



《操作系统原理》

## 第8章 设备管理(输入/输出管理)

教师：苏曙光

华中科技大学网安学院

2023年10月-2024年01月

# 第8章 设备管理(输入/输出管理)

## ● 内容

- 设备管理概述
- 缓冲技术
- 设备分配
- SPOOL技术
- 设备驱动程序

## ● 重点

- 理解缓冲机制的作用
- 理解“设备是文件”的概念
- 掌握设备驱动程序开发



## 8.1 设备管理概念

# 设备类型和特征

- 按交互对象分类

- 人机交互：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：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

- 按外设特性分类

- 使用特征：存储设备、输入设备、输出设备
- 数据传输率：低速(键盘)、中速(打印机)、高速(网卡、磁盘)

# 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**。例：硬盘，磁盘

- 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

- (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

## ● 状态跟踪

- 记录设备的基本属性、状态、操作接口及进程访问信息。

### ◆ 设备控制块 (Device Control Block, DCB)

设备名
设备属性
命令转换表
在I/O总线上的设备地址
设备状态
当前用户进程指针
I/O请求队列指针

- 设备名
  - 设备的物理名
- 设备属性
  - 设备当前状态（一组属性）
- 命令转换表
  - 设备操作接口
    - ◆ 设备I/O例程（可为NULL）



# 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(缓冲区)



设备等待队列wait

PCB<sub>0</sub>

PCB<sub>1</sub>

PCB<sub>2</sub>

PCB<sub>2</sub>



# 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 )  
{  
    printf("Can'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",  
                    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

- 设备物理名

- I/O系统中实际安装的设备

- 物理名：设备端口、中断号或主/次设备号等

- 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逻辑设备，而不用考虑对应物理设备的特殊结构和操作方式。

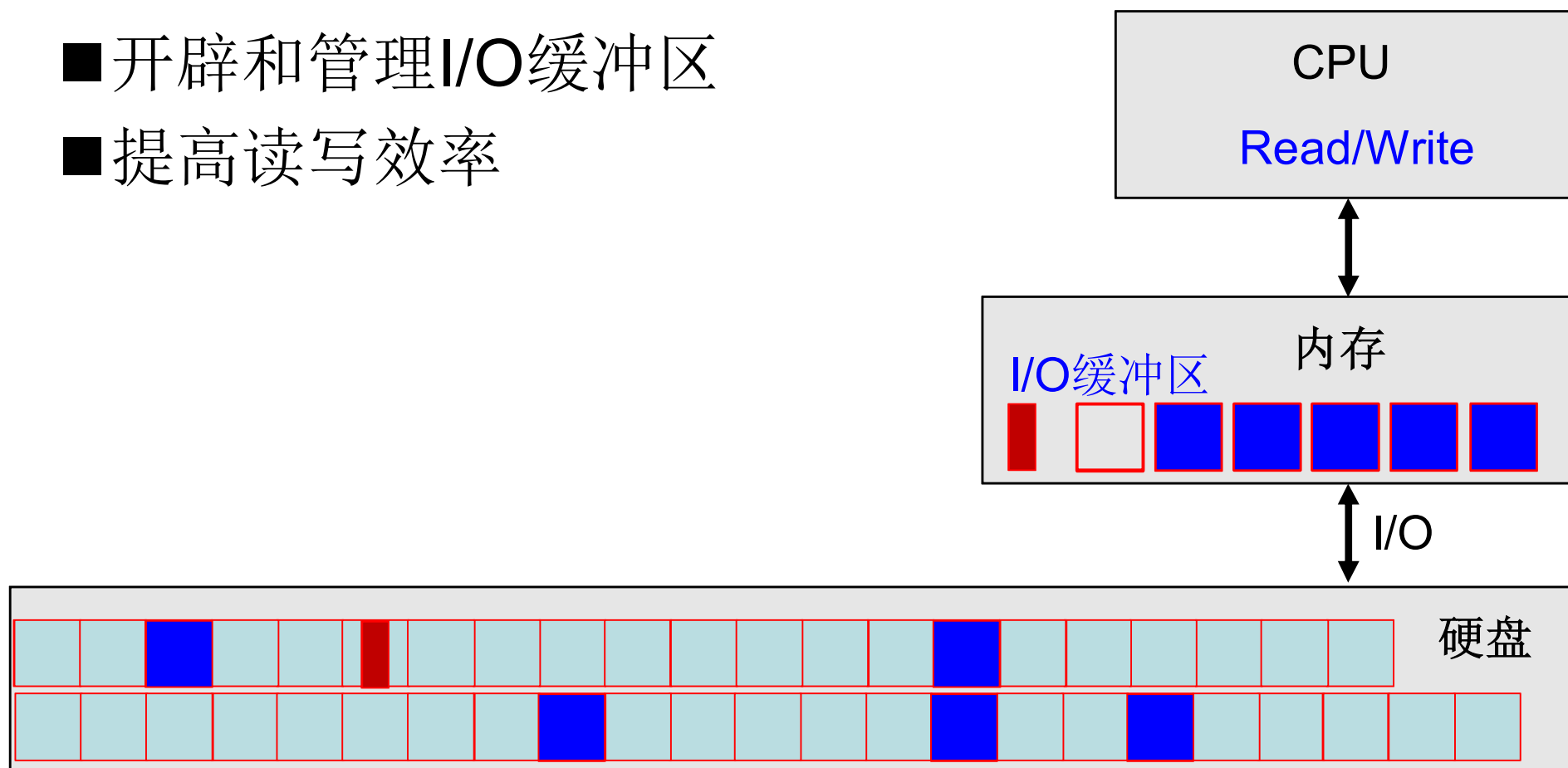
```
1      #include "stdio.h"
2
3      void main(void)
4      {
5          // write (0, "Hello World!\n")
6          printf("Hello World!\n");
7      }
8
```



# 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



# 设备管理功能>设备驱动

## ● 设备驱动

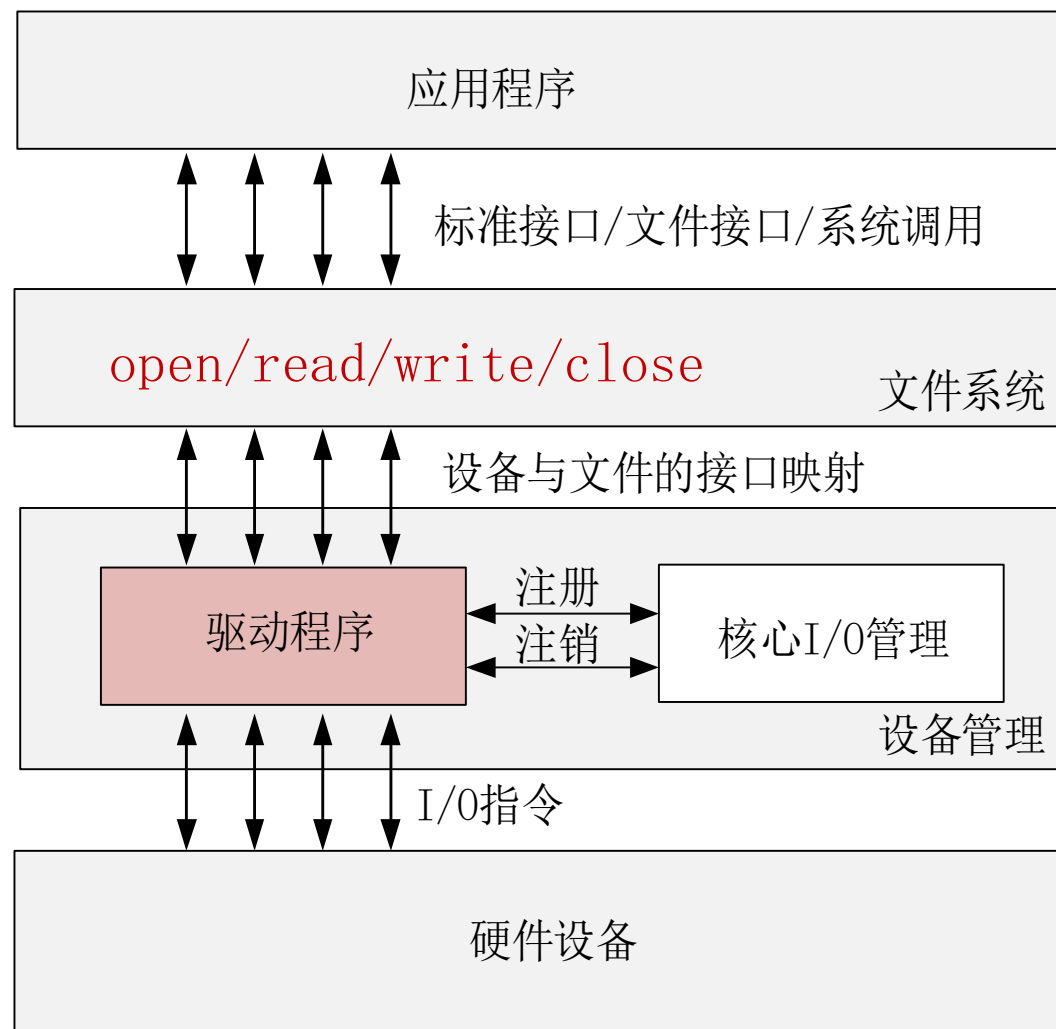
■ 对物理设备进行I/O操作（IN/OUT指令）。

■ 把应用对设备的读/写请求转换为对设备I/O操作。

■ 应用读写请求采用文件接口

◆ open/read/write/close

◆ 设备是文件。



# 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 缓冲技术

# 缓冲技术

- 缓冲作用

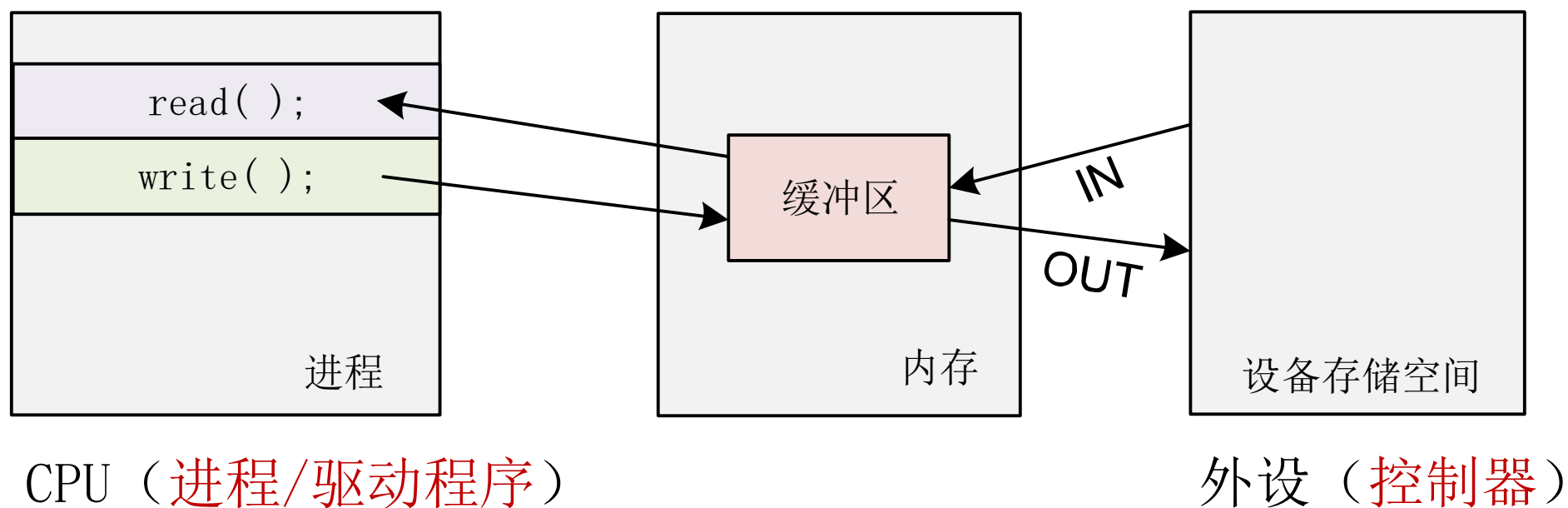
- 1) 连接不同数据传输速度的设备
- 2) 协调数据记录大小的不一致
- 3) 正确执行应用程序的语义拷贝

# 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

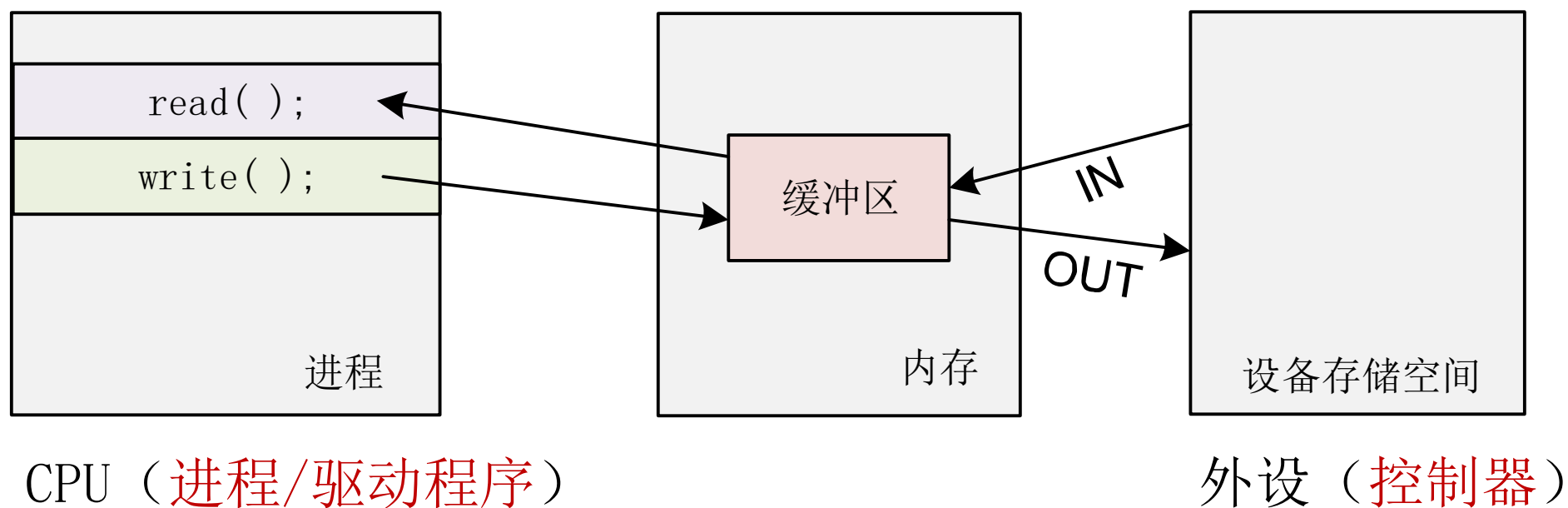


# 缓冲作用

## ● 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）：设备（字节）

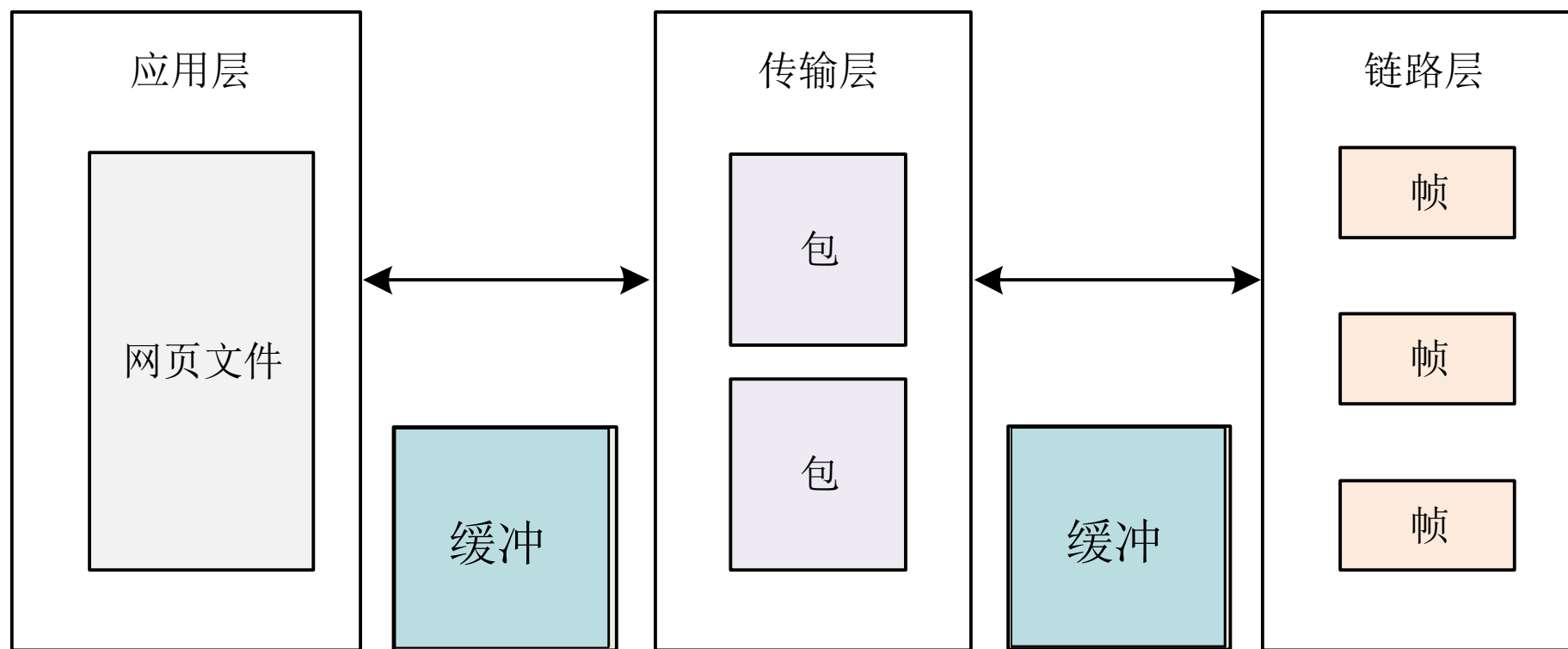


# 缓冲作用

## ● 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：不同网络层之间的数据记录：帧、包、文件



# 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

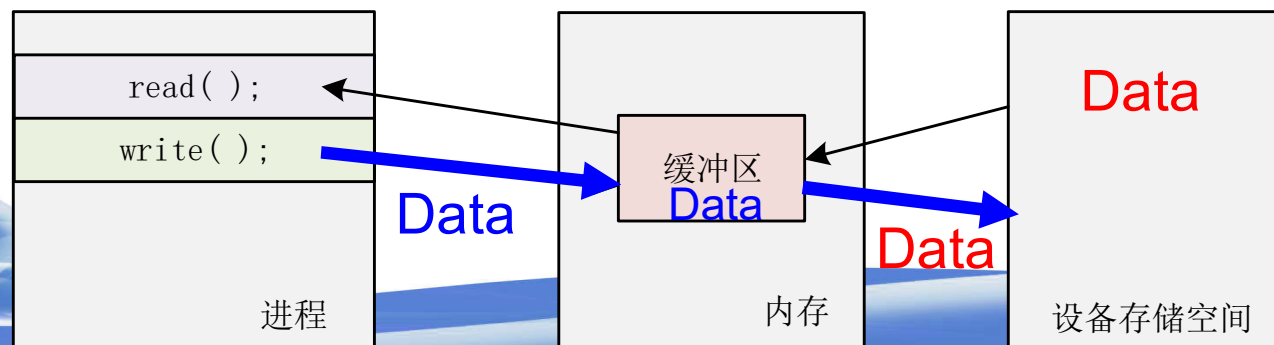
□ 应用等待内核写完磁盘再返回。(实时性差)

◆ 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

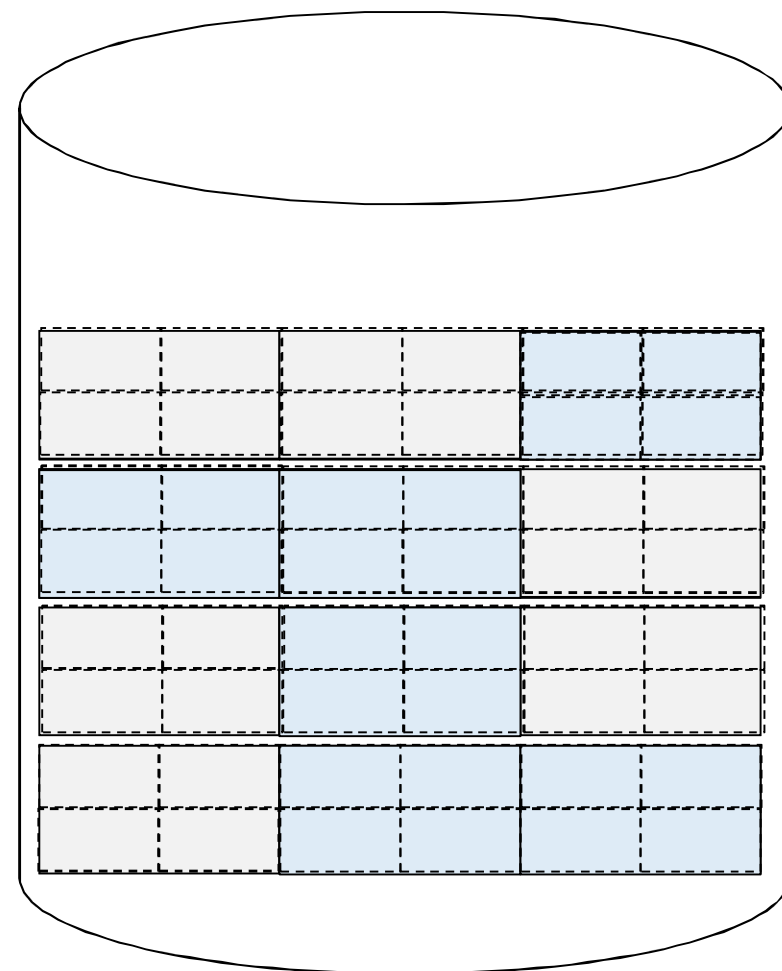
- 块 =  $2^n \times$  扇区

- Linux块 = 1KB ( $n=1$ )

- Linux缓冲机制

- 内存开辟高速缓冲区

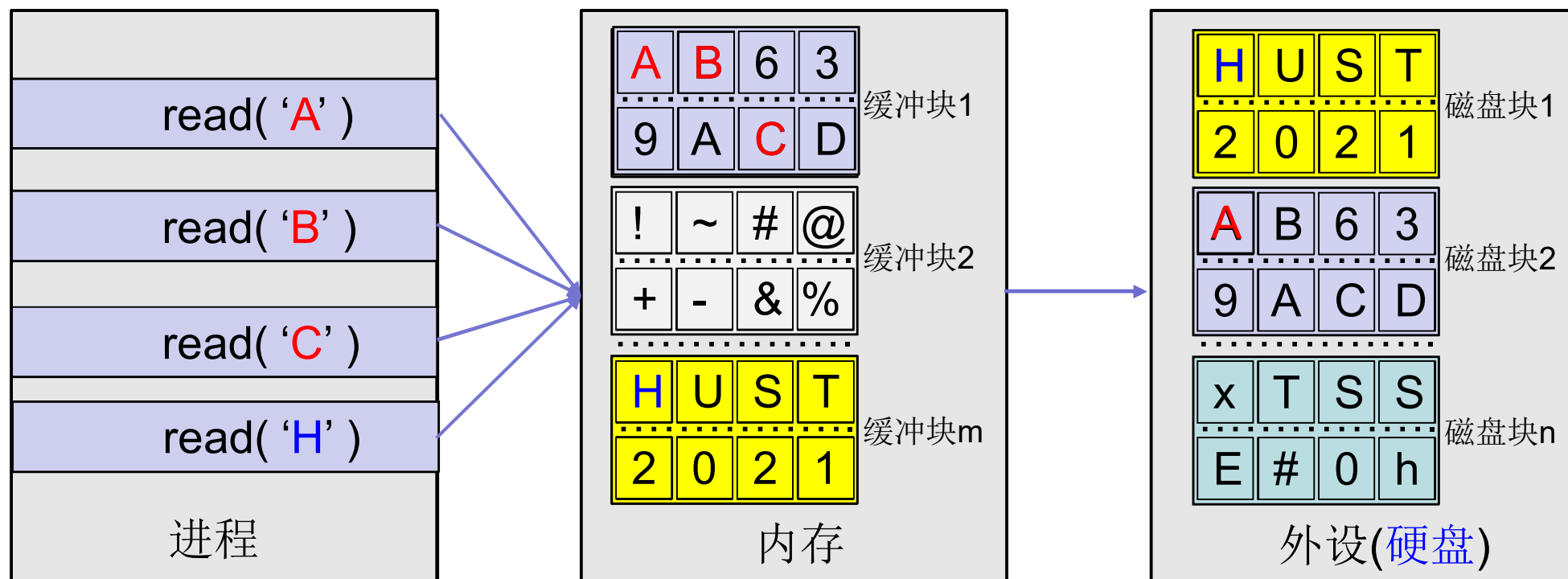
- 提前读/延后写



# 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。



$$m \ll n$$



# Linux缓冲机制应用（块设备）

- 延后写

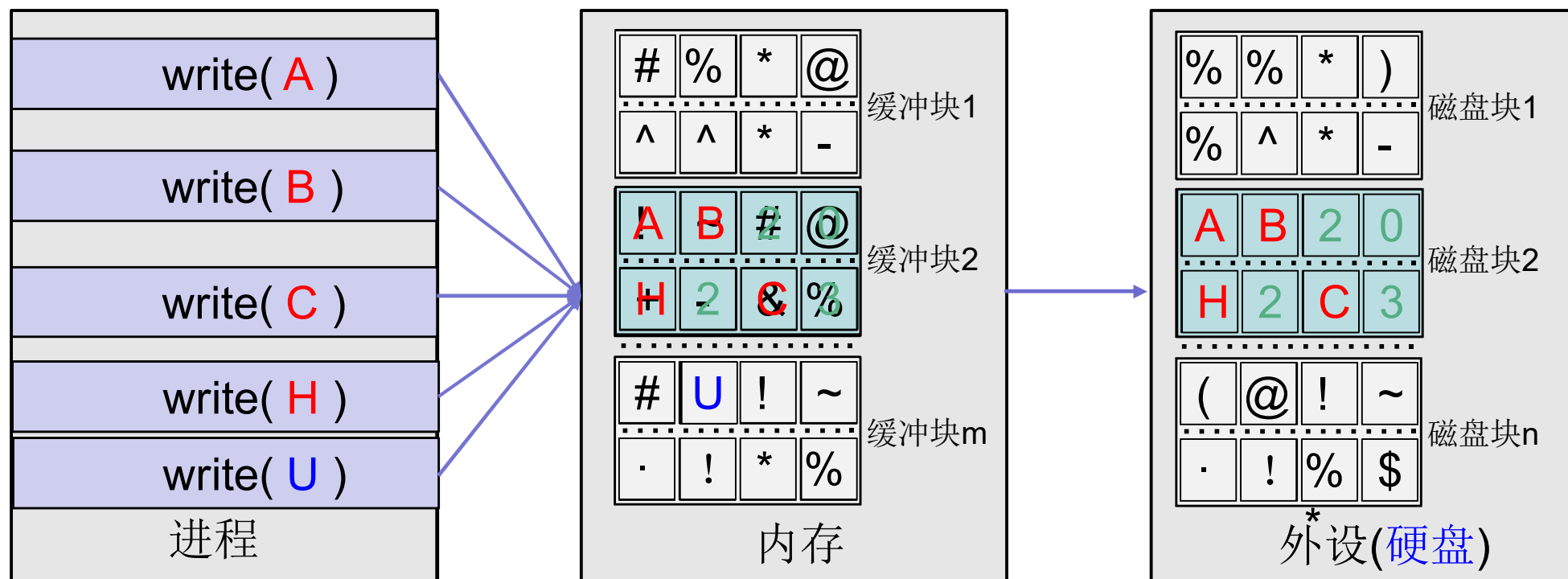
- 进程写时，数据先存在缓冲区，等到特定事件发生或足够时间后（已延迟），再启动外设完成写入。



# 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）

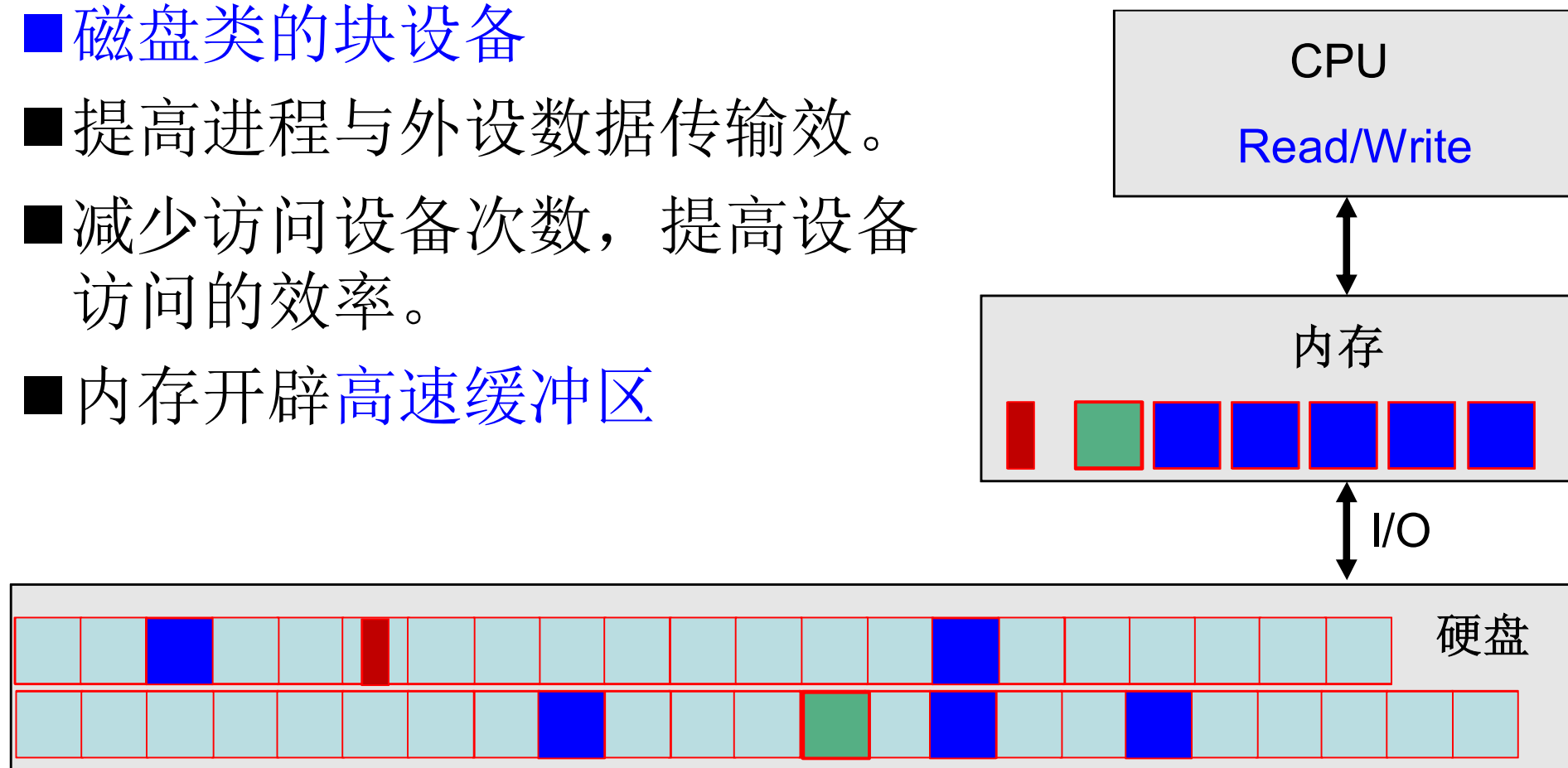
## ● 提前读与延后写

### ■ 磁盘类的块设备

■ 提高进程与外设数据传输效。

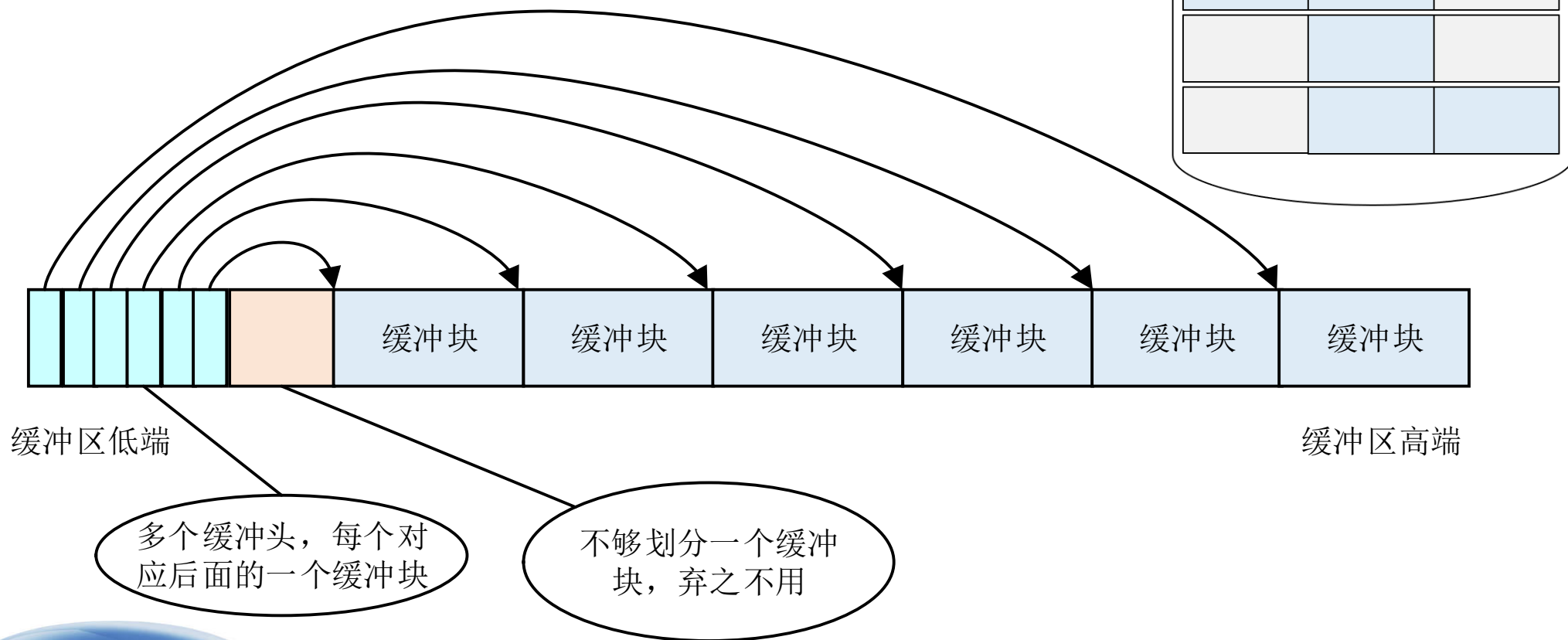
■ 减少访问设备次数，提高设备访问的效率。

■ 内存开辟高速缓冲区



# Linux缓冲机制应用（块设备）

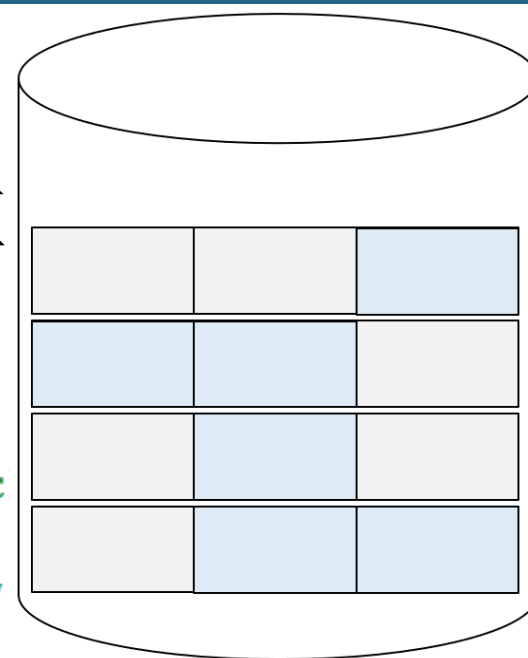
- 高速缓冲区(内存区)
  - 按块分为缓冲块(数据块)，与磁盘块对应
  - 缓冲头（buffer\_head）：描述缓冲块



# 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 block */  
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: 指向缓冲块对应的数据区
    - ◆ b\_blocknr: 设备中的块号
    - ◆ b\_dev: 设备号
    - ◆ b\_lock: 表示该缓冲块是否已被锁定
    - ◆ b\_count: 缓冲块被多少个进程引用
    - ◆ b\_dirt: 延迟写字段，即脏位字段
    - ◆ b\_uptodate: 数据有效位字段
    - ◆ b\_wait: 指向访问缓冲块的等待队列

# Linux缓冲机制应用（块设备）

- 进程读写设备数据

- 进程read/write → 文件访问请求 → 块读取bread( )

- 块读取函数bread(设备号,块号)

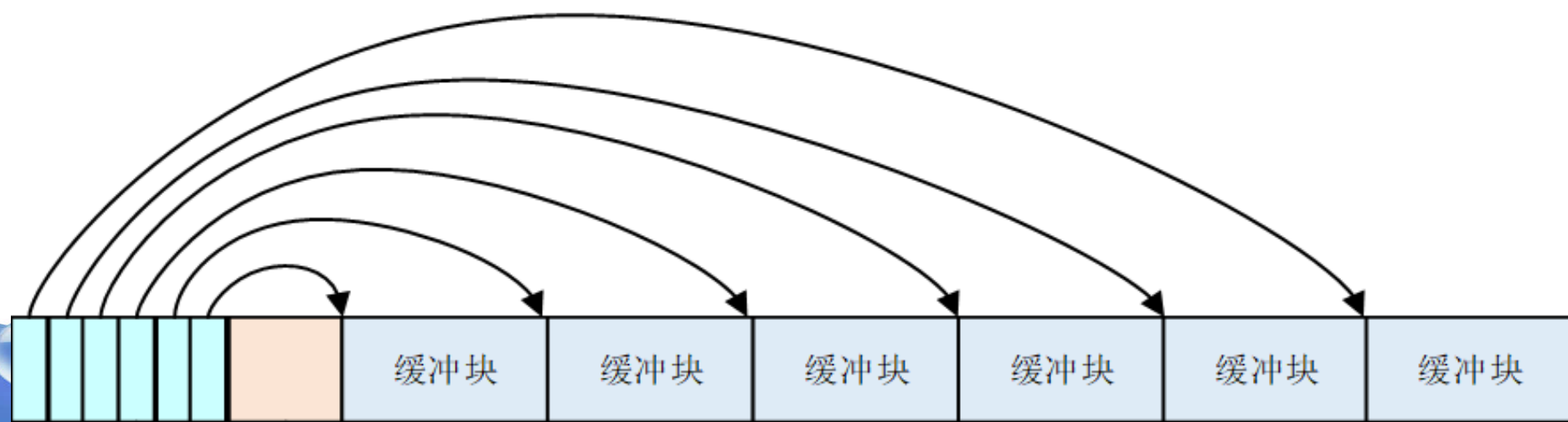
- 以(设备号, 块号)为索引搜索高速缓冲区，查找对应的缓冲块

- ◆ 若找到，直接读回

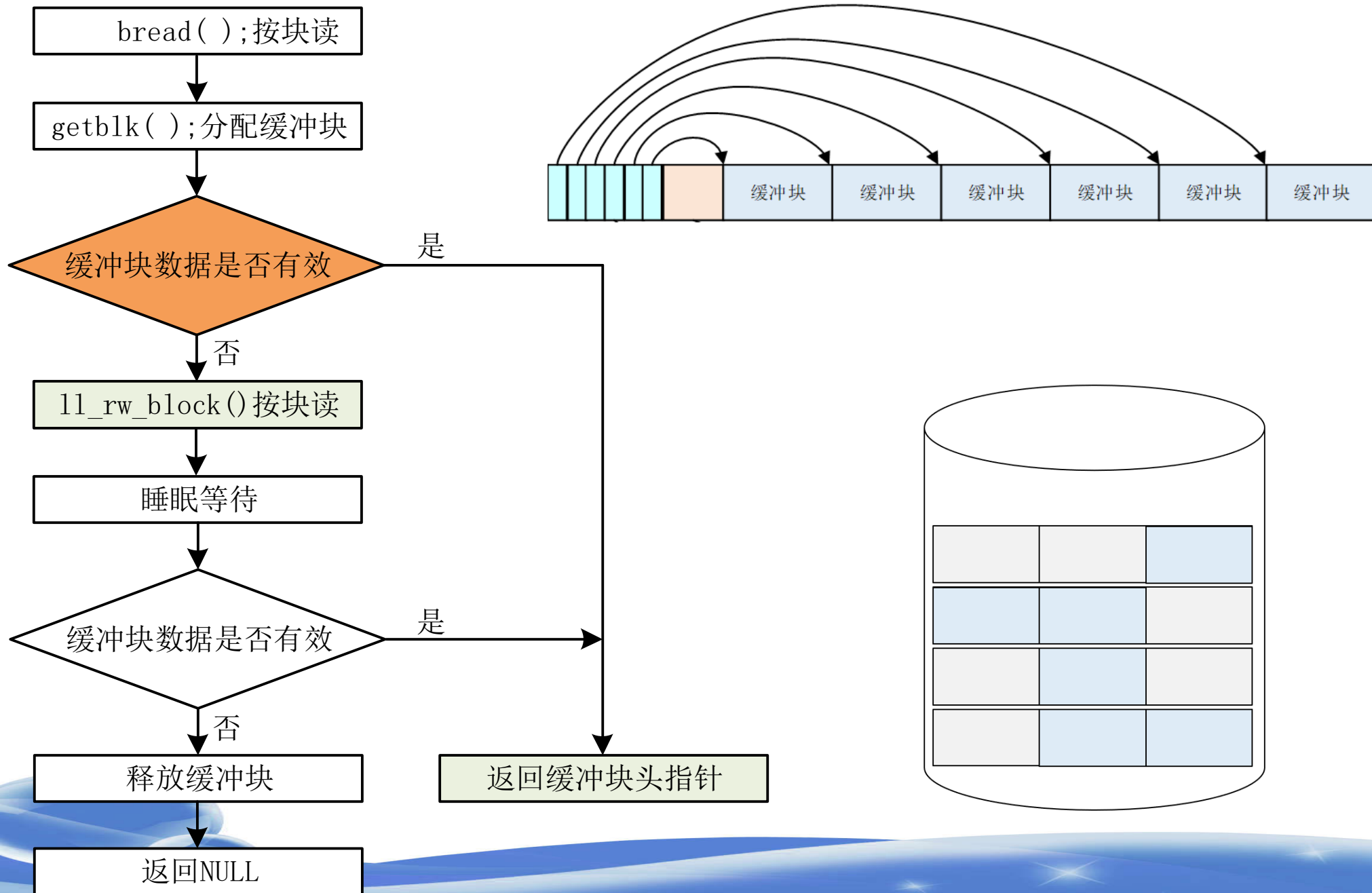
- ◆ 若没有找到

- 分配一个新缓冲块

- 调用ll\_rw\_block( )读相应磁盘块到新分配的缓冲块



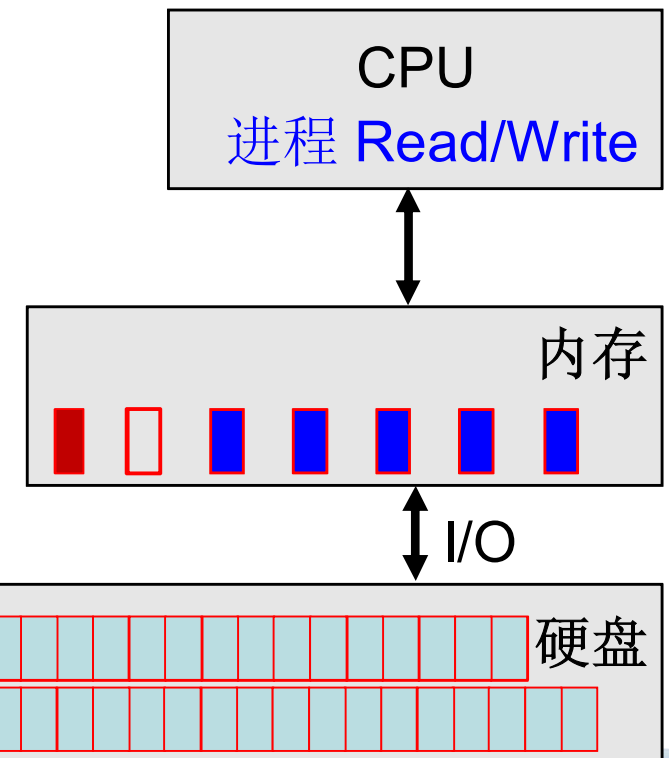
# bread(设备号,块号)函数





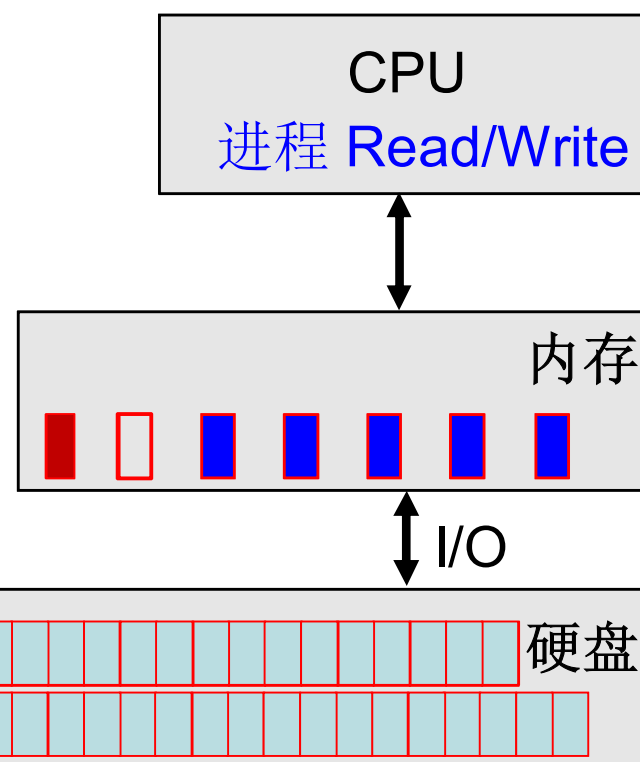
# bread(设备号,块号)函数

```
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 }
```



# file\_read(m\_inode\*, file\*)函数

```
1 int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
2 {
3     int left, chars, nr;
4     struct buffer_head * bh;
5
6     if ((left=count)<=0)
7         return 0;
8     while (left) {
9         if (nr = bmap(inode, (filp->f_pos)/BLOCK_SIZE)) {
10             if (!(bh=bread(inode->i_dev,nr)))
11                 break;
12         } else
13             bh = NULL;
14         nr = filp->f_pos % BLOCK_SIZE;
15         chars = MIN( BLOCK_SIZE-nr , left );
16         filp->f_pos += chars;
17         left -= chars;
18         if (bh) {
19             char * p = nr + bh->b_data;
20             while (chars-->0)
21                 put_fs_byte(*(p++),buf++);
22             brelse(bh);
23         } else {
24             while (chars-->0)
25                 put_fs_byte(0,buf++);
26         }
27     }
28     inode->i_atime = CURRENT_TIME;
29     return (count-left)?(count-left):-ERROR;
30 }
```



# 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】

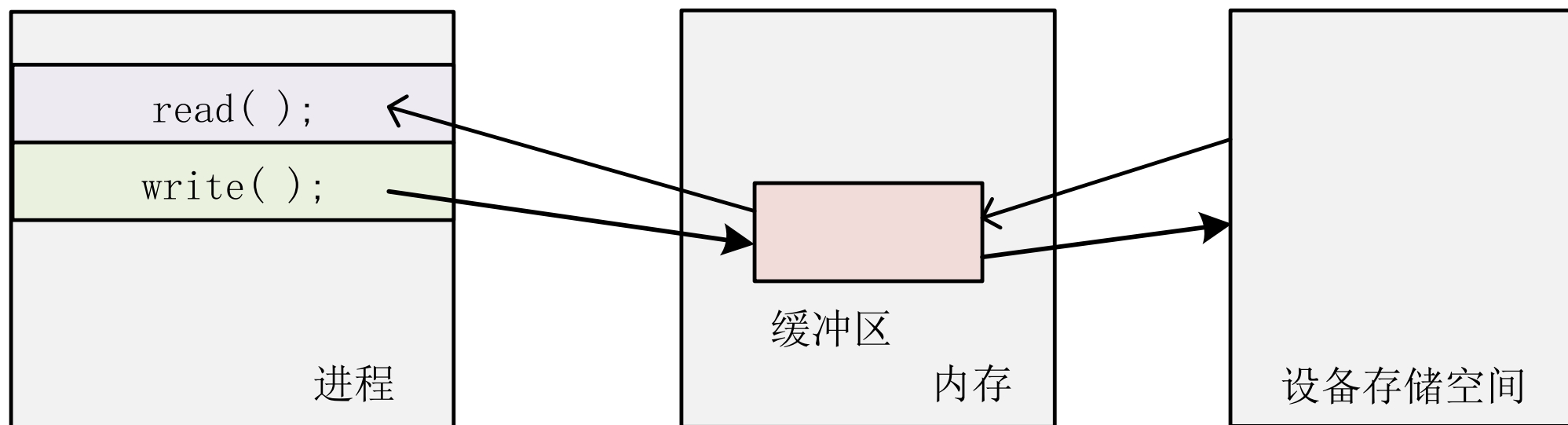
# 缓冲的实现

- 单缓冲
- 双缓冲
- 环形缓冲
- 缓冲池

# 缓冲的实现>单缓冲

## ● 单缓冲

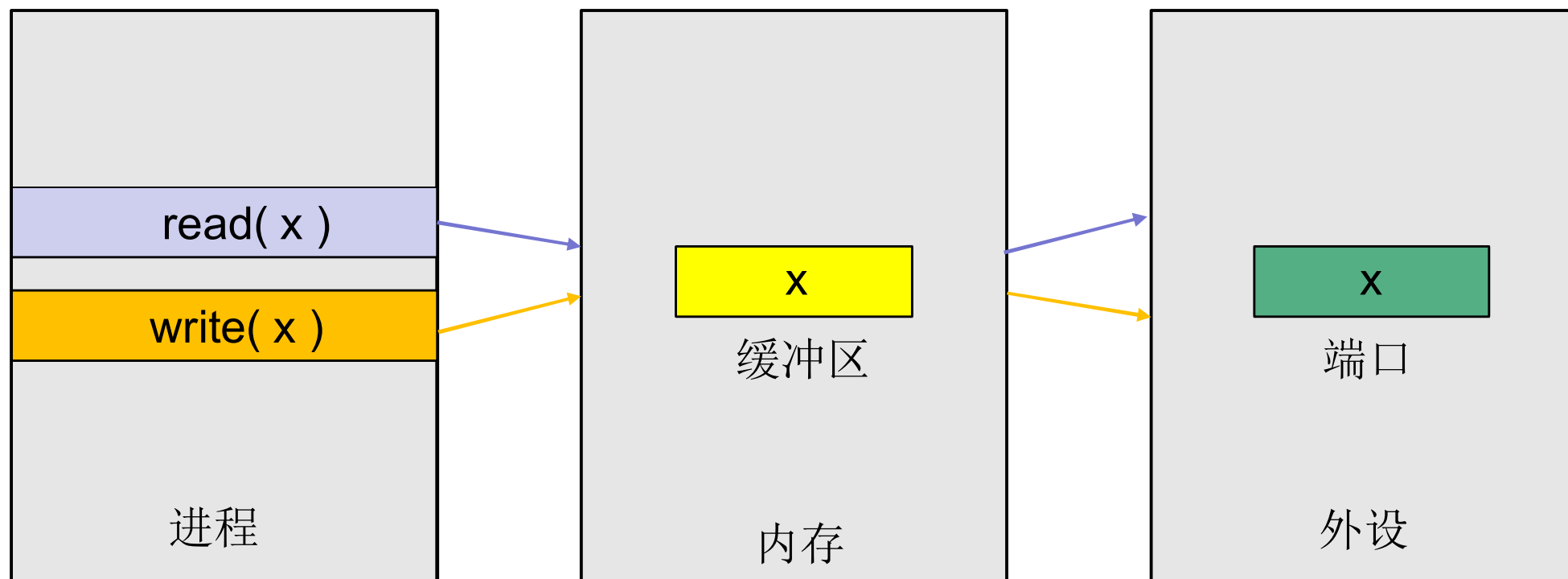
■ 缓冲区仅有1个单元



# 缓冲的实现>单缓冲

## ● 单缓冲

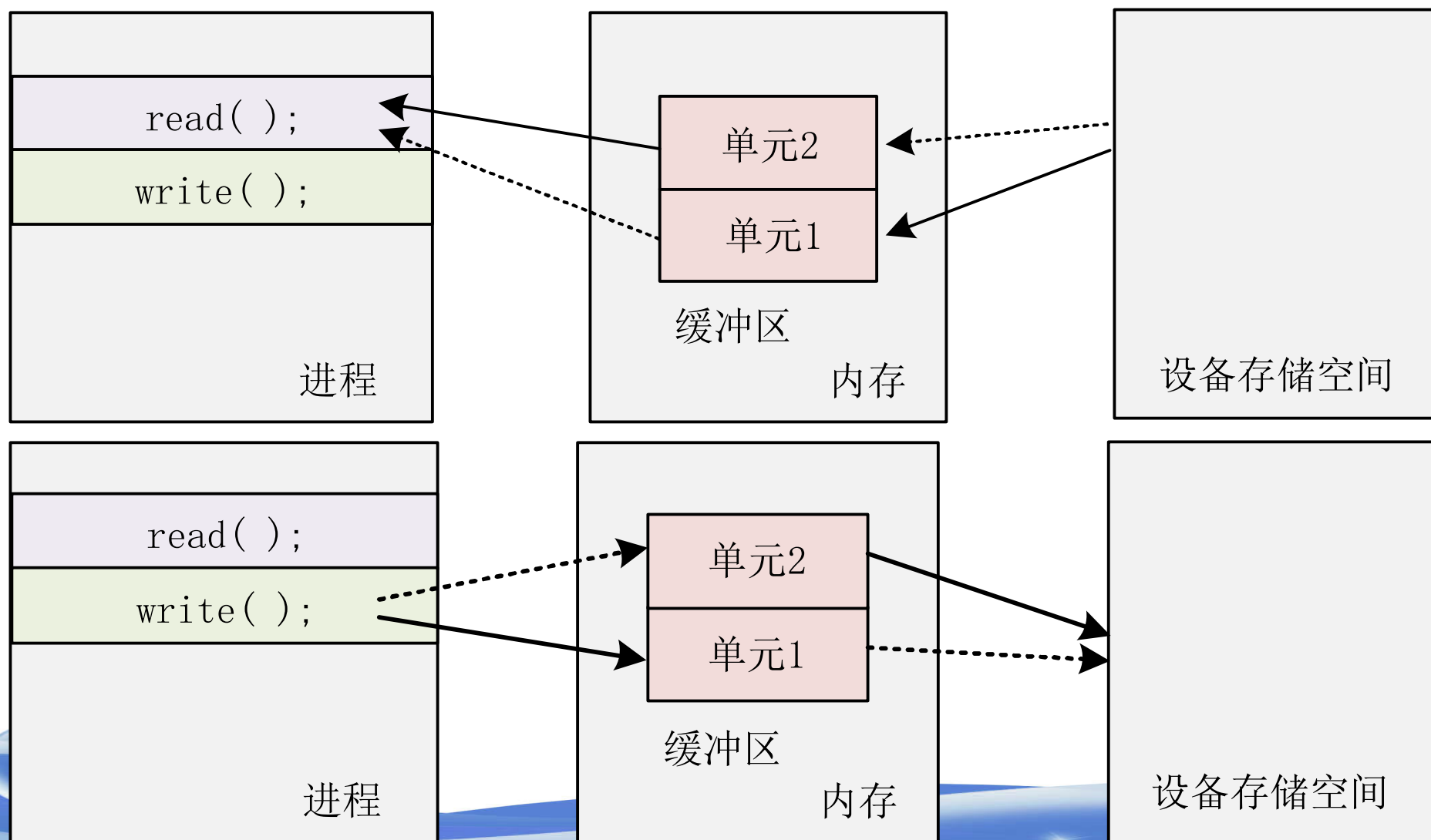
■ 缓冲区仅有1个单元



# 缓冲的实现>双缓冲

## ● 双缓冲

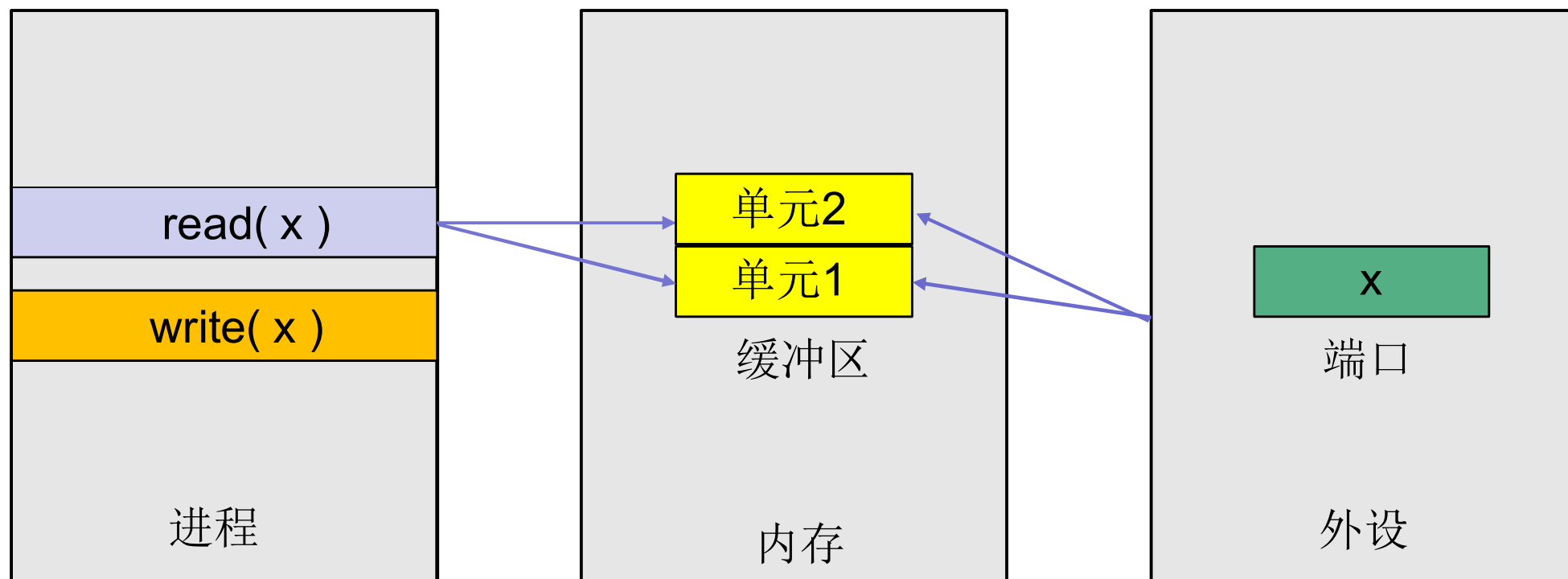
### ■ 缓冲区有2个单元



# 缓冲的实现>双缓冲

- 双缓冲

- 缓冲区有2个单元

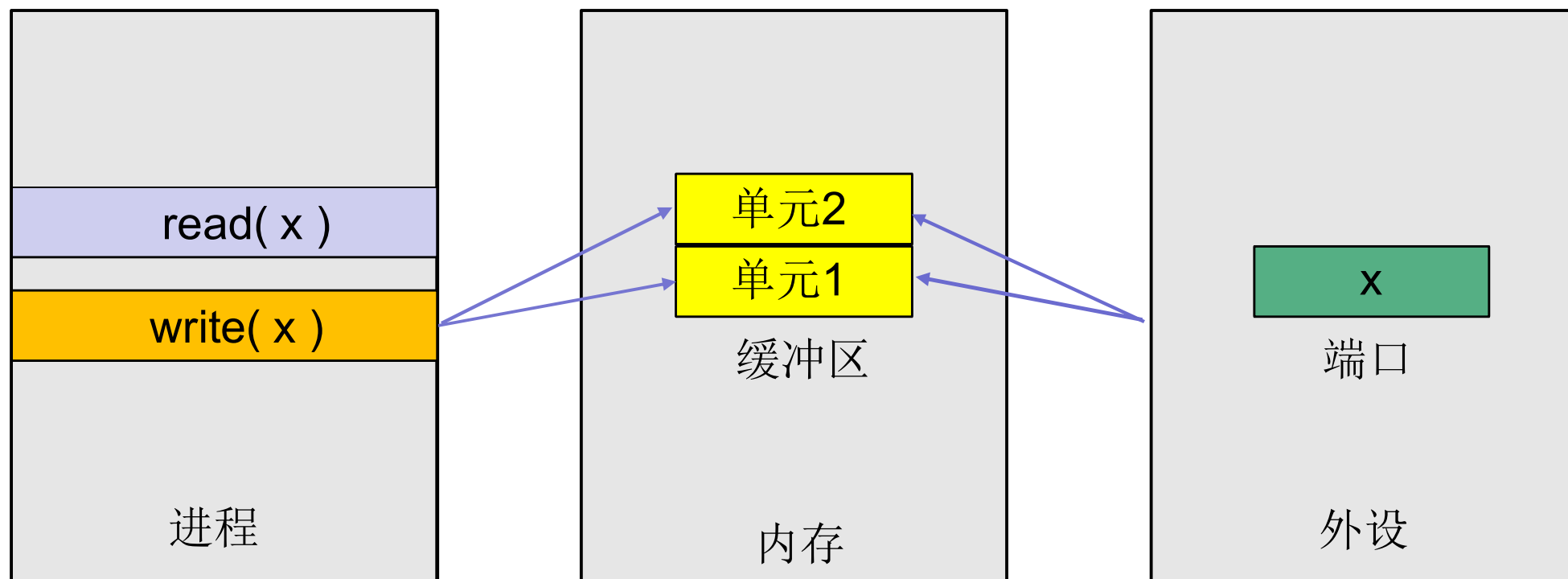




# 缓冲的实现>双缓冲

- 双缓冲

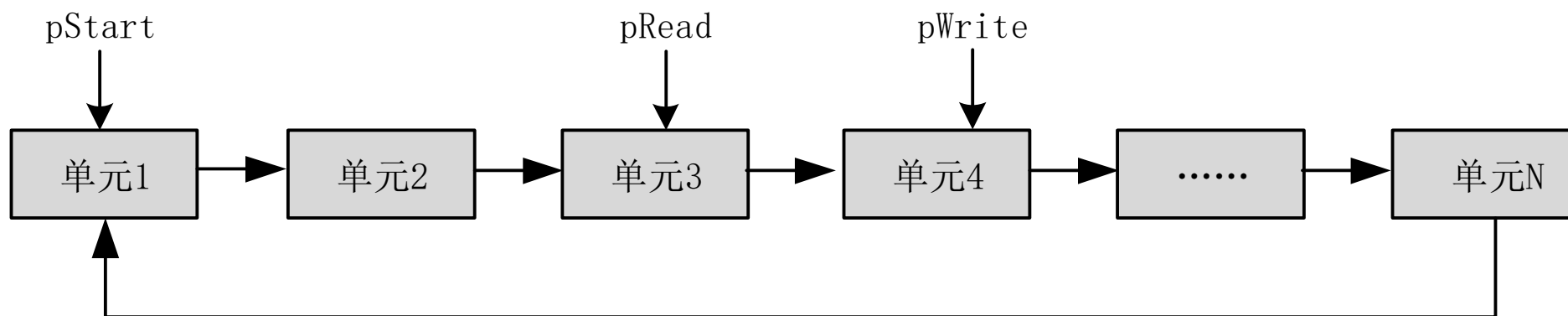
- 缓冲区有2个单元



# 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





## 8.3 设备驱动程序

# Linux字符设备驱动示例

## ● 例：应用程序（读/写设备的状态）

```
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（读/写设备的状态）

```
13 static int chr_open(struct inode * pinode, struct file *pfile)
14 { // chr_open(): 打开设备 //打开文件 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 { // chr_read(): 读设备的状态 //读文件 read( fd )
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 { // chr_write(): 写设备的状态 //写文件 write( fd )
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 inode * pinode, struct file *pfile)
31 { // chr_release(): 关闭设备 //关闭文件 close( fd )
32     MOD_DEC_USE_COUNT;
33     return 0;
34 }
```



# Linux字符设备驱动示例

## ● 例：驱动程序（读/写设备的状态）

### ■ 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       = chr_release,
6     .open          = chr_open,
7     .unlocked_ioctl = chr_ioctl,
8 };
```

```
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（读/写设备的状态）

### ■ 实现设备的注册函数和注销函数

```
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 }
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 }
50 module_init(DevInit);
51 module_exit(DevExit);
```



# Linux字符设备驱动示例

- 例：驱动程序（读/写设备的状态）

- 设备注册

- ◆ 将用户定义的设备加入到系统的设备数组

```
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 }
```

```
int register_chrdev(
    int major,
    char *name,
    struct file_operations *fops);
```

# Linux字符设备驱动示例

- 例：驱动程序（读/写设备的状态）

- 设备注销

- ◆ 释放设备，将设备从系统的设备数组删除

```
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 }
```

- unregister\_chrdev( )

- unregister\_blkdev( )

# Linux字符设备驱动示例

- 例：驱动程序（读/写设备的状态）
  - 实现设备的注册函数和注销函数

```
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 }
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 }
50 module_init(DevInit);
51 module_exit(DevExit);
```

# 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
```

## ● 安装/删除驱动程序

# insmod RWDevState.ko

# rmmod RWDevState

## ● 创建设备文件

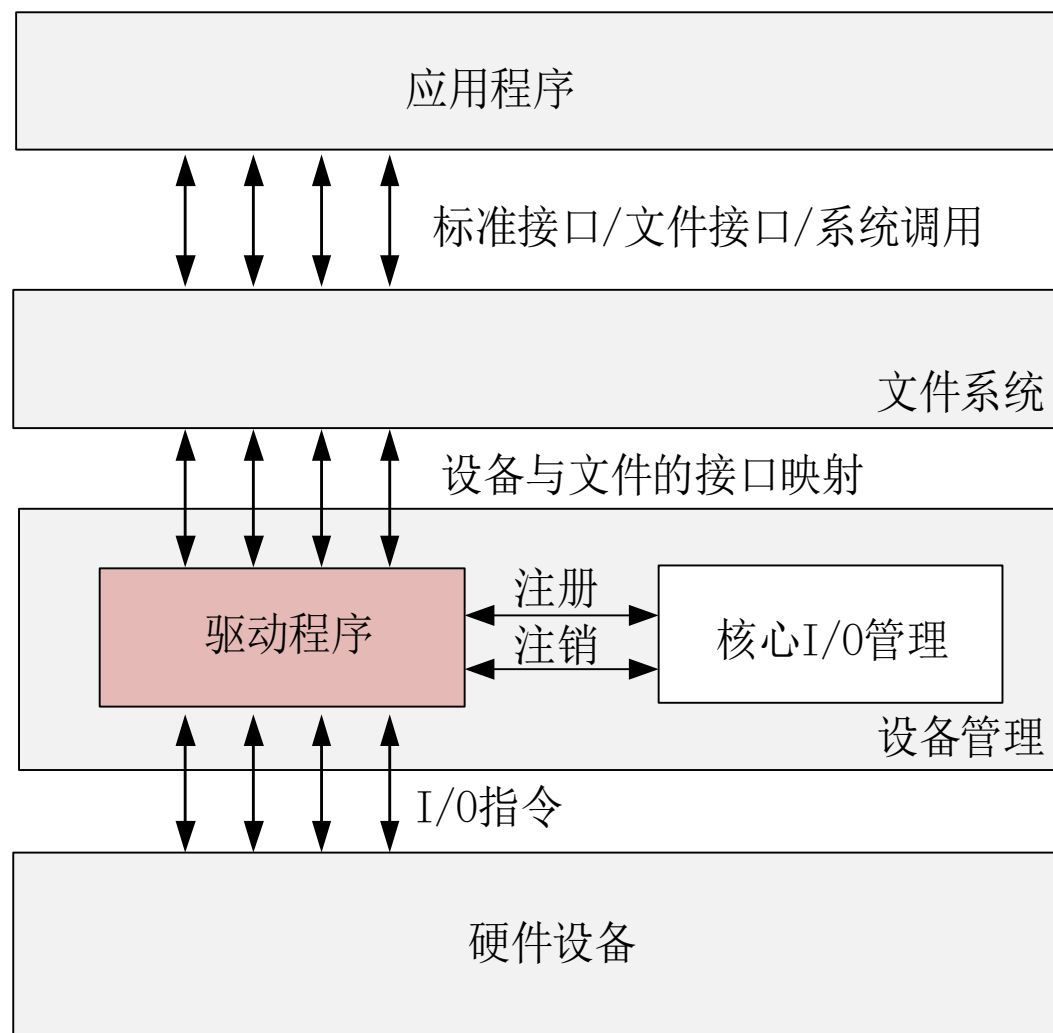
mknod /dev/RWDevState c 252 0

```
4 //打开设备，设备文件名是： /dev/RWDevState
5 fd = open("/dev/RWDevState", O_RDWR);
```

# 驱动程序在系统中的地位

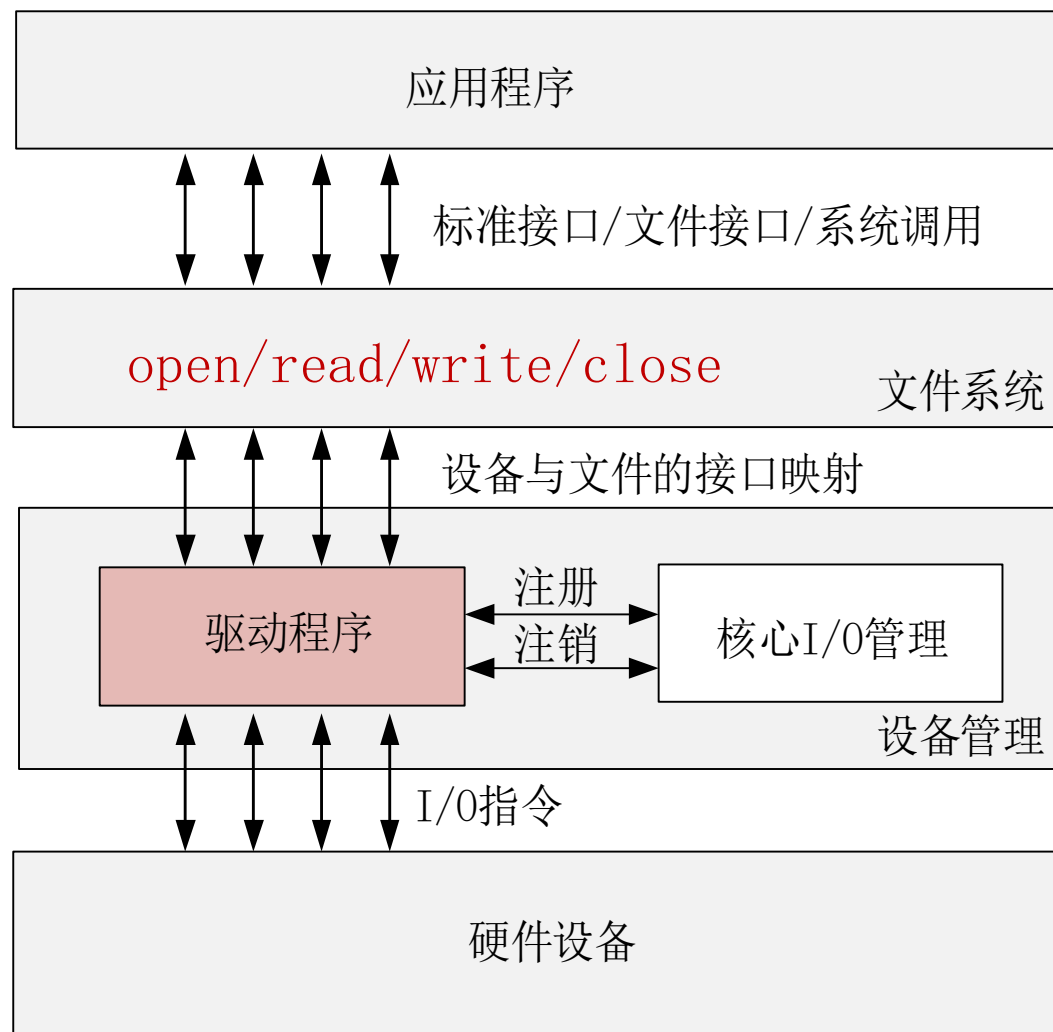
## ● 驱动程序的基本接口

- 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(命令)

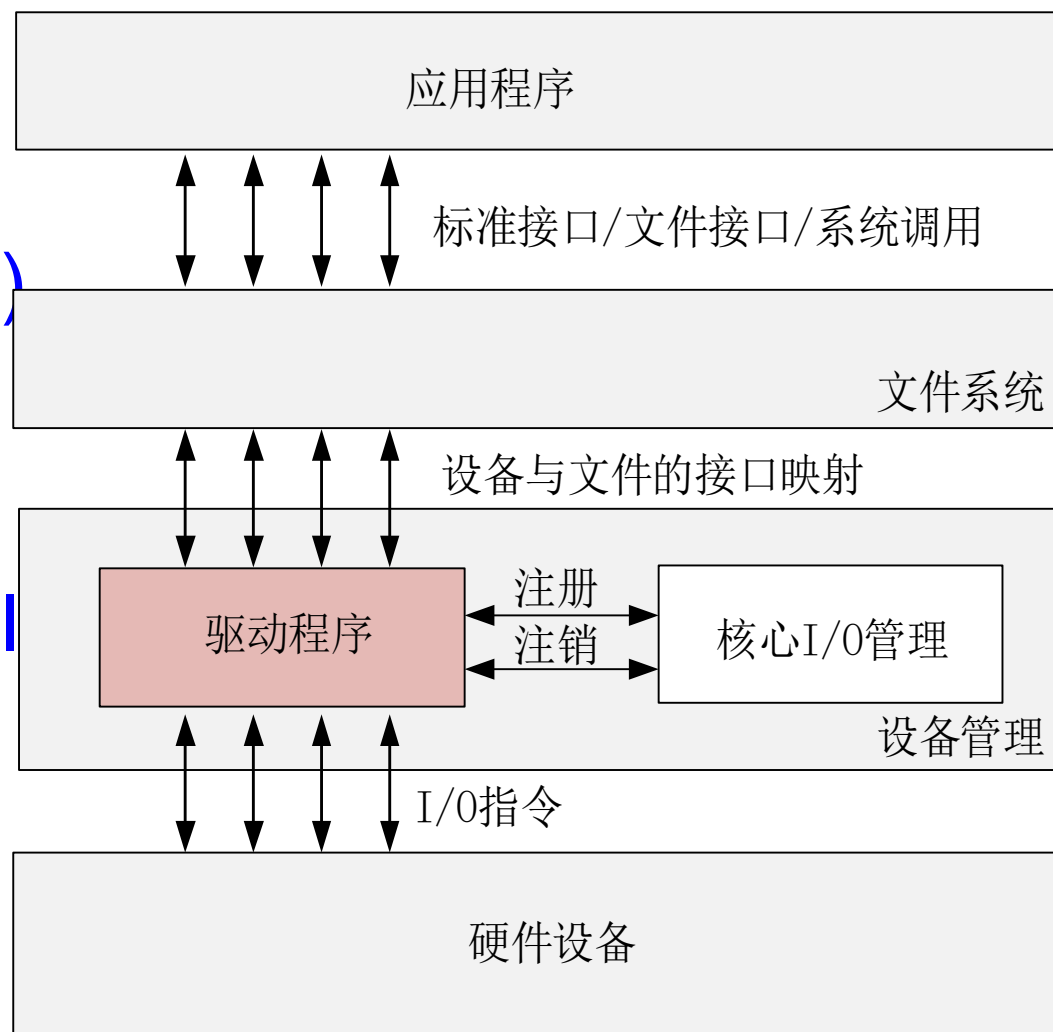
- module\_init() (API)

- 注销函数

- ◆ rmmod (命令)

- module\_exit() (API)

- 必需的数据结构



# 驱动程序在系统中的地位

## ● 面向设备的接口

### ■ 实现设备的端口操作

◆ 无条件传送

◆ 查询传送

◆ 中断传送

◆ DMA传送

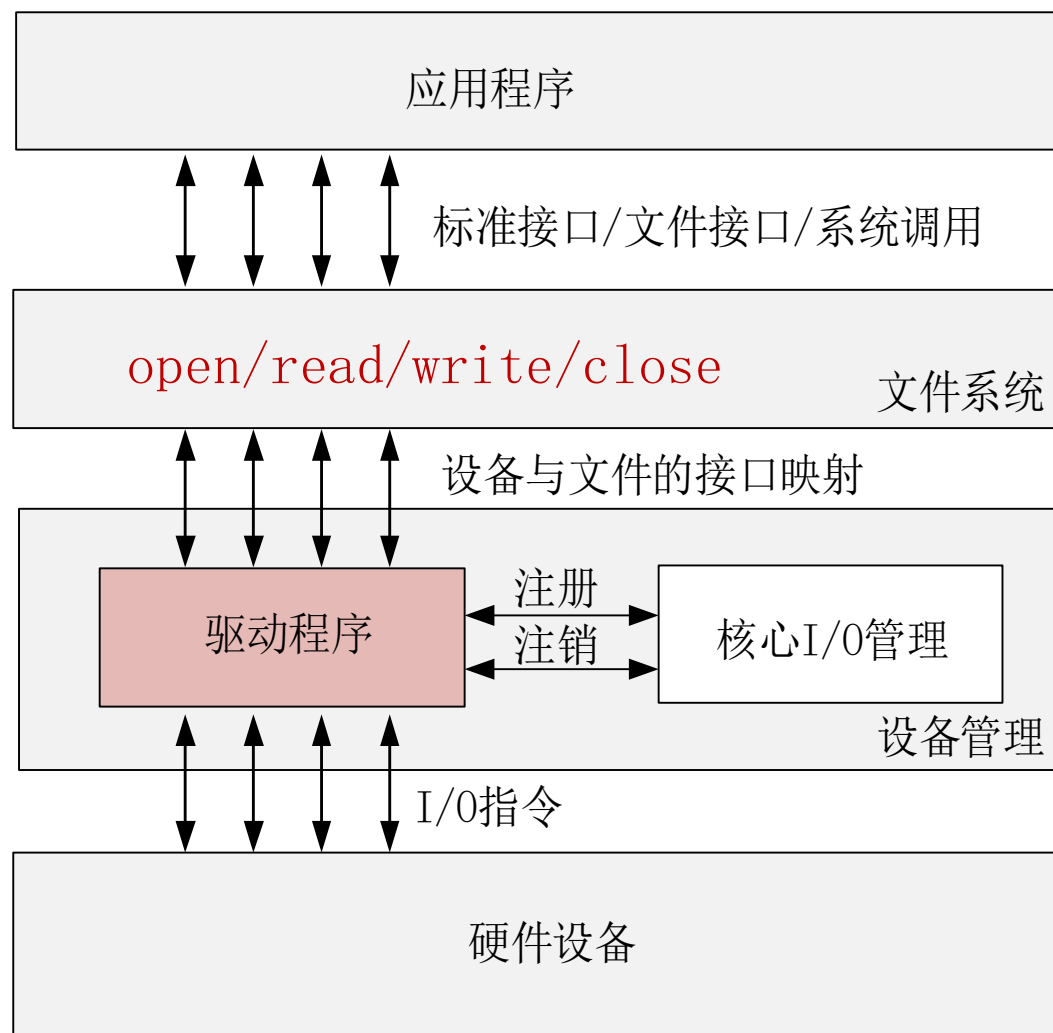
### ■ 例：

◆ 打开设备 `chr_open()`

◆ 读设备 `chr_read()`

◆ 写设备 `chr_write()`

◆ 关闭设备 `chr_release()`





# Linux设备的分类

- 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**。例：硬盘，磁盘。

- 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**。

# 驱动程序工作在核态

- 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( )

- ◆ put\_user( )

- ◆ copy\_from\_user ( )

- ◆ copy\_to\_user ( )

# 设备文件

- 设备文件

- 硬件设备作为文件看待
- 使用文件操作接口来完成设备的打开、关闭、读写和I/O控制等操作。
- 仅字符设备和块设备通过设备文件访问
  - ◆ 创建设备文件： `mknod`

# 设备文件 ( ls -l /dev )

```
susg : bash
File Edit View Bookmarks Settings Help
[susg@localhost ~]$ ls -l /dev/
total 0
crw-----. 1 root root    10, 235 5月  1 17:06 autofs
drwxr-xr-x. 2 root root    280 5月  1 17:06 block
drwxr-xr-x. 2 root root     80 5月  2 2017 bsg
crw-----. 1 root root    10, 234 5月  1 17:06 btrfs-control
drwxr-xr-x. 3 root root     60 5月  2 2017 bus
lrwxrwxrwx. 1 root root      3 5月  1 17:06 cdrom -> sr0
drwxr-xr-x. 2 root root   3400 5月  1 17:06 char
crw-----. 1 root root     5,   1 5月  1 17:06 console
lrwxrwxrwx. 1 root root    11 5月  2 2017 core -> /proc/kcore
drwxr-xr-x. 6 root root    140 5月  2 2017 cpu
crw-----. 1 root root    10,  62 5月  1 17:06 cpu_dma_latency
crw-----. 1 root root    10, 203 5月  1 17:06 cuse
drwxr-xr-x. 4 root root     80 5月  2 2017 disk
brw-rw----. 1 root disk   253,   0 5月  1 17:06 dm-0
brw-rw----. 1 root disk   253,   1 5月  1 17:06 dm-1
brw-rw----. 1 root disk   253,   2 5月  1 17:06 dm-2
brw-rw----. 1 root disk   253,   3 5月  1 17:06 dm-3
drwxr-xr-x. 2 root root    100 5月  2 2017 dri
crw-rw----. 1 root video   29,   0 5月  1 17:06 fb0
lrwxrwxrwx. 1 root root     13 5月  2 2017 fd -> /proc/self/fd
```

**c:** 字符设备  
**b:** 块设备  
**5:** 主设备号  
**1:** 次设备号  
设备文件

# 设备文件

- 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

- 驱动注册过程发生变化

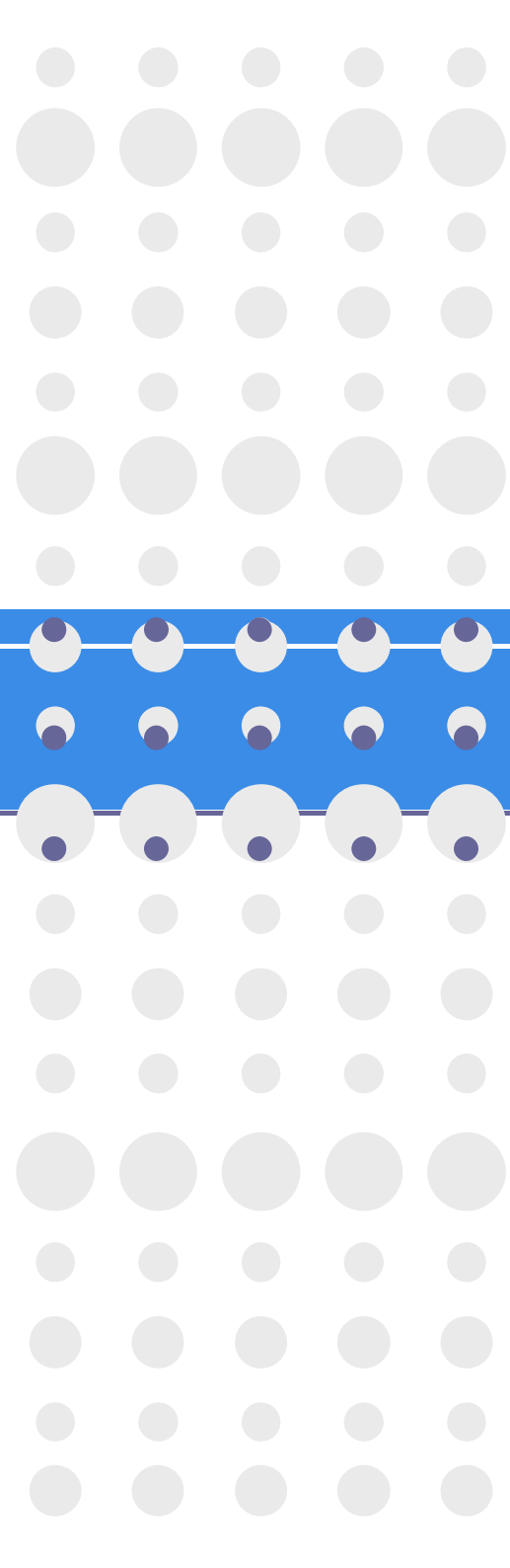
//V2.4 字符设备注册

```
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);
```





## 8.4 I/O控制（浏览一下即可， 考试不做要求）

## ● 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”；

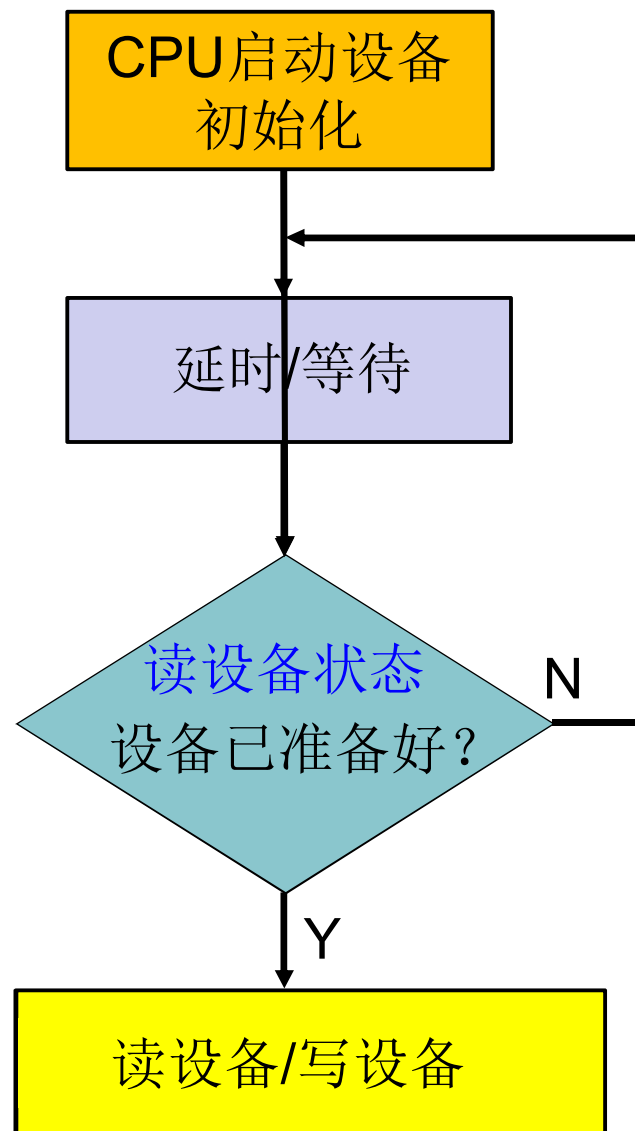
◆ 输出时：外设“准备好”接收。

## ● 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



# 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。

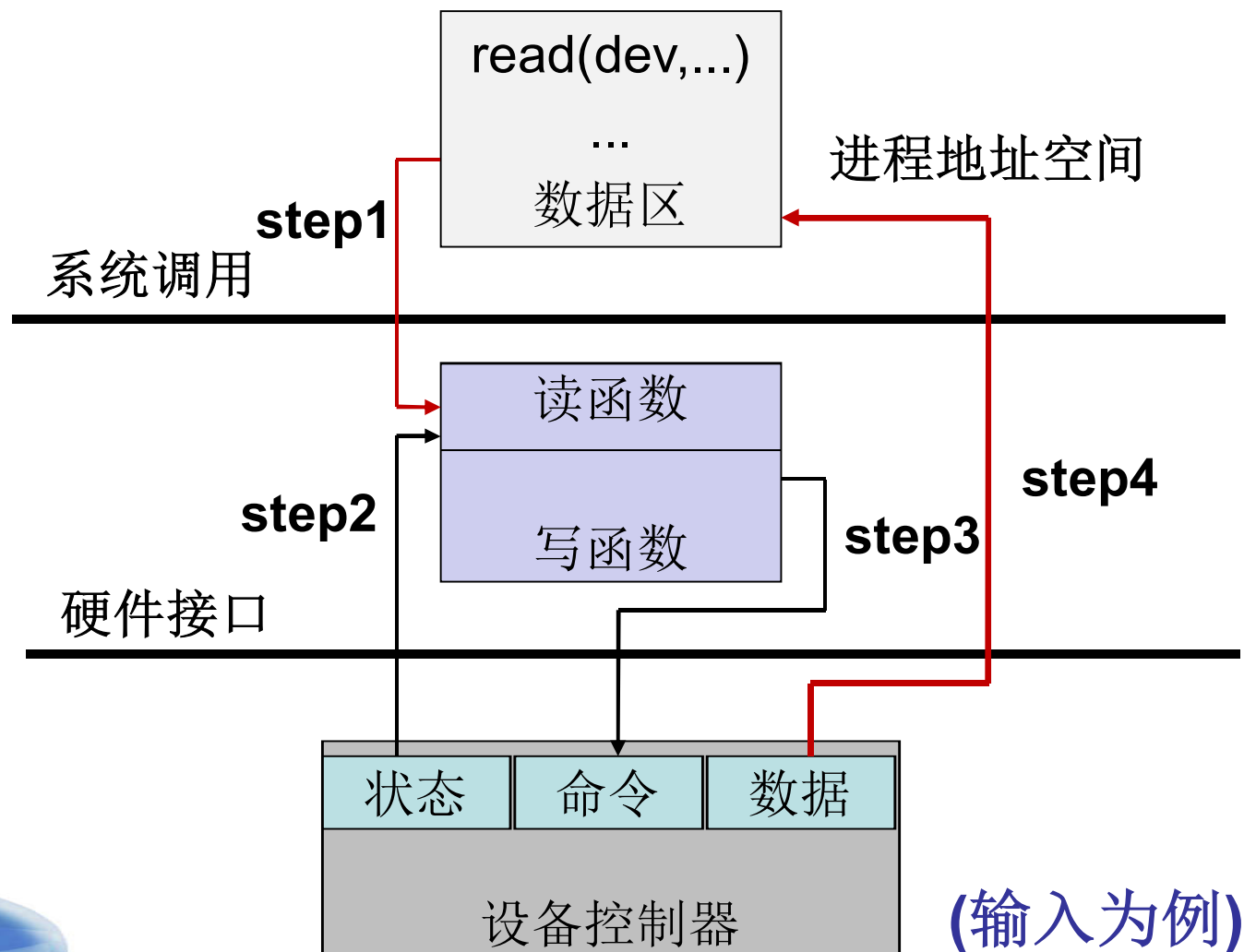
```
1 POLL:
2     IN     AL,     PORT_State ;读状态端口: PORT_State
3     TEST   AL,     80H       ;80H是掩码检查READY位是否为1
4     JZ     POLL      ;未准备好, 转POLL
5     IN     AL,     PORT_Data ;读数据端口: PORT_Data

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

# 查询方式（异步传送）（不要求）

## ● 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。



# 中断方式

- 工作原理

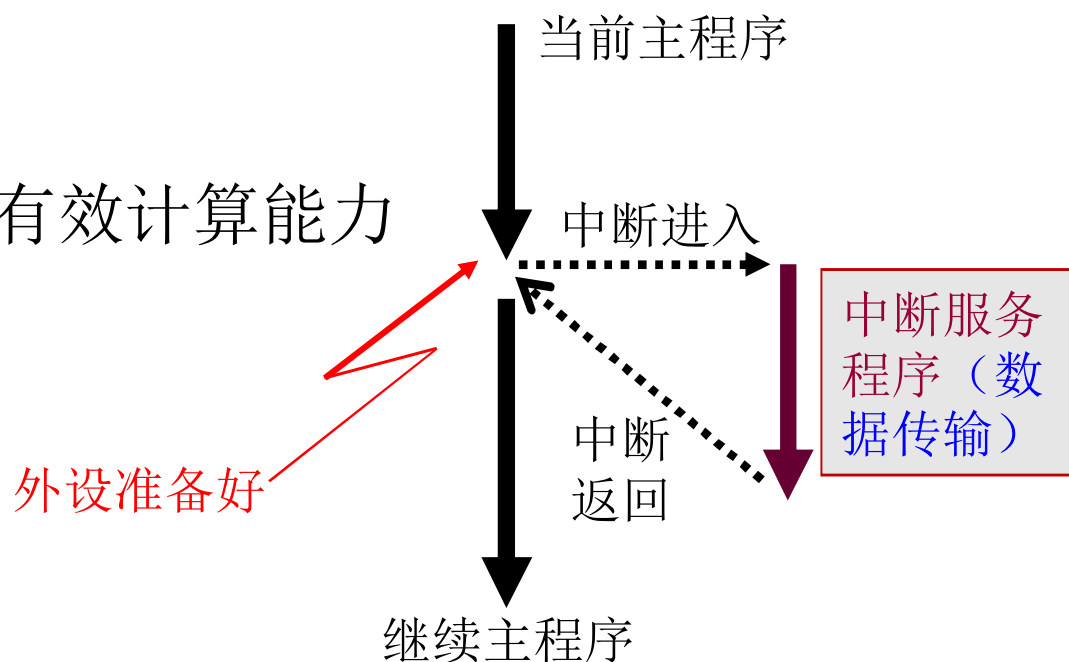
- 外设数据准备好或准备好接收时，产生中断信号
- CPU收到中断信号后，停止当前工作，执行数据传输。
- CPU完成数据传输后继续原来工作。

- 特点

- CPU和外设并行，CPU效率高

- 缺点

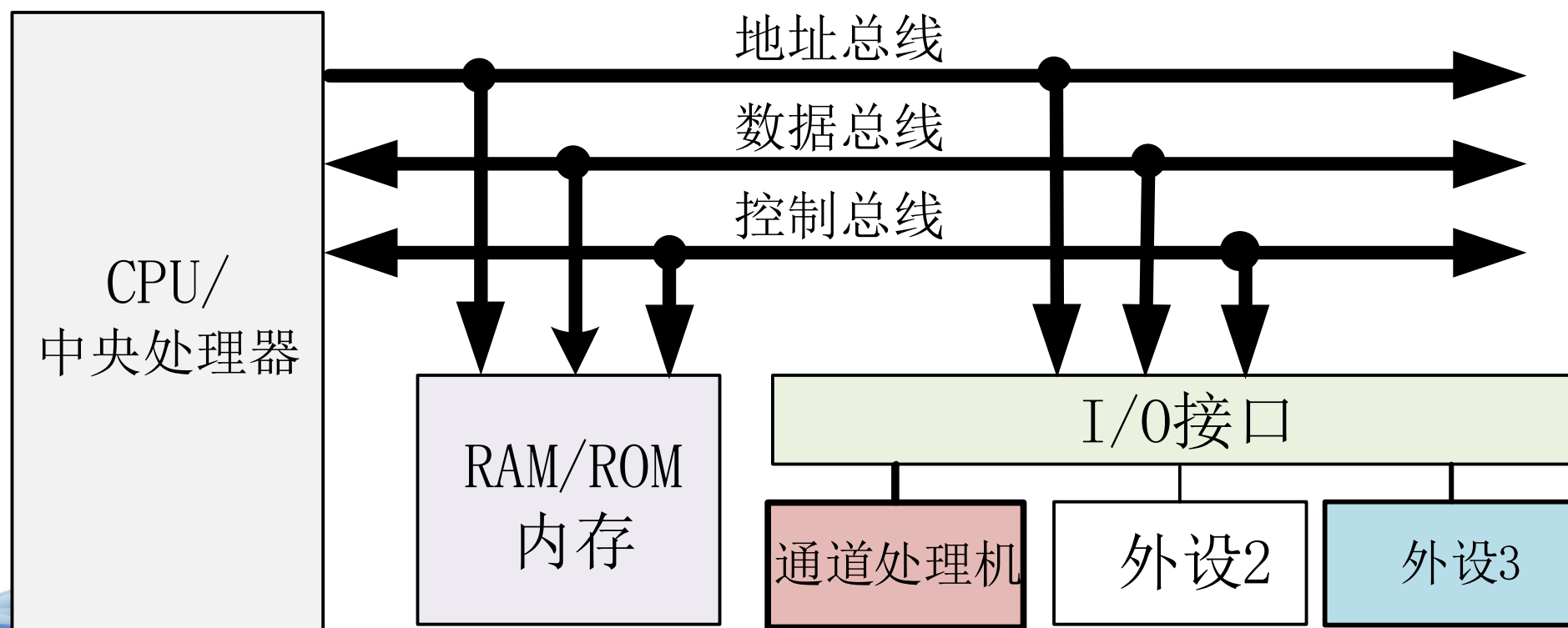
- 若设备频繁中断，影响CPU有效计算能力
- 数据吞吐率低
  - ◆ 适合少量数据低速传输。



# 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

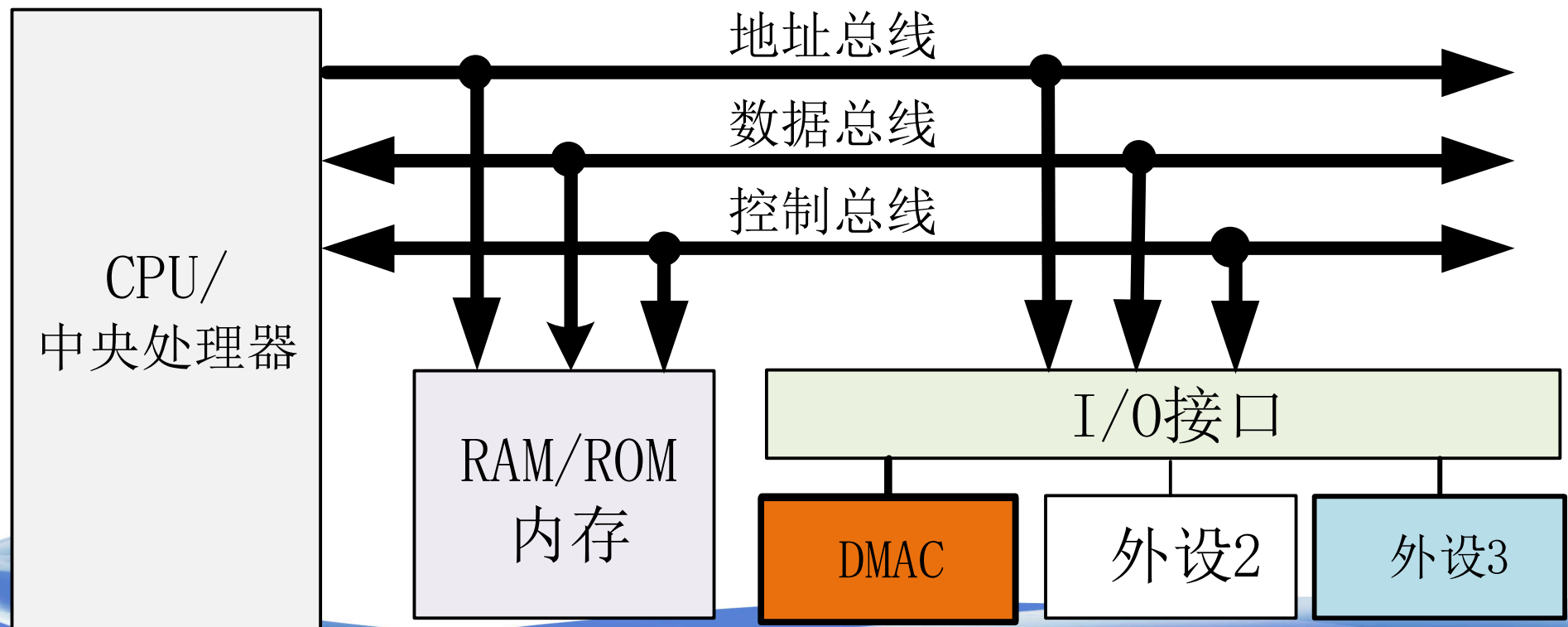
## ● 概念

■ 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**，**内存**，**硬盘**）

- ◆ 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

- 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

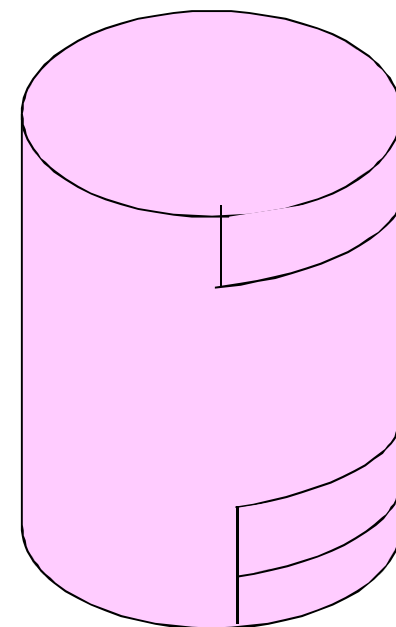
- 共享分配

- 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

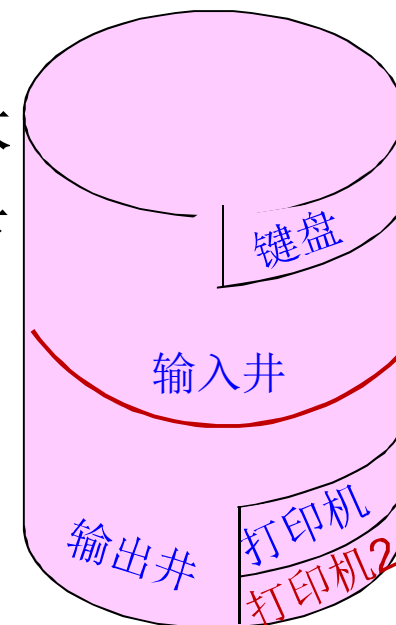
## ● 虚拟分配

### ■ 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



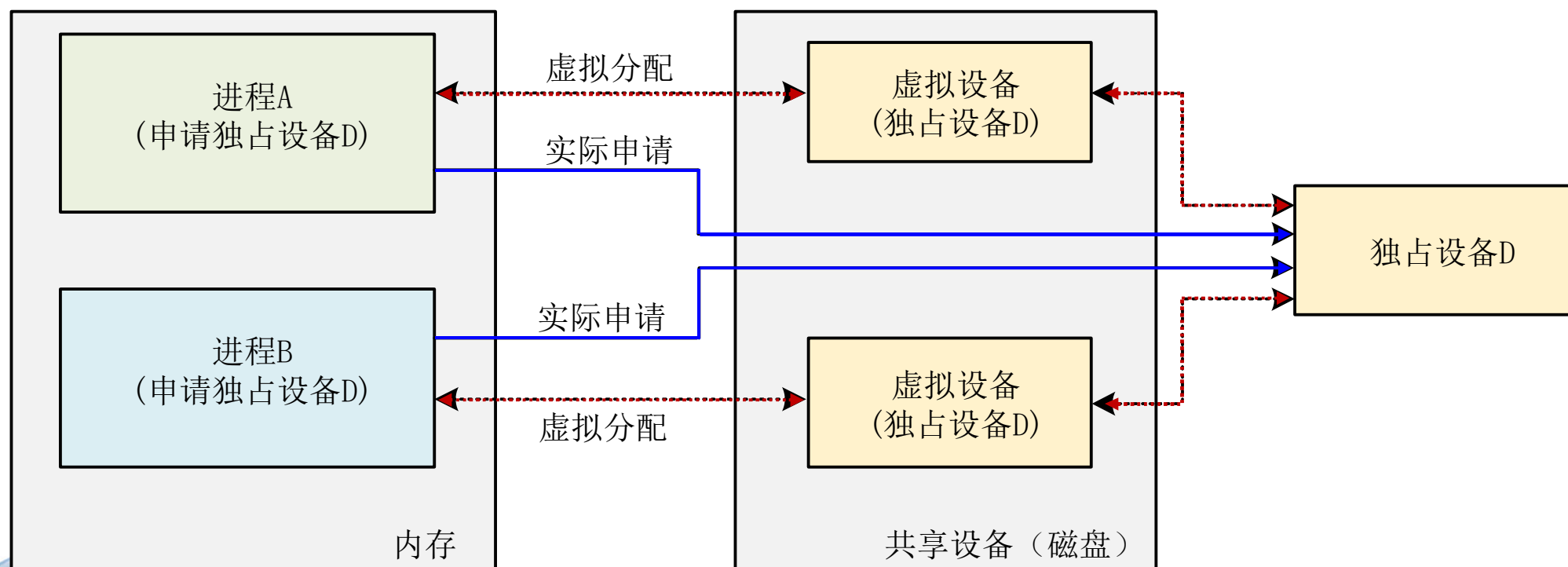
# 设备分配方法

## ● 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

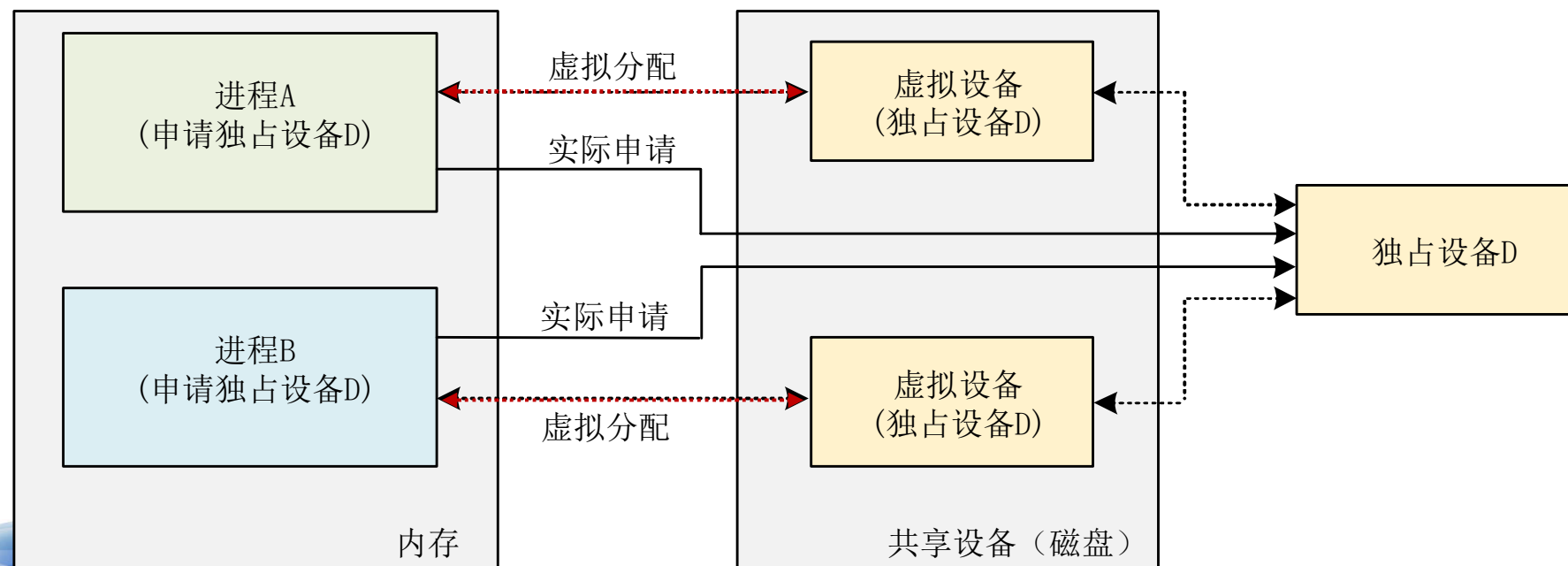
## ● 虚拟分配

■ 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

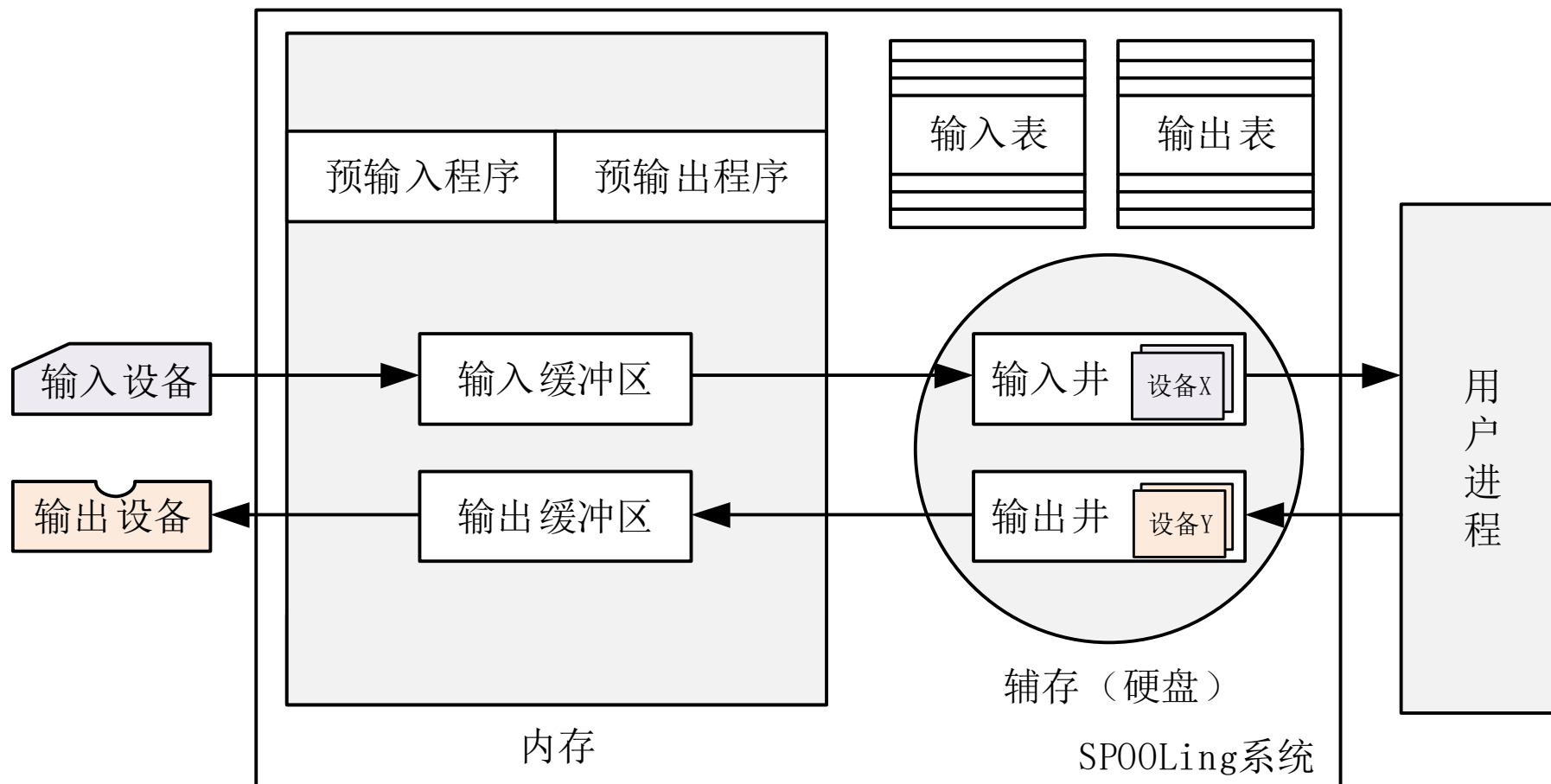
  - ◆ Simultaneous Peripheral Operations OnLine

  - ◆ 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



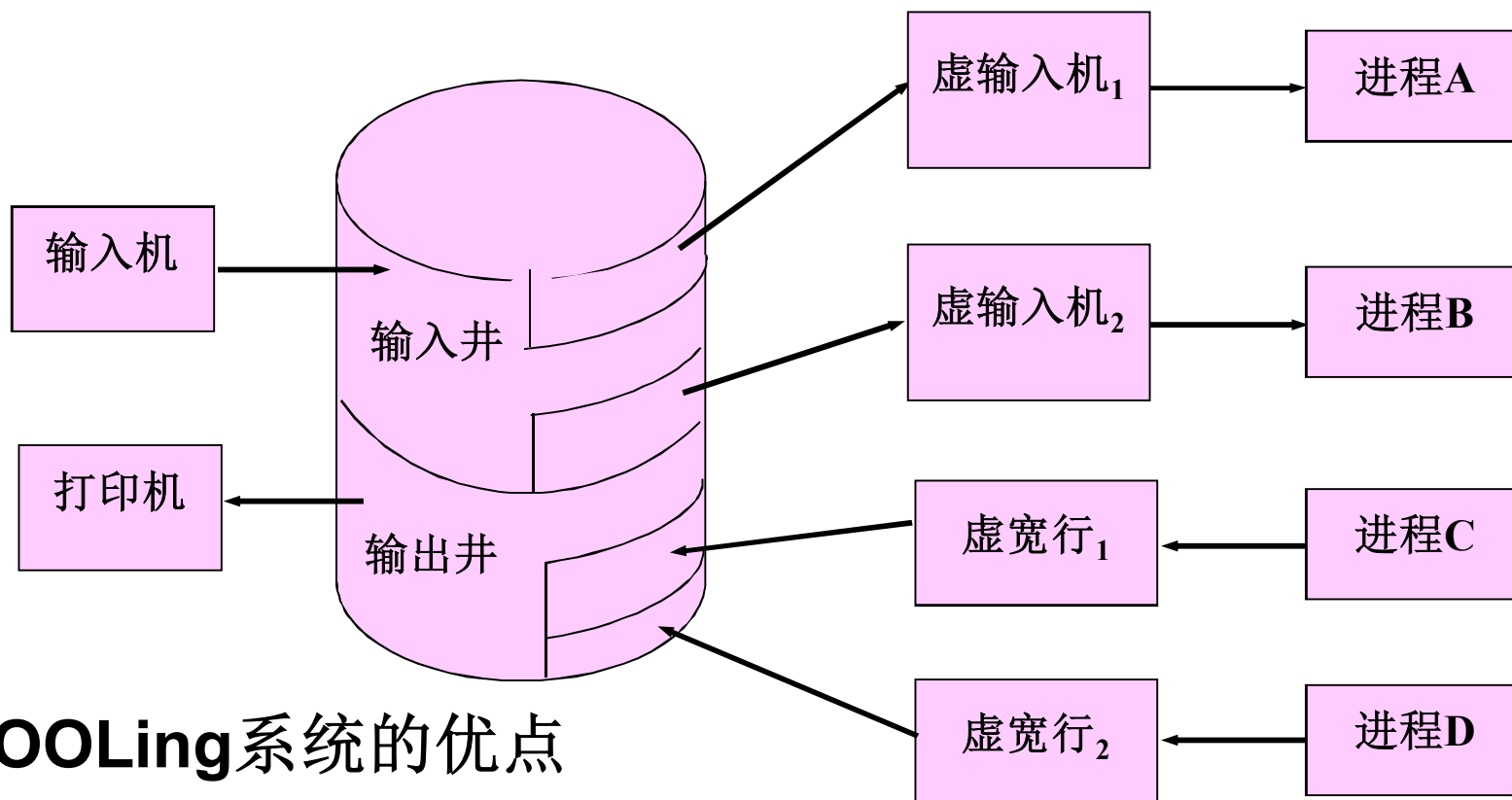
# SPOOLing系统的结构（硬件）

- 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 $\leftrightarrow$ 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 $\leftrightarrow$ 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