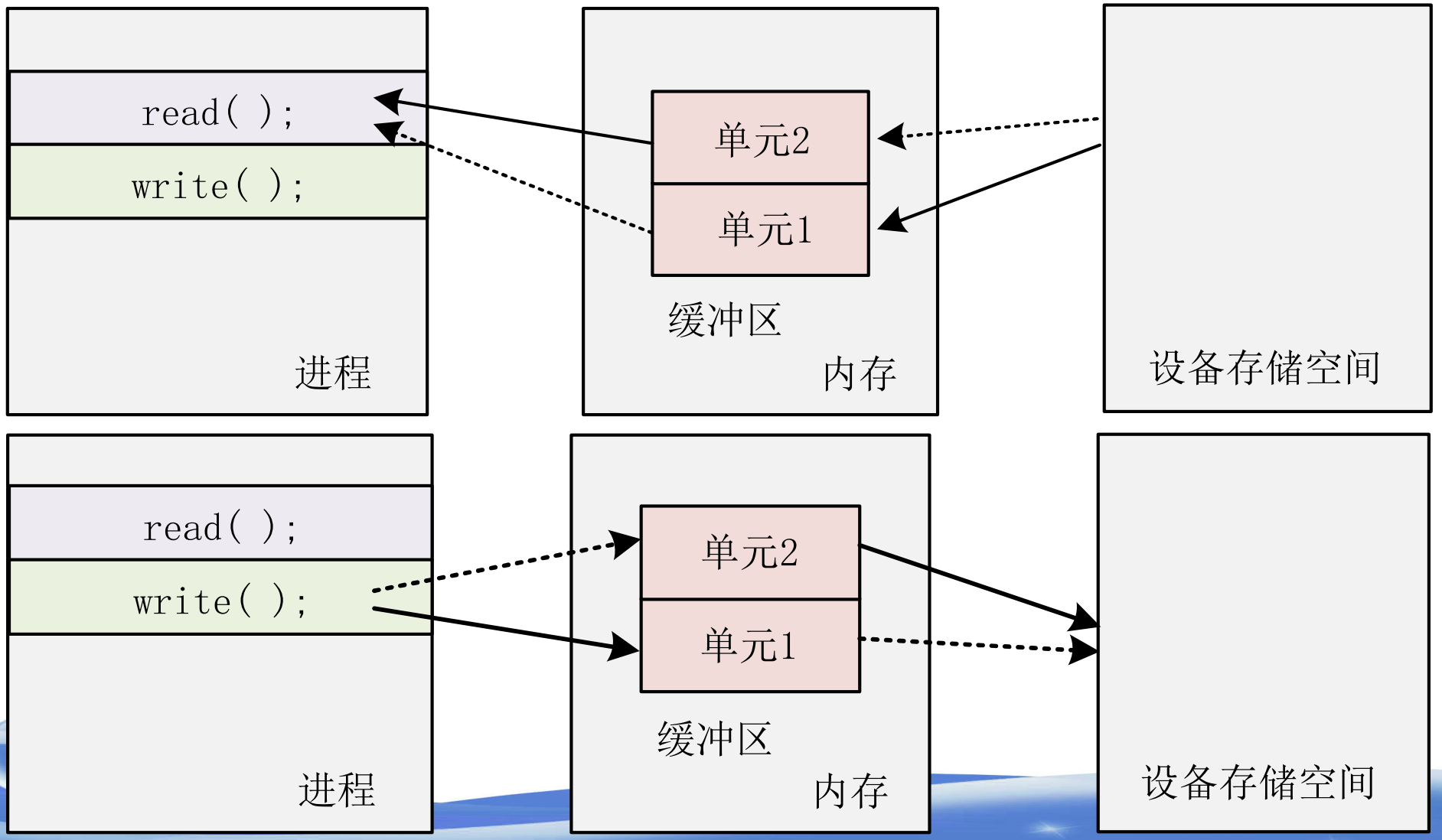
1. 单缓冲

- 单缓冲
  - 缓冲区仅有1个单元



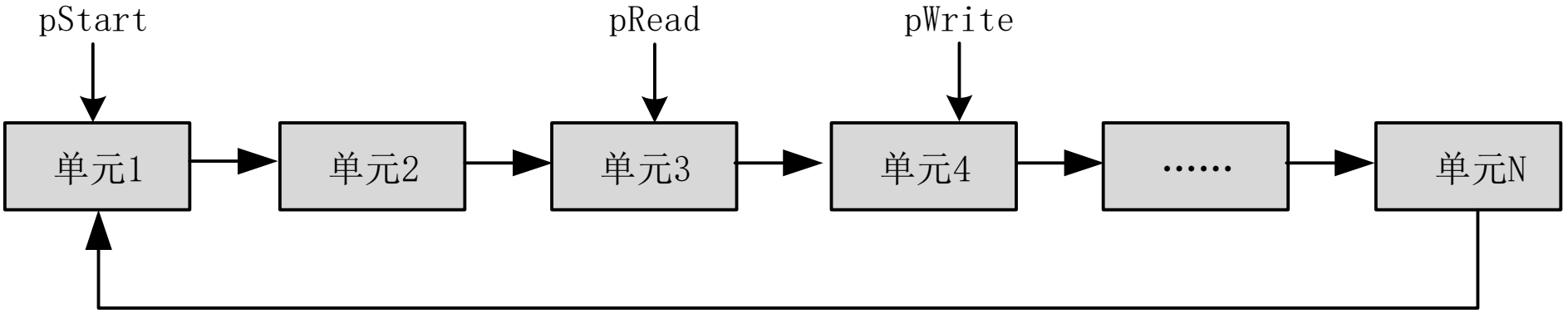
2. 双缓冲

- 双缓冲
  - 缓冲区有2个单元



3.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。



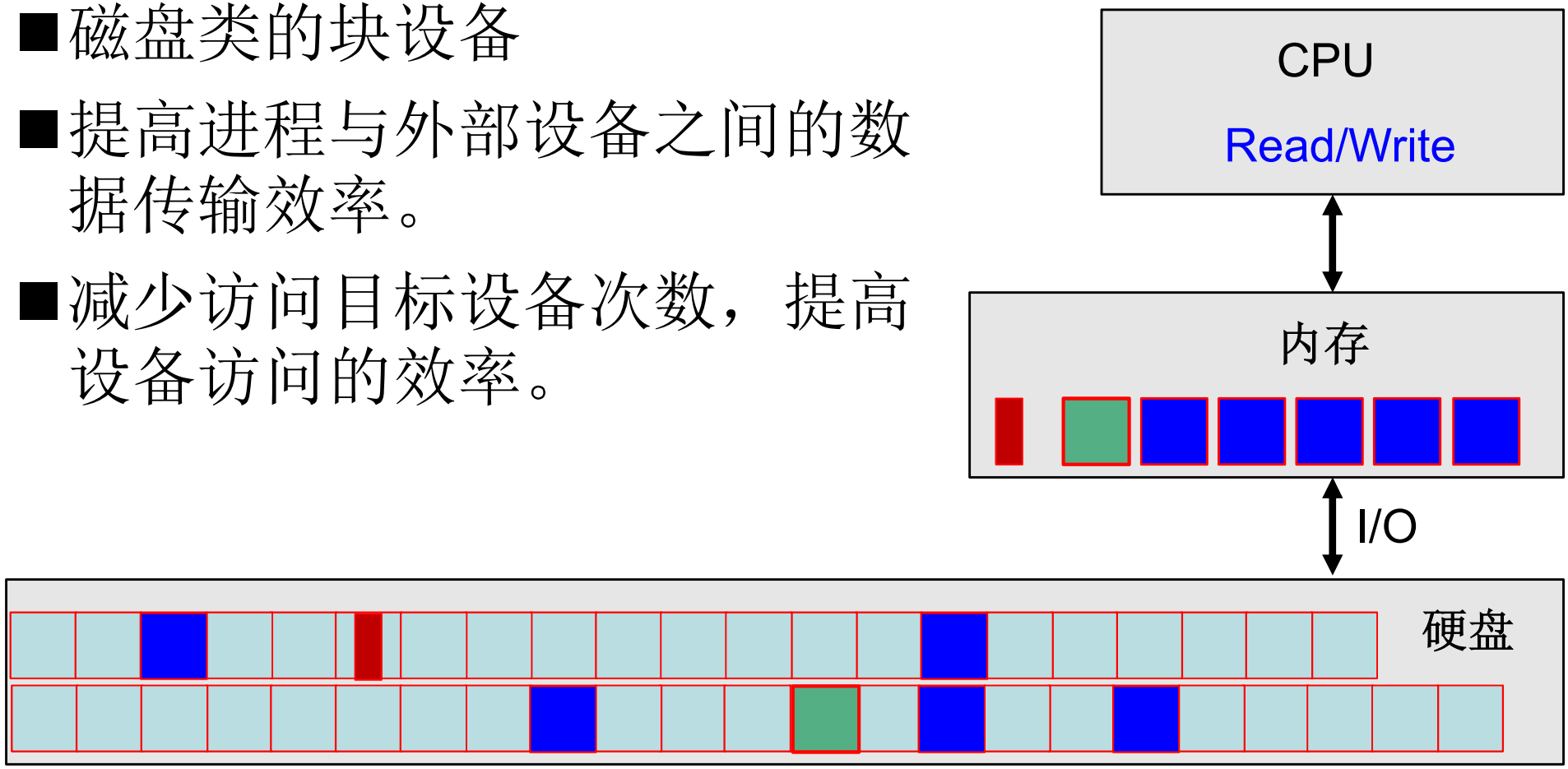
  - 起始指针pStart
  - 输入指针pWrite
  - 输出指针pRead

4.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
  - 提高缓冲区的利用率，减少内存浪费的情况。

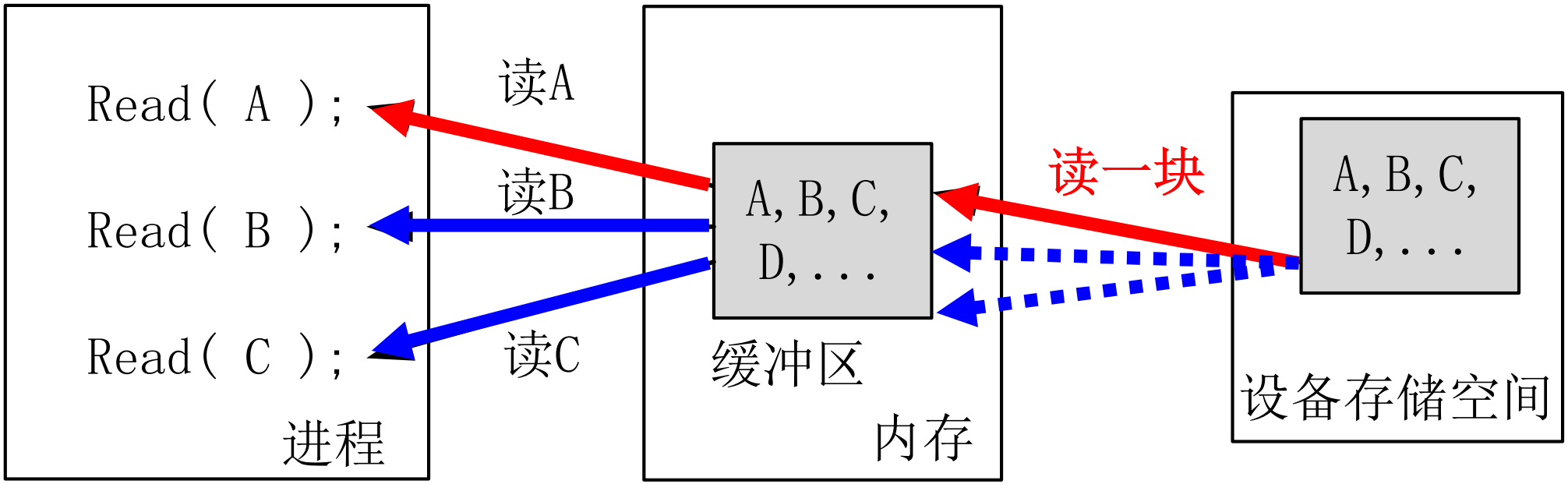
缓冲区的存取

- 提前读与延后写
  - 磁盘类的块设备
  - 提高进程与外部设备之间的数据传输效率。
  - 减少访问目标设备次数，提高设备访问的效率。



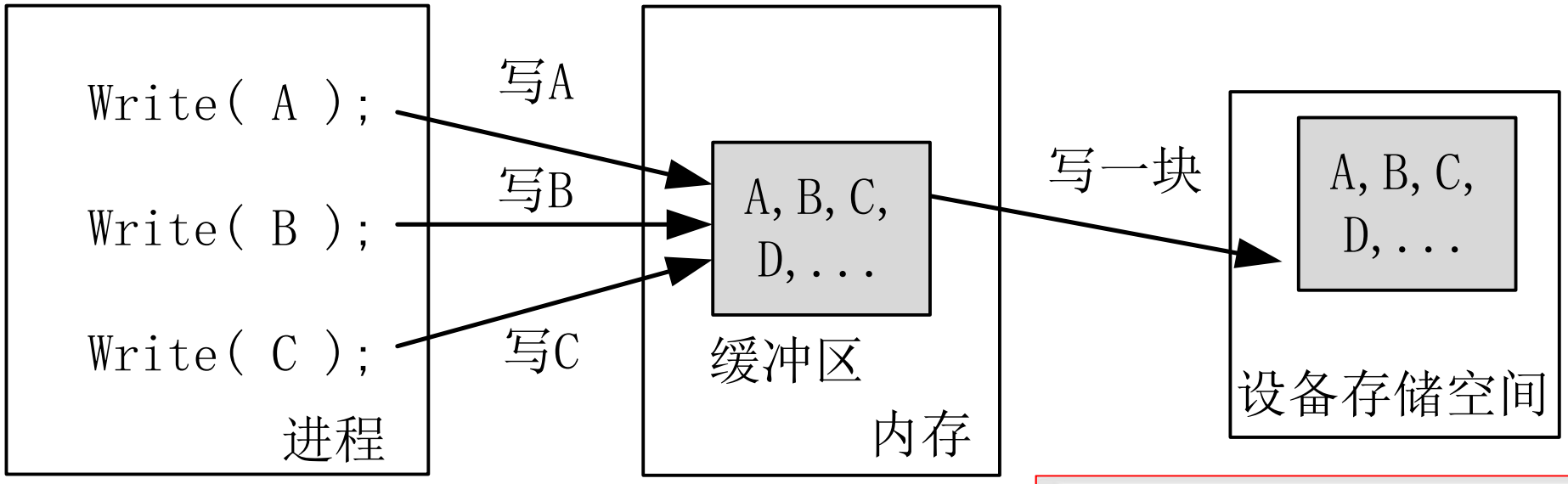
缓冲区的存取

- 提前读
  - 进程需要从外设读取的数据事先已被提前读取到了缓冲区中，不需要继续启动外设执行读取操作。

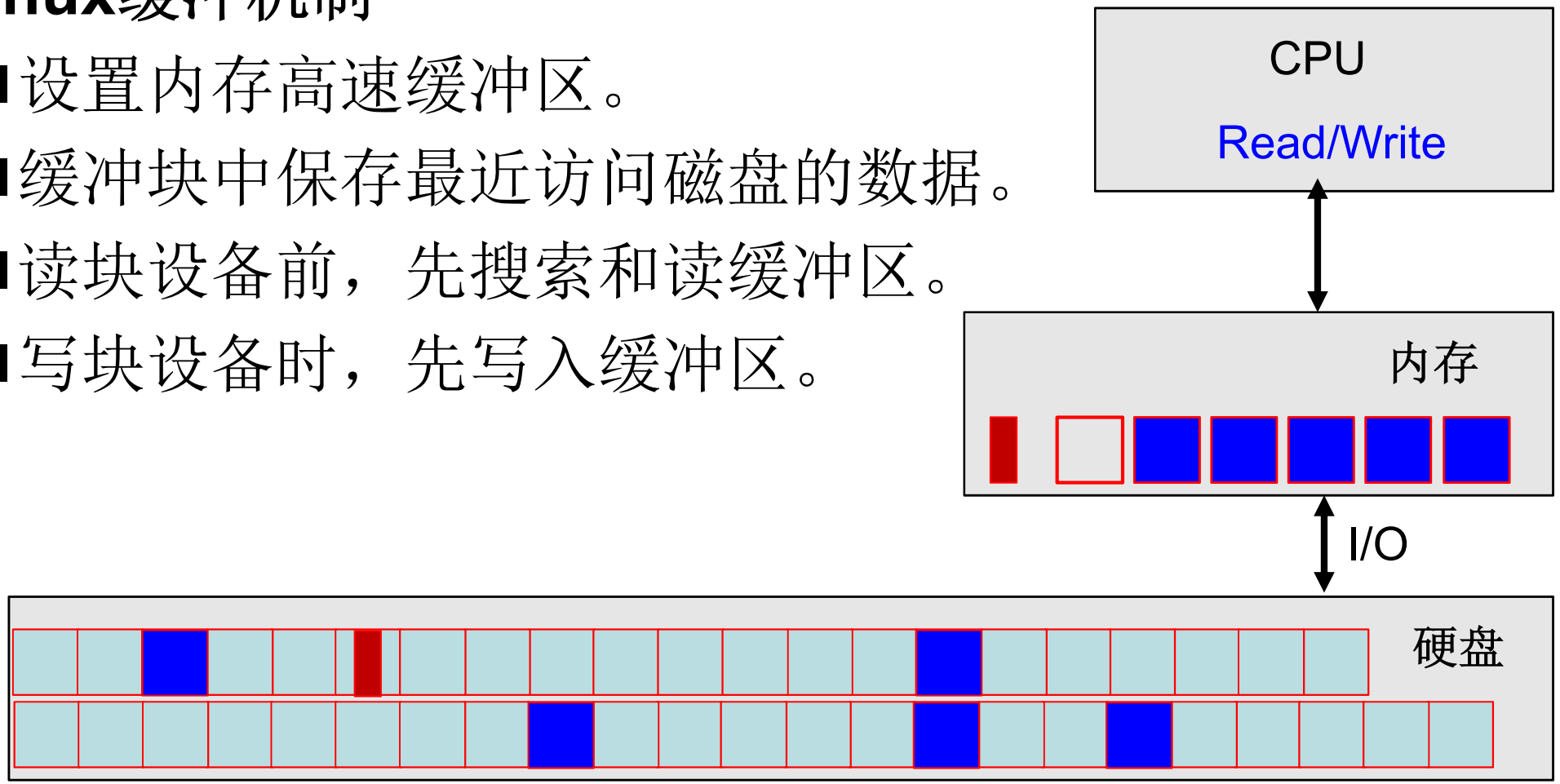


缓冲区的存取

- 延后写
  - 进程向外设写入的数据先缓存起来，延迟到特定事件发生或足够时间后，再启动外设，完成数据真正写入。

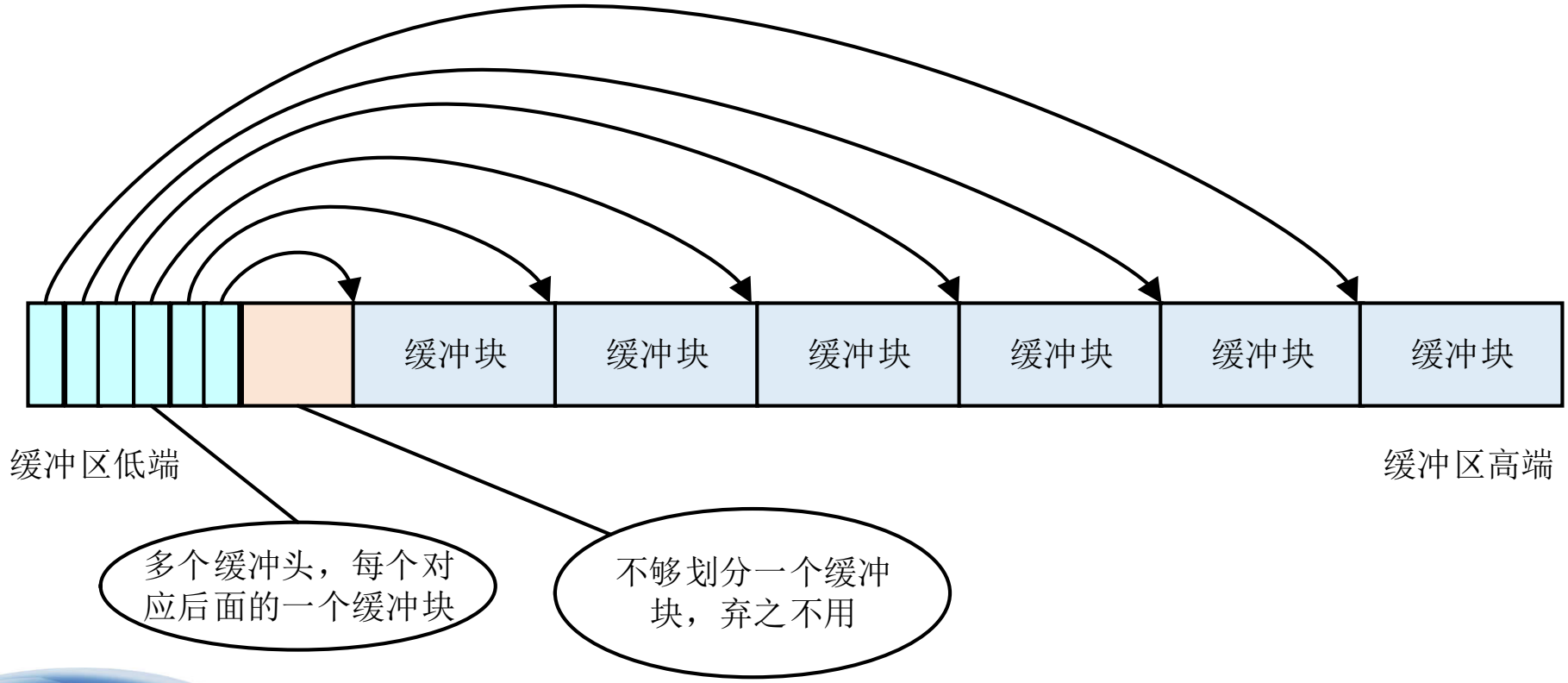


- **Linux缓冲机制**
  - 设置内存高速缓冲区。
  - 缓冲块中保存最近访问磁盘的数据。
  - 读块设备前，先搜索和读缓冲区。
  - 写块设备时，先写入缓冲区。



- 典型的块设备
  - 硬盘、软盘、RAM DISK等
  - 硬盘最小寻址单元是扇区。
  - 块（block）是文件系统数据传输单位
    - 块数倍于扇区大小。
    - Linux block是1024Byte。

- 高速缓冲区被划分为缓冲块
  - 每个缓冲块与一个磁盘块对应。每个缓冲块都用一个叫缓冲头buffer_head的结构体来描述。



8.3 设备分配

设备分类

- 独占设备
  - 不可抢占设备
  - 每次只供一个进程使用，如键盘、打印机等，只有进程释放它们之后才能被别的进程申请到。
- 共享设备
  - 可抢占设备，允许多个作业或进程同时使用。
  - CPU，内存（空分复用），存储设备（空分复用）
- 虚拟设备
  - 借助虚拟技术，在共享设备上模拟独占设备。

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
  - 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

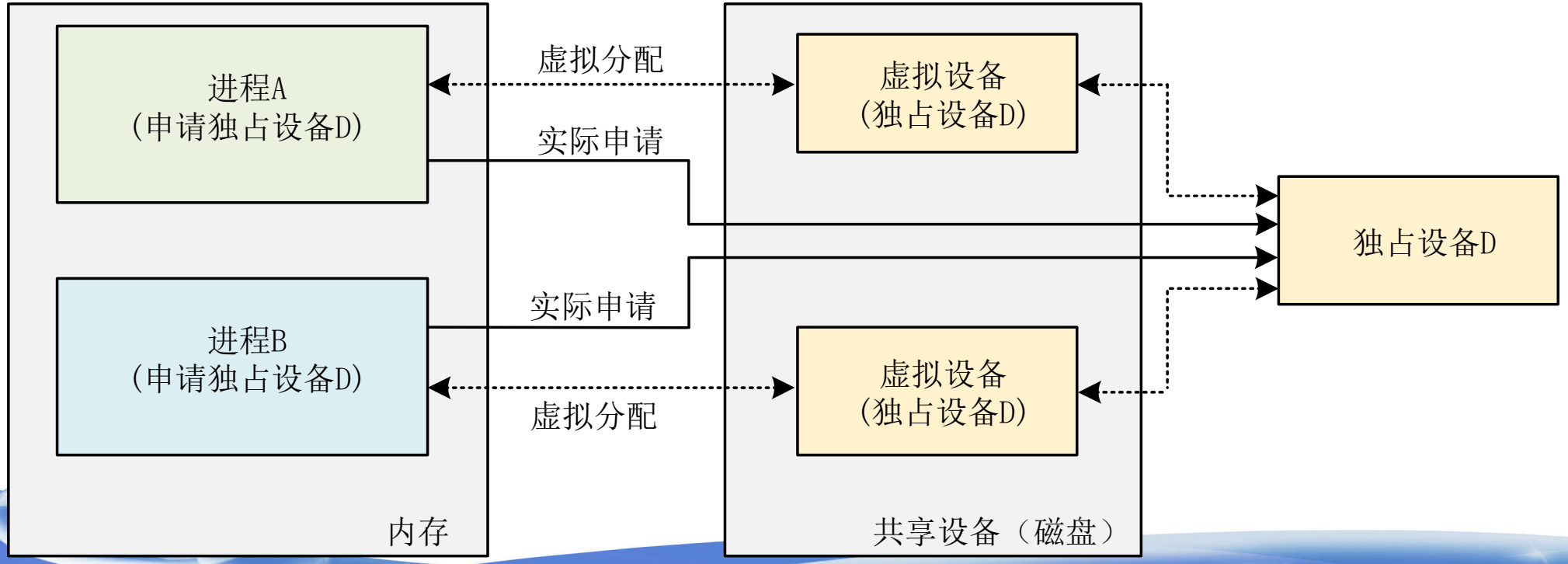
- 共享分配
  - 共享设备一般采用共享分配方式。
    - 硬盘就是典型的共享设备。
  - 当进程申请使用共享设备时，操作系统能立即为其分配共享设备的一块空间，不会让进程产生阻塞。
  - 共享分配使得进程使用设备十分简单和高效，随时申请，随时可得。

设备分配方法

- 虚拟分配
  - 虚拟技术
    - 在一类物理设备上模拟另一类物理设备的技术
    - 通常借助辅存部分区域模拟独占设备，将独占设备转化为共享设备。
  - 虚拟设备
    - 用来模拟独占设备的辅存区域称为虚拟设备
      - 具有独占设备的逻辑特点
    - 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

- 虚拟分配
  - 当进程需要与独占设备交换信息时，采用虚拟技术将与该独占设备所对应的虚拟设备分配给它。
    - 首先，采用共享分配为进程分配虚拟独占设备；
    - 然后，将虚拟设备与指定的独占设备关联。



设备分配方法

- 虚拟分配
  - 当进程需要与独占设备交换信息时，采用虚拟技术将与该独占设备所对应的虚拟设备分配给它。
    - 首先，采用共享分配为进程分配虚拟独占设备；
    - 然后，将虚拟设备与指定的独占设备关联。
  - 进程运行过程中，直接与虚拟设备进行交互，传输速度快，进程推进速度得到了提高。

设备分配方法

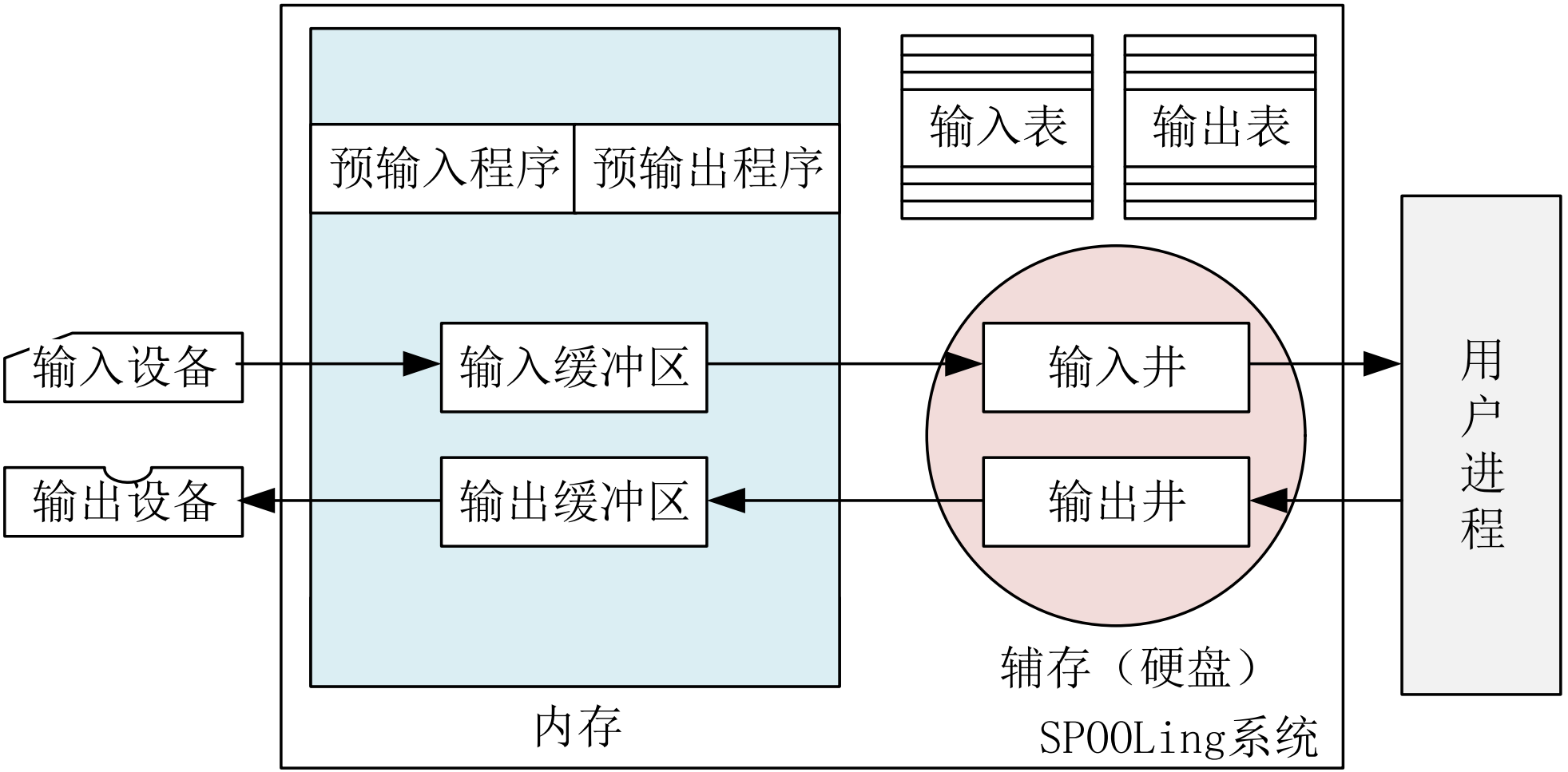
- 虚拟分配
  - 当进程需要与独占设备交换信息时，采用虚拟技术将与该独占设备所对应的虚拟设备分配给它。
  - SPOOLing系统
    - SPOOLing是虚拟技术和虚拟分配的实现
    - Simultaneaus Periphernal Operations OnLine
    - 外部设备同时联机操作
    - 假脱机输入/输出

设备分配方法

- **SPOOLing系统**
  - SPOOLing是虚拟技术和虚拟分配的实现
  - Simultaneaus Periphernal Operations OnLine
  - 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。

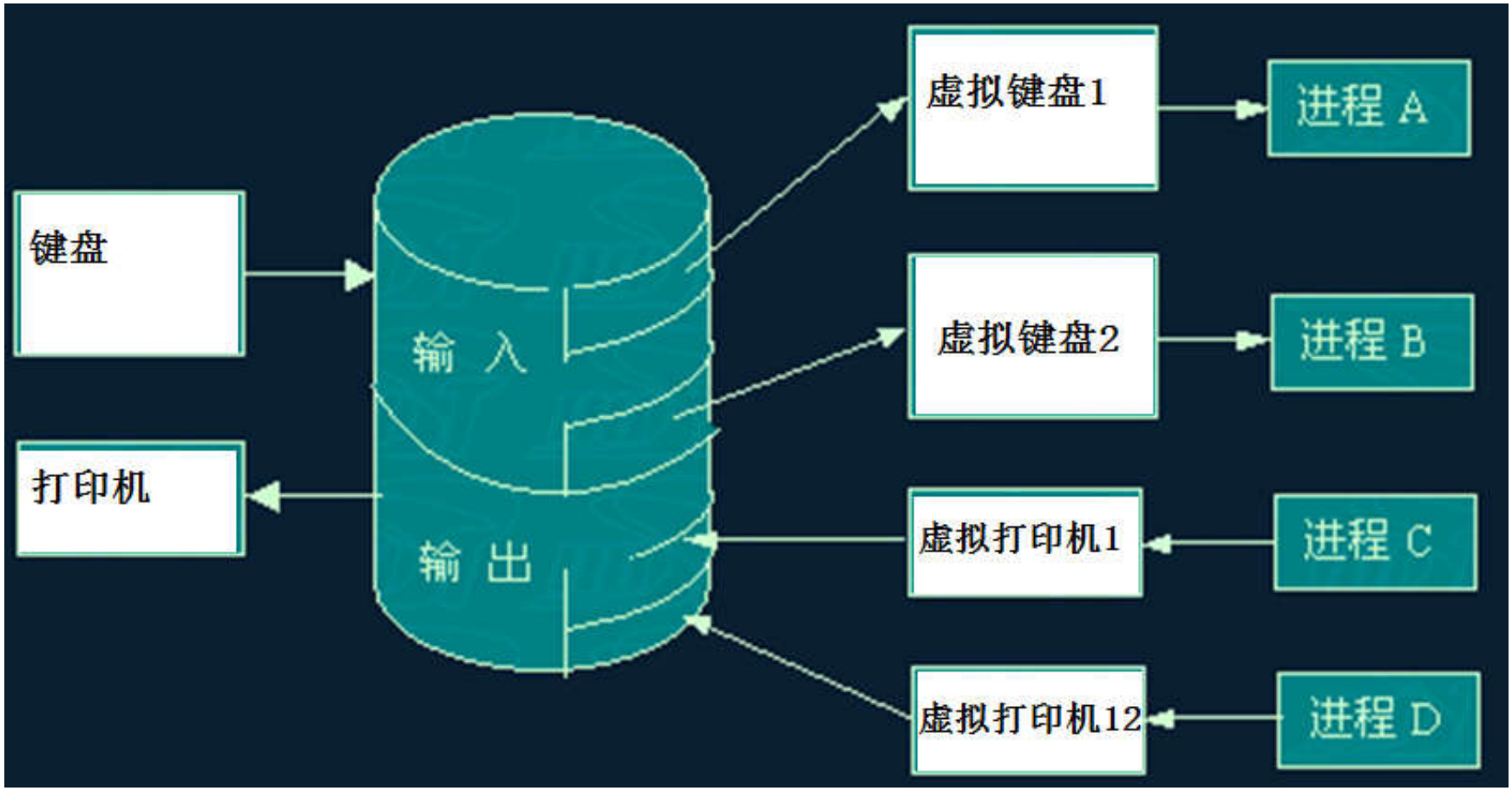
SPOOLing系统的结构（软件）

- 预输入程序
  - 控制信息从独占设备输入到辅存
- 预输入表
  - 从哪台设备输入，存放在输入井的位置；
- 缓输出程序
  - 控制信息从辅存输出到独占设备
- 缓输出表
  - 输出信息在输出井的位置，从哪台设备输出。
- 井管理程序
  - 控制用户程序和辅存之间的信息交换

SPOOLing的结构

- 预输入进程
  - 预输入进程模拟脱机输入的卫星机，将用户要求的数据从输入设备通过输入缓冲区再传送输入井。当用户进程需要数据时，直接从输入井读入所需数据；
- 缓输出监控进程
  - 缓输出进程模拟脱机输出的卫星机。用户进程将输出数据从内存先传送到输出井。当输出设备空闲时，再将输出井的数据送到输出设备上。

SPOOLing的例子



- **SPOOLing系统原理小结**
  - 任务执行前：预先将程序和数据输入到输入井中
  - 任务运行时：使用数据时，从输入井中取出
  - 任务运行时：输出数据时，把数据写入输出井
  - 任务运行完：外设空闲时输出全部数据和信息
- **SPOOLing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

8.4 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

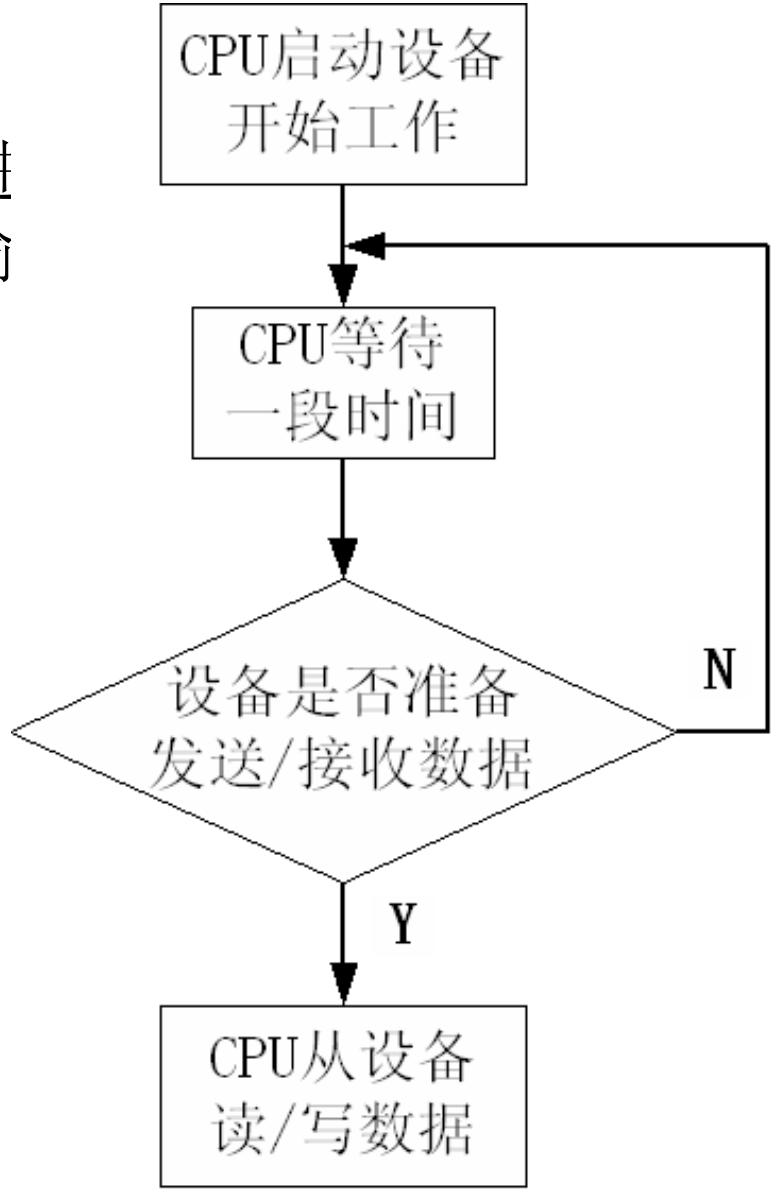
无条件传送（同步传送）

- 工作过程
  - 进行I/O时无需查询外设状态，直接进行。
  - 主要用于外设时钟固定而且已知的场合。
  - 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
  - 传送数据之前，CPU先对外设状态进行检测，直到外设准备好才开始传输
    - ◆输入时：外设数据“准备好”；
    - ◆输出时：外设 “准备好”接收。
- 特点
  - I/O操作由程序发起并等待完成
    - ◆指令：IN / OUT
  - 每次读写必须通过CPU。



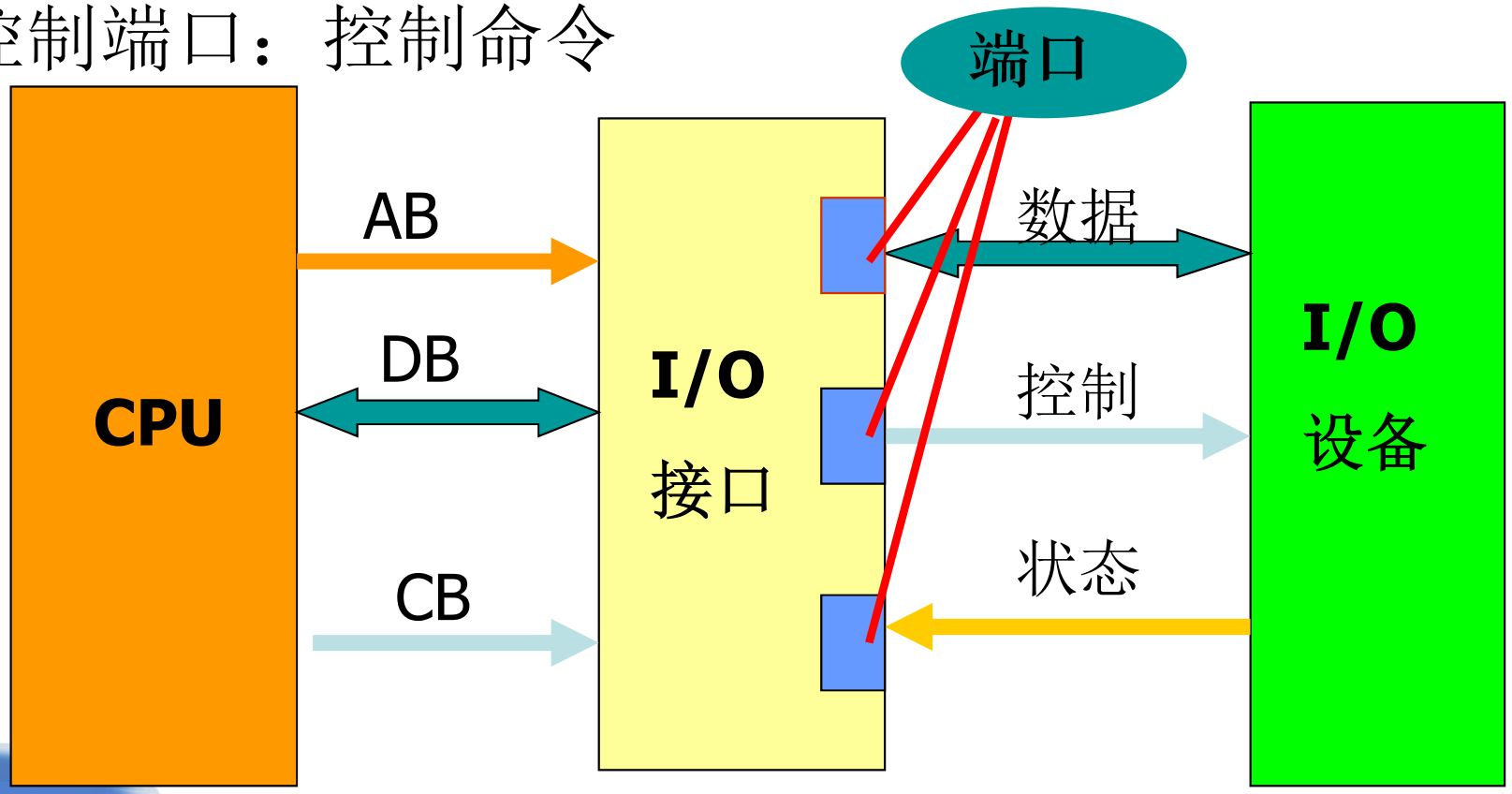
查询方式（异步传送）

- 基本原理
  - 传送数据之前，CPU先对外设状态进行检测，直到外设准备好才开始传输。

```
POLL:
     IN      AL,    PORT_State   ;读状态端口： PORT_State
     TEST    AL,    80H          ;80H是掩码检查READY位是否为1
     JZ      POLL                ;未准备好， 转POLL
     IN      AL,    PORT_Data    ;读数据端口： PORT_Data
```

```
POLL:
     IN      AL,          PORT_State ;输入状态信息
     TEST    AL,          10H        ;检查EMPTY位是否为1
     JZ      POLL                    ;外设不空（忙）转POLL
     MOV     AX,          2021H      ; 2021H是需要输出的数据
     OUT     PORT_Data,   AX         ;向数据寄存器中输出数据
```

- 外设接口的组成
  - 数据端口：暂存数据
  - 状态端口：暂存状态
  - 控制端口：控制命令



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，处理该中断事情：完成数据传输。
  - CPU处理完毕后继续原来工作。
- 特点
  - CPU和外设并行工作
  - CPU效率提高
- 缺点
  - 设备较多时中断频繁，影响CPU的有效计算能力。
  - CPU数据吞吐小（几个字节），适于低速设备。

通道方式

- 概念
  - 通道是用来控制外设与内存数据传输的专门部件。
  - 通道有独立的指令系统，既能受控于CPU又能独立于CPU。
  - I/O处理机
- 特点
  - 有很强I/O能力，提高CPU与外设的并行程度
  - 以内存为中心，实现内存与外设直接数据交互。
  - 传输过程基本无需CPU参与。

DMA(直接内存访问)方式

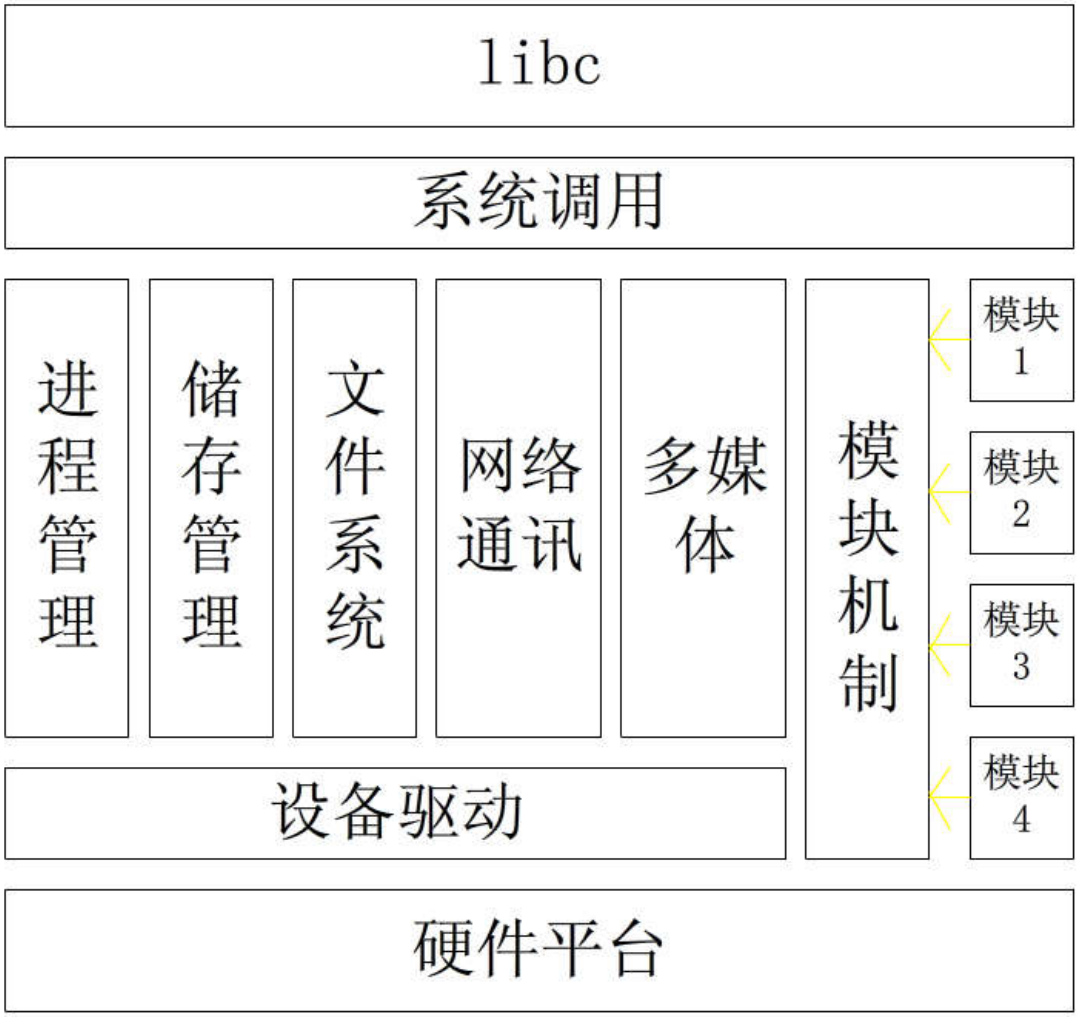
- 概念
  - Direct Memory Access
  - 外设和内存之间直接进行数据交换，不需CPU干预
  - 只有数据传送开始（初始化）和结束时（反初始化）需要CPU参与。传输过程不需要CPU参与。
  - DMA控制器：DMA Controller（DMAC）
  - DMA的局限
    - 不能完全脱离CPU
      - 传送方向，内存始址，数据长度由CPU控制
    - 每台设备需要一个DMAC
      - 设备较多时不经济
  - 微机广泛采用

8.5 设备驱动程序

- Linux模块机制
- Linux驱动(LDD: Linux Device Driver)
- Windows驱动（WDM）

8.5.1 Linux模块

- **Linux内核模块**
  - Loadable Kernel Module：LKM
  - 一种未经链接的可执行代码。
  - 可以动态地加载或卸载模块。
  - 经过链接可成为内核一部分。
  - 设备驱动可通过模块方式添加到内核



最简单的模块程序

```
#include <linux/module.h>
static int hello_init(void)
{
    printk("Hello, Kernel!\n");
    return 0;
}
static void hello_exit()
{
    printk(“Exit Kernel!\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

最简单的模块程序

```
static int hello_init(void)
{
    printk("Hello, Kernel!\n");
    return 0;
}
static void hello_exit()
{
    printk(“Exit Kernel!\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

- 安装模块

  $sudo insmod hellomodule.ko

  ```
  [27948.531569]Hello Kernel!
  ```

- 删除模块

  $sudo rmmod module

  ```
  [27520.195551]Exit Kernel!
  ```

- 编译模块

  $gcc -o hellomodule.ko _D__KERNEL__ -DMODULE hello.c

- 安装模块

  $sudo insmod hellomodule.ko

  ```
  [27948.531569]Hello Kernel!
  ```

- 查看模块

  $lsmod

- 删除模块

  $sudo rmmod module

  ```
  [27520.195551]Exit Kernel!
  ```

- 查看内核信息

  $dmesg

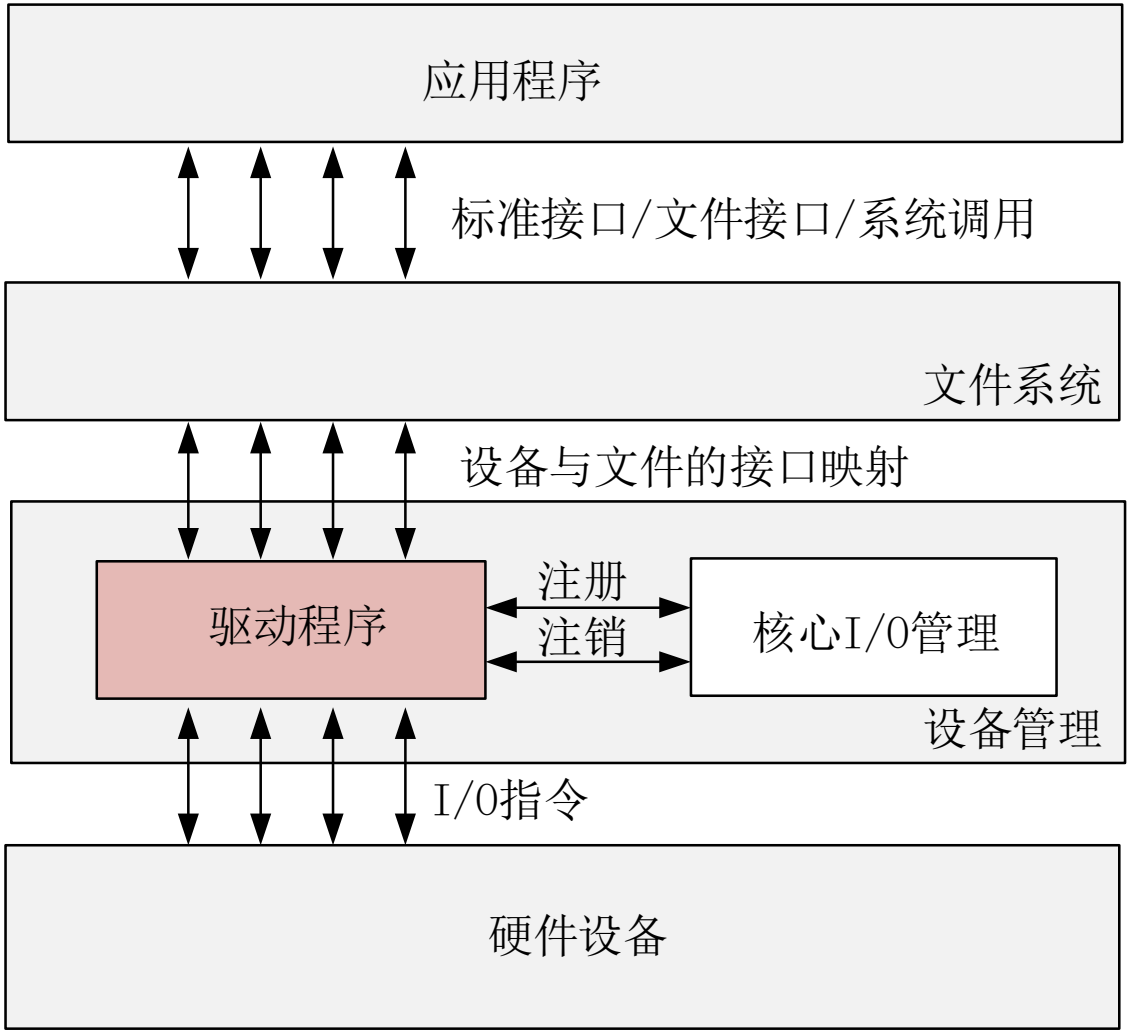
8.5.2 Linux设备驱动(LDD)

设备驱动概念

- 程序访问设备有两种典型的方法
  - 方法一
    - 直接通过I/O指令操作硬件
  - 方法二
    - 通过系统调用间接控制硬件

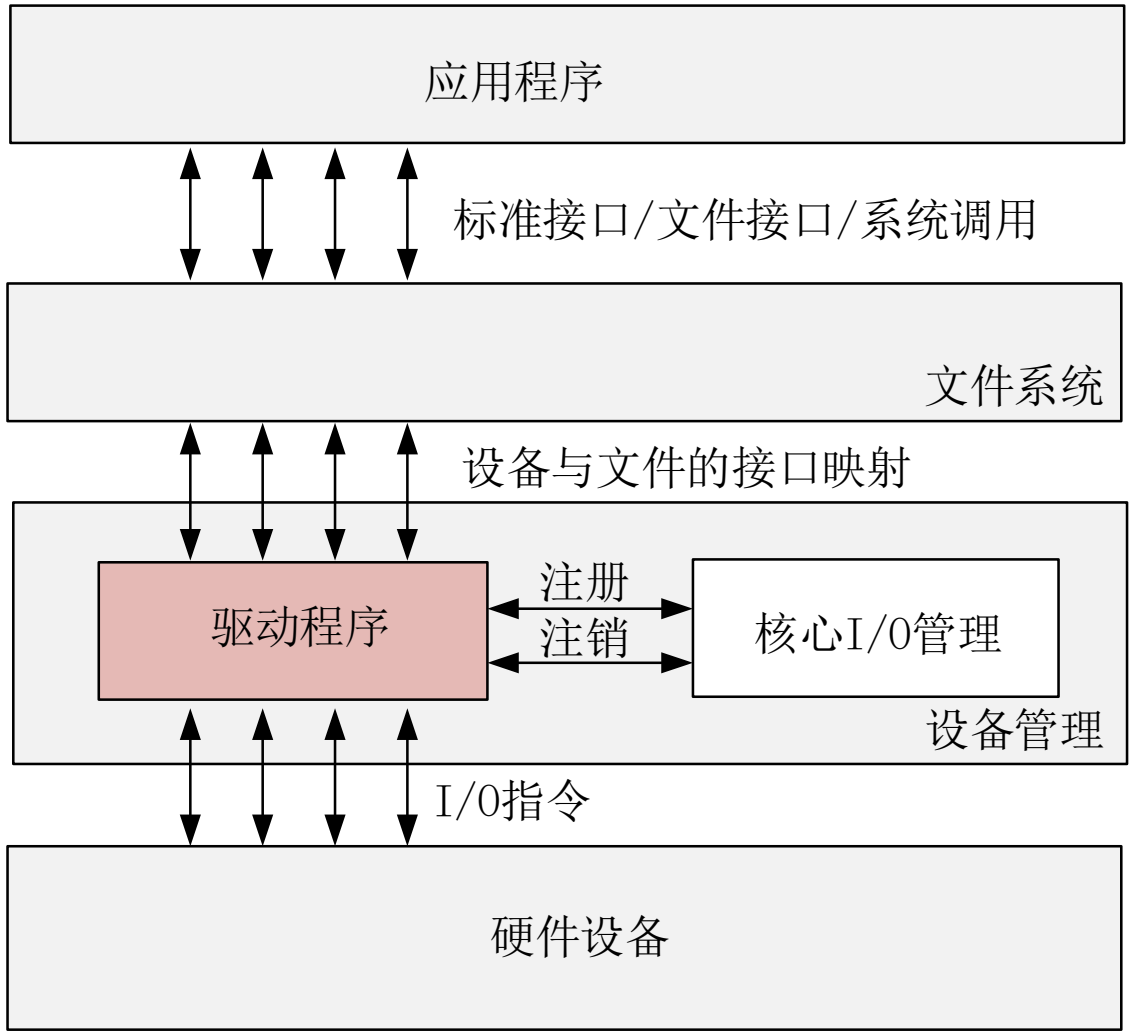
驱动程序在系统中的地位

- 设备驱动程序
  - Device Driver
  - 硬件设备的接口程序，直接控制硬件各种操作。
  - 向上为文件系统接口提供服务
  - 向下执行I/O指令控制硬件设备工作



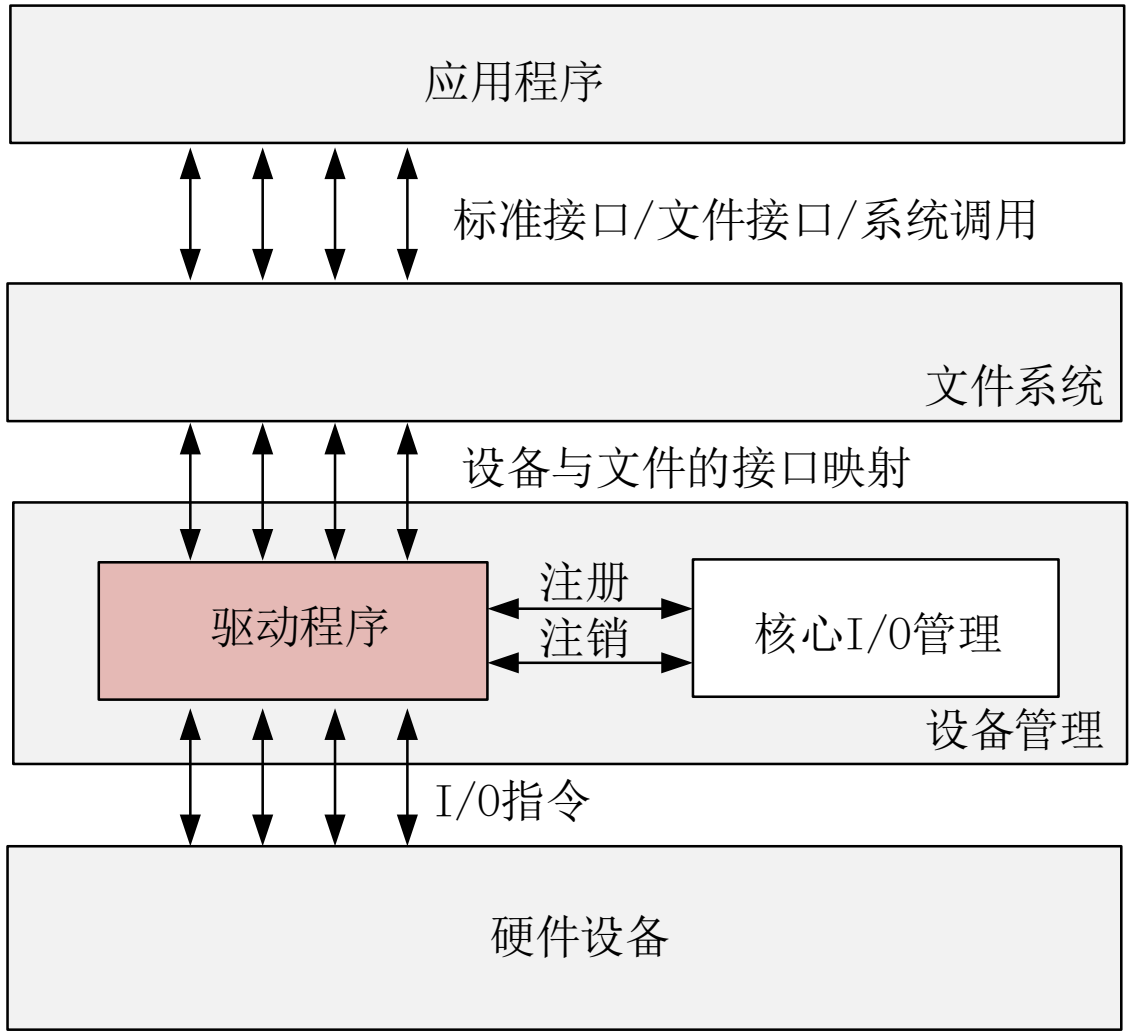
驱动程序在系统中的地位

- 驱动程序的基本接口
  - 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



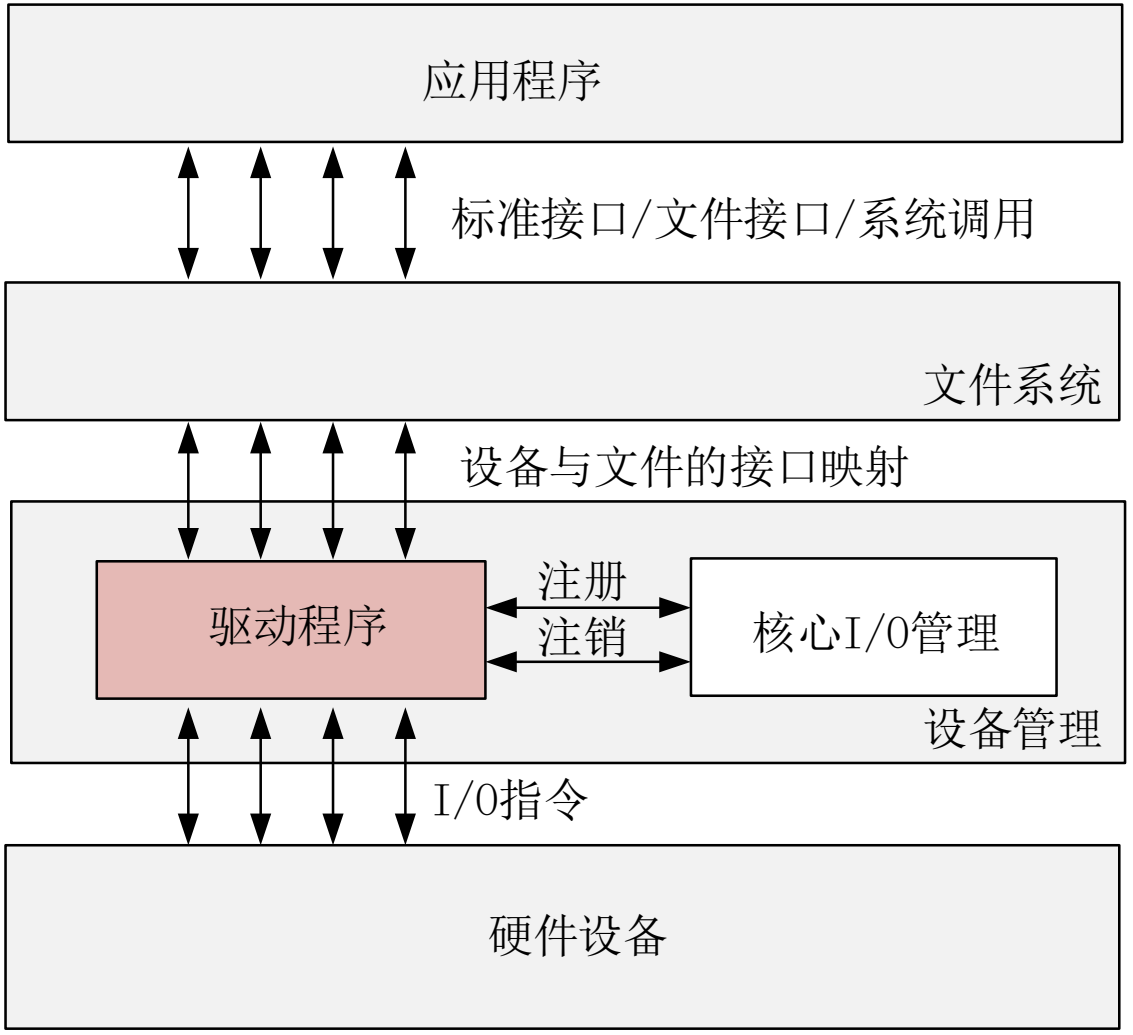
驱动程序在系统中的地位

- 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

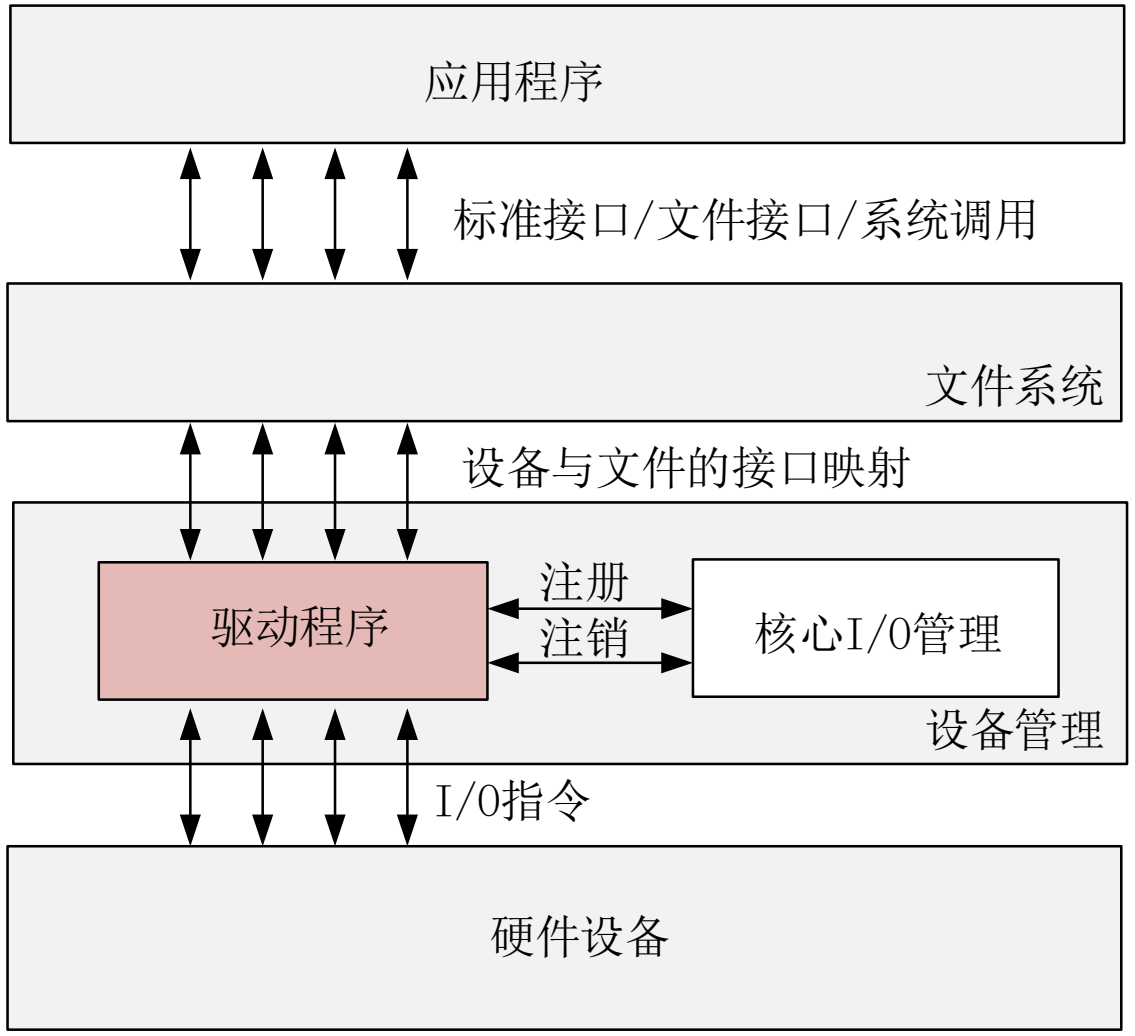


驱动程序在系统中的地位

- 面向I/O管理器的接口
  - 注册函数
    - insmod
  - 注销函数
    - rmmod
  - 必需的数据结构

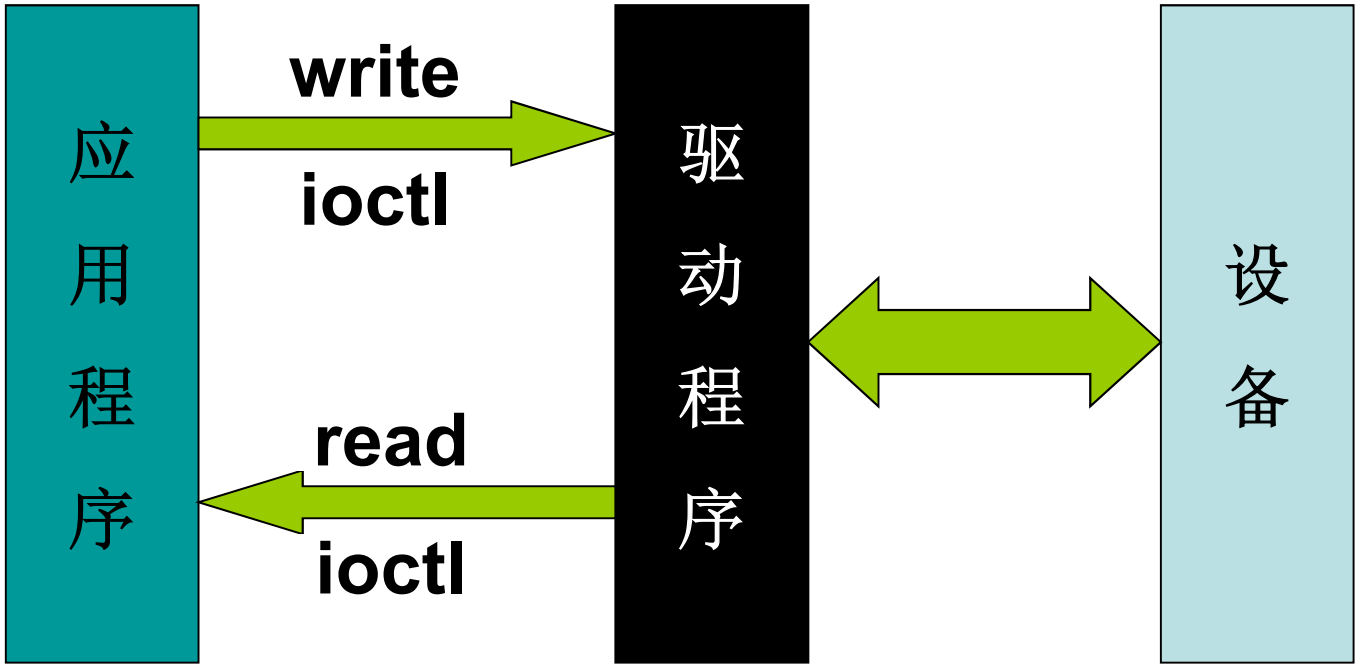


- 面向设备的接口
  - 实现与设备相关一系列的端口操作
    - 无条件传送
    - 查询传送
    - 中断传送
    - DMA传送



- **Linux设备的分类**
  - 字符设备
    - 以字节为单位逐个进行I/O操作
    - 字符设备中的缓存是可有可无
    - 不支持随机访问
    - 如串口设备
  - 块设备
    - 块设备的存取是通过buffer、cache来进行
    - 可以进行随机访问
    - 例如IDE硬盘设备
    - 支持可安装文件系统
  - 网络设备
    - 通过BSD套接口访问（SOCKET）

- LDD程序概念

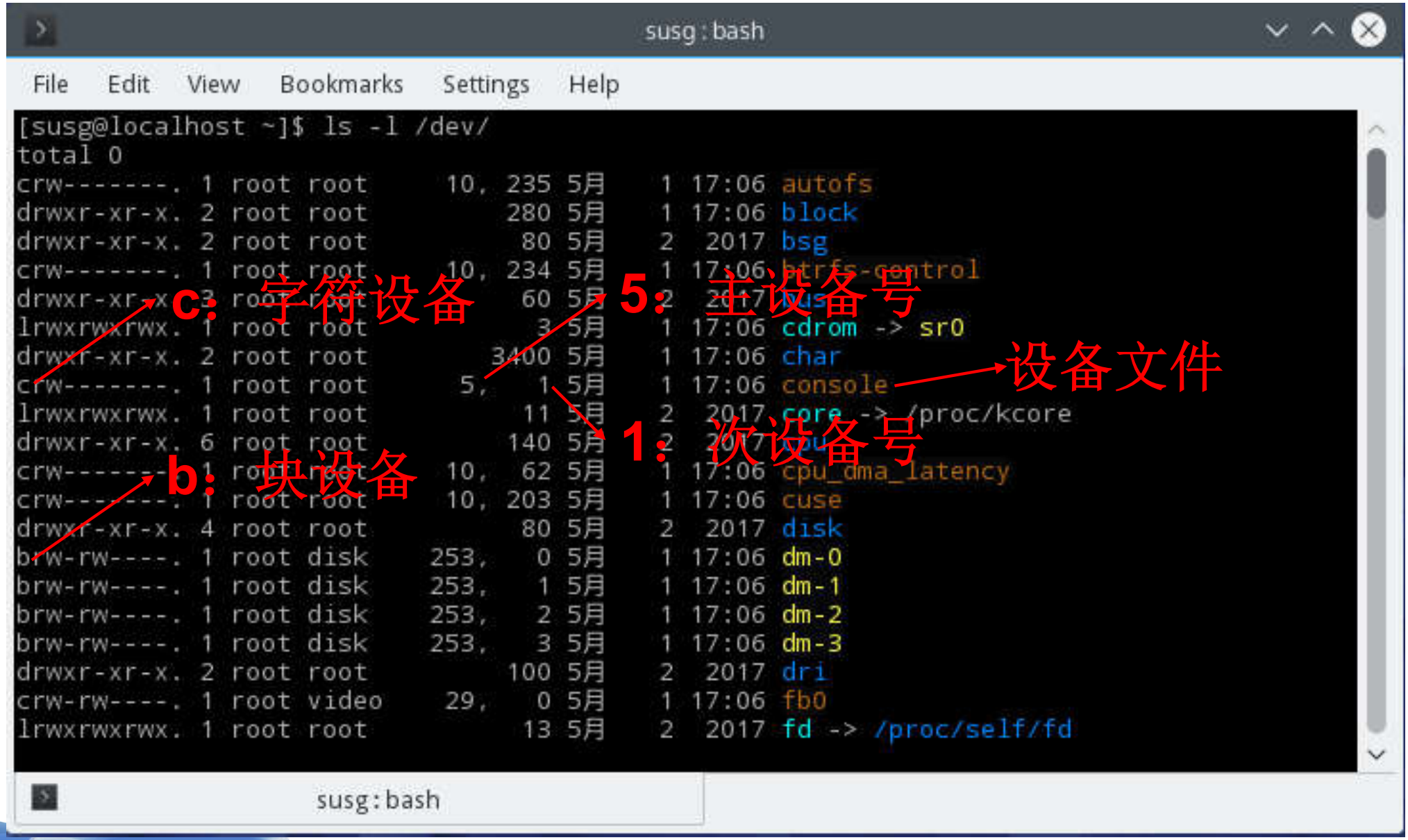


LDD程序概念

- 用户态与内核态
  - Linux的两种运转模式。
    - 内核态：
    - 用户态：
  - 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
    - put_user
    - copy_from_user
    - copy_to_user

- 设备文件
  - 硬件设备作为文件看待
  - 可以使用和文件相同的调用接口来完成打开、关闭、读写和I/O控制等操作
  - 字符设备和块设备通过设备文件访问。
    - Linux文件系统中可以找到（或者使用mknod创建）设备对应的文件，这种文件为设备文件。

列出设备文件 ls –l /dev



- 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

功能完整的Linux设备驱动程序结构

- 功能完整的LDD结构
  - 驱动程序/模块安装接口(设备注册)
  - 驱动程序/模块删除接口(设备注销)
  - 设备的打开
  - 设备的释放
  - 设备的读操作
  - 设备的写操作
  - 设备的控制操作
  - 设备的中断和轮询处理

- 例子：字符设备驱动程序
  - LDD程序概念
  - LDD程序结构
  - LDD程序加载方式
  - LDD应用程序测试

简单字符驱动程序的例子：实现了5个函数

```
static int my_open(struct inode * inode, struct file * filp)
{ 设备打开时的操作 … }

static int my_release(struct inode * inode, struct file * filp)
{ 设备关闭时的操作 … }

static int my_write(struct file *file, const char * buffer, size_t count,
loff_t * ppos)
{ 设备写入时的操作 … }

static int __init my_init(void)
{设备的注册：初始化硬件，注册设备，创建设备节点… }

static void __exit my_exit(void)
{设备的注销：删除设备节点，注销设备… }
```

打开设备和关闭设备

- my_open和my_release在设备打开和关闭时调用

```
static int my_open(struct inode * inode, struct file * filp){
        MOD_INC_USE_COUNT;
        return 0;
}
```

```
static int my_release(struct inode * inode, struct file * filp
        MOD_DEC_USE_COUNT
        return 0;
}
```

  - MOD_INC_USE_COUNT
  - MOD_DEC_USE_COUNT

写操作

```
static int my_write(struct file *file, const char * buffer, size_t count,
loff_t * ppos){
    char led_status = 0;
    copy_from_user(&led_sta
    if(led_status == 0x01) {
            SetOutput(LED);
    } else {
            ClearOutput(LED);     //非1则熄灭LED
    }
    return 0;
}
```

```
fd = open("/dev/my_led", O_RDWR);

write(fd, &led_on, 1);
close(fd);
```

打开设备和关闭设备

- my_open和my_release在设备打开和关闭时调用

```
static int my_open(struct inode * inode, struct file * filp){
        MOD_INC_USE_COUNT;
        return 0;
}
```

```
static int my_release(struct i
        MOD_DEC_USE_COUNT
        return 0;
}
```

```
fd = open("/dev/my_led", O_RDWR);

write(fd, &led_on, 1);
close(fd);
```

  - MOD_INC_USE_COUNT
  - MOD_DEC_USE_COUNT

文件操作接口：文件操作结构体

```
struct   file_operations{
  struct   module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t(*read) (struct  file *, char *, size_t, loff_t*);
  ssize_t(*write) (struct  file *, const char *, size_t, loff_t*);
  int(*readdir) (struct file *, void *, filldir_t);
  unsigned int(*poll) (struct file *, struct poll_table_struct *);
  int(*ioctl) (struct inode*, struct file *, unsigned int, unsigned long);
  int(*mmap) (struct file *, struct vm_area_struct *);
  int(*open) (struct inode*, struct file *);
  int(*flush) (struct file *);
  int(*release) (struct inode*, struct file *);
  int(*fsync) (struct file *, struct dentry*, intdatasync);
  int(*fasync) (int, struct file *, int);
  int(*lock) (struct file *, int, struct file_lock*);
  ssize_t(*readv) (struct file *, const struct iovec*, unsigned long, loff_t*);
  ssize_t(*writev) (struct file *, const struct iovec*, unsigned long, loff_t*);
  ssize_t(*sendpage) (struct file *, struct page *, int, size_t, loff_t*, int);
  unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
};
```

文件操作结构体初始化

```
fd = open("/dev/my_led", O_RDWR);

write(fd, &led_on, 1);
close(fd);
```

```
static struct file_operations my_fops = {
        open:   my_open,
        write:  my_write,
        release: my_release
};
```

设备注册（初始化）

```
static int __init my_init(void){
    //硬件初始化
    Enable(LED);
    OutputEnable(LED);
    //字符设备注册
    Led_Major = register_chrdev(0, DEVICE_NAME, &my_fops);
    // 创建设备文件，也可mknod手工创建
    #ifdef CONFIG_DEVFS_FS
        Devfs_Led_Dir = devfs_mk_dir(NULL, “my_led", NULL);
        Devfs_Led_Raw = devfs_register(Devfs_Led_Dir, "0",
            DEVFS_FL_DEFAULT, Led_Major, 1,
      S_IFCHR|S_IRUSR|S_IWUSR,&my_fops, NULL);
    #endif
}
```

设备注销（反初始化）

```
static void __exit my_exit(void)
{
    //删除设备文件
    #ifdef CONFIG_DEVFS_FS
       devfs_unregister(Devfs_Led_Raw);
       devfs_unregister(Devfs_Led_Dir);
    #endif
    //注销设备
       unregister_chrdev(Led_Major, DEVICE_NAME);
}
```

设备注册（初始化）和设备注销（反初始化）的登记

```
//向Linux系统记录设备初始化的函数名称
module_init(my_init);
//向Linux系统记录设备退出的函数名称
module_exit(my_exit);
```

驱动程序编译

- **Makefile**文件

```
OBJ=my_led.o
SOURCE=my_led.c
CC=gcc
COMP=-Wall -O2 -DMODULE -D_KERNEL_ -I /home/linux/linux-2.4.19-rmk7/include –c
$(OBJ):$(SOURCE)
        $(CC) $(COMP) $(SOURCE)
clean:
        rm $(OBJ)
```

- 运行**make** ,生成名为**my_led.o**的驱动程序

驱动程序加载

- 动态加载
  - 通过insmod等命令
  - 调试过程

模块动态加载

- 驱动程序模块插入内核

```
#insmod my_led.o
```

- 查看是否载入？载入成功会显示设备名my_led

```
#cat /proc/devices
```

- 从内核移除设备

```
#rmmod my_led
```

驱动测试应用程序

```
int main(void)
{
    int fd;
    char led_on = 0x01;
    fd = open("/dev/my_led", O_RDWR);  //打开led设备

    write( fd, &led_on, 1 );     //LED开
    close( fd );          //关闭设备文件
    return 0;
}
```