《操作系统原理》

# 第8章 设备管理

教师：苏曙光

华中科技大学软件学院

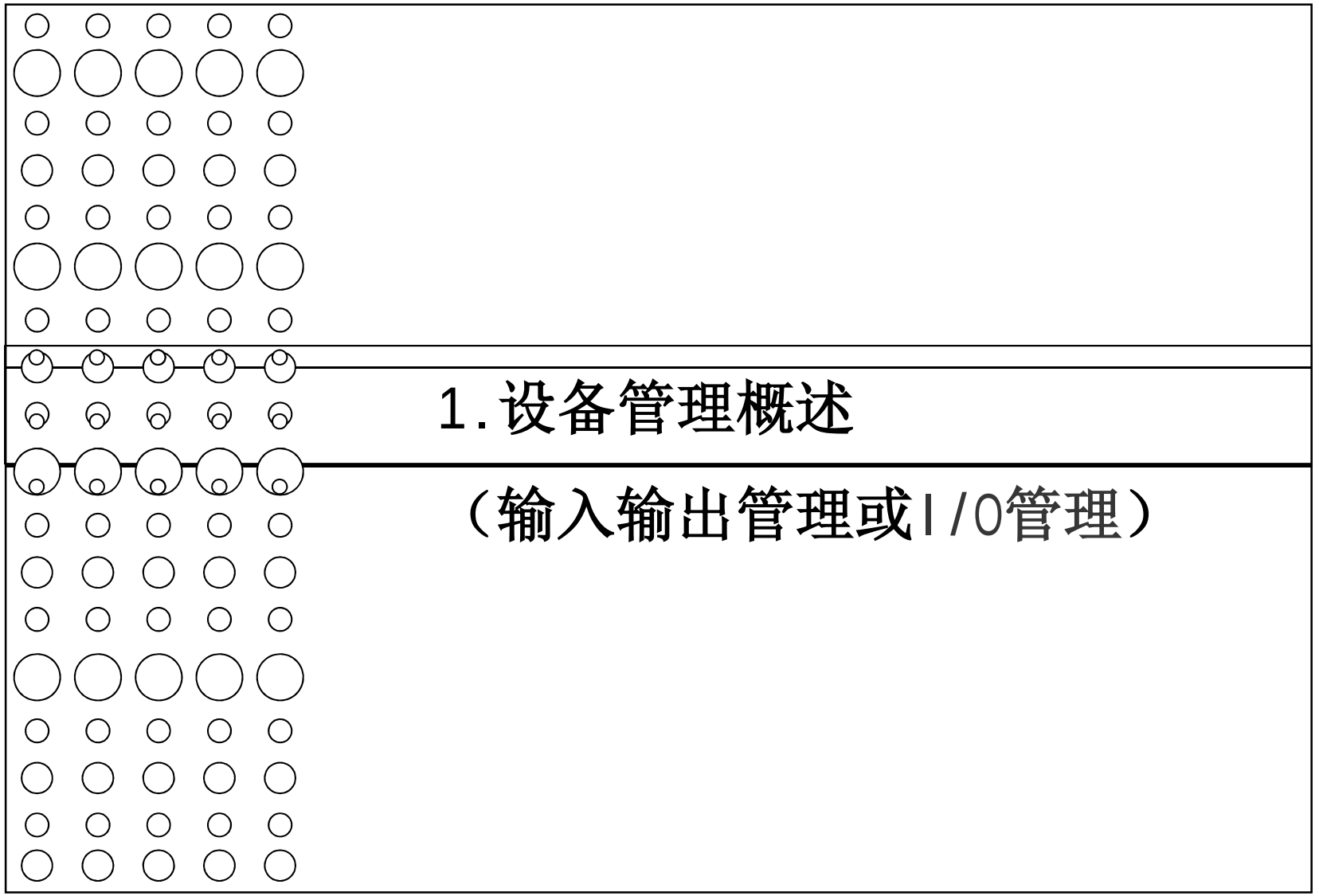
2015年3月-5月

## 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

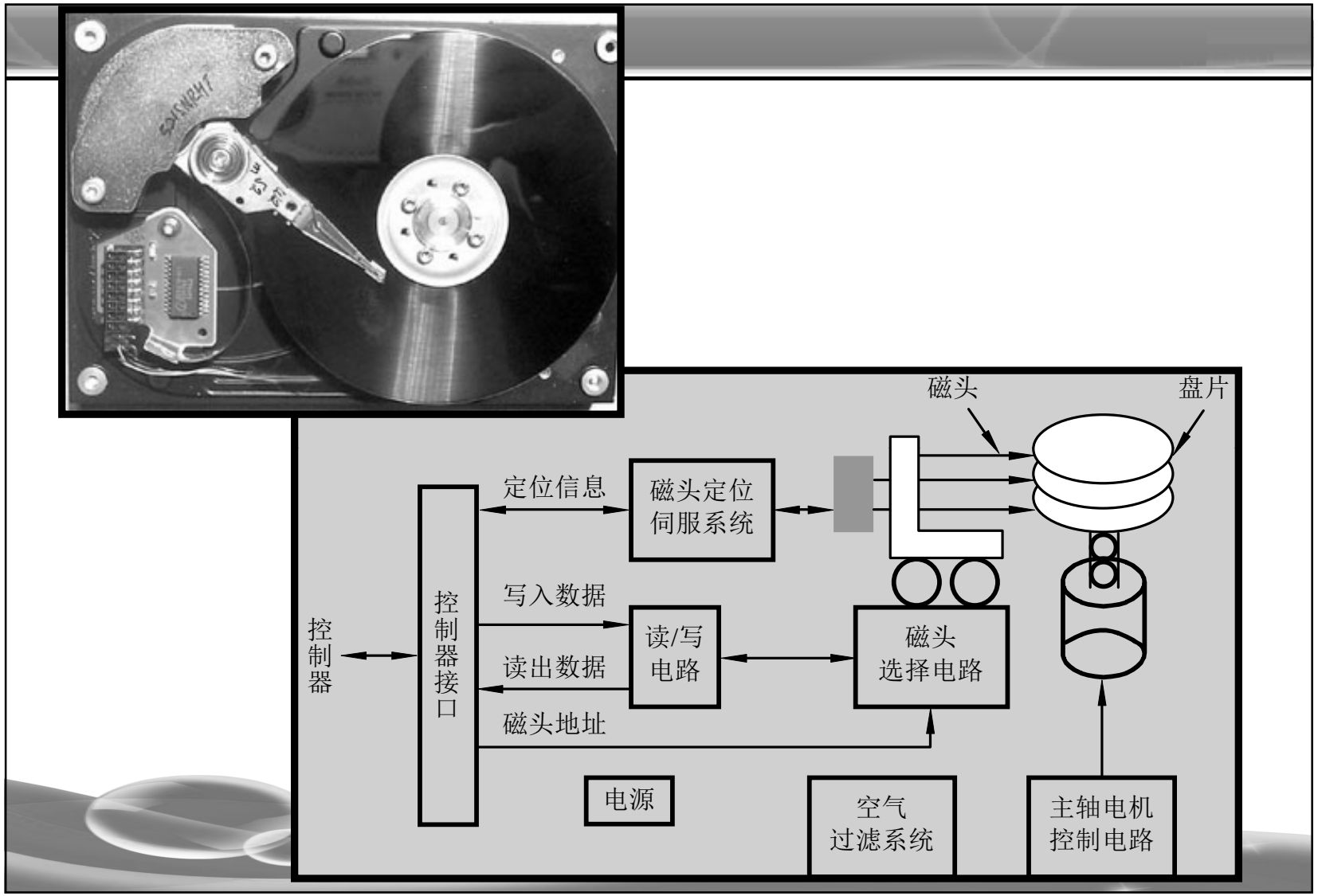
## 重点

- 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



# 1.设备管理概述

## （输入输出管理或I/O管理）


磁头
盘片
定位信息
磁头定位
伺服系统
控制器
控制器接口
写入数据
读出数据
磁头地址
读/写
电路
磁头
选择电路
电源
空气
过滤系统
主轴电机
控制电路

# 设备类型和特征

- **1. 按交互对象分类**
  - 人机交互设备：视频显示设备、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：磁盘、网卡
- **3. 按外设特性分类**
  - 使用特征：存储、输入/输出、终端
  - 数据传输率：低速(如键盘)、中速(如打印机)、高速(如网卡、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

# 设备管理系统的4个设计目标

- 1.提高设备利用率
- 2.设备的统一管理
- 3.设备独立性
- 4.字符代码的独立性

- 1.提高设备利用率
  - 合理分配和使用外部设备
  - 提高设备同CPU的并行程度
    - 中断/通道/DMA等技术和缓冲技术

- 2.设备的统一管理
  - 隐蔽设备的差别，向用户提供统一的设备使用接口
    - read/write
    - UNIX把外设作为特别文件处理，用文件的方法操作设备

## 3.设备独立性

- 具体物理设备对用户透明，用户使用统一规范的方式使用设备。
  - 物理名：ID或字符串
- 用户编程时使用设备逻辑名，由系统实现逻辑设备到物理设备的转换。
  - 逻辑名是友好名（Friendly Name）

# Winodws的例子

- 友好名：应用程序可见的名称：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, ……..);
……..
```

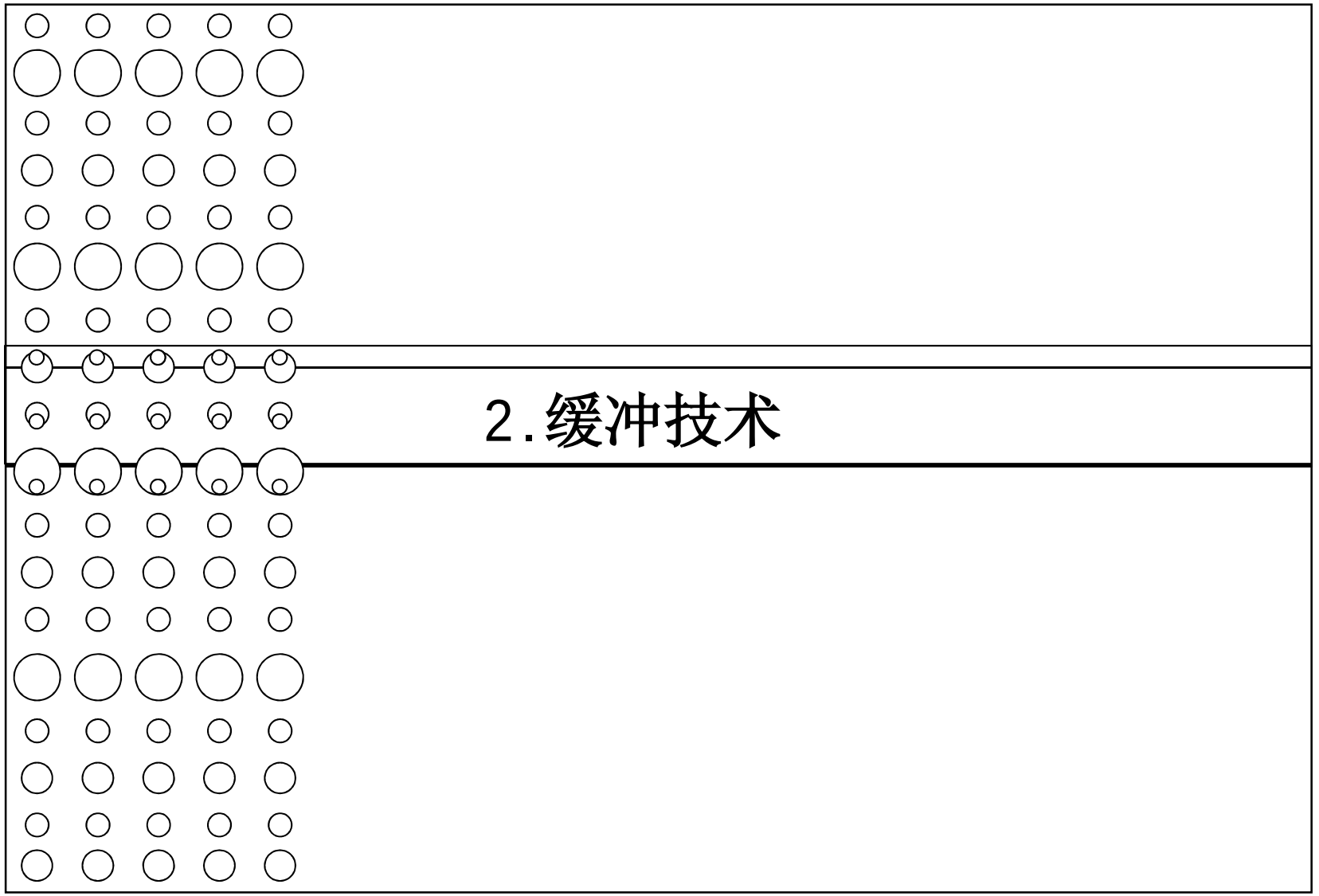
# Linux的例子

- 友好名：应用程序可见的名称：**/dev/test**

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

## 4.字符代码的独立性

- 各种外部设备使用的字符代码可能不同。设备管理系统必须能处理不同的字符代码。
- ASCII码（美国信息交换标准码）


2.缓冲技术

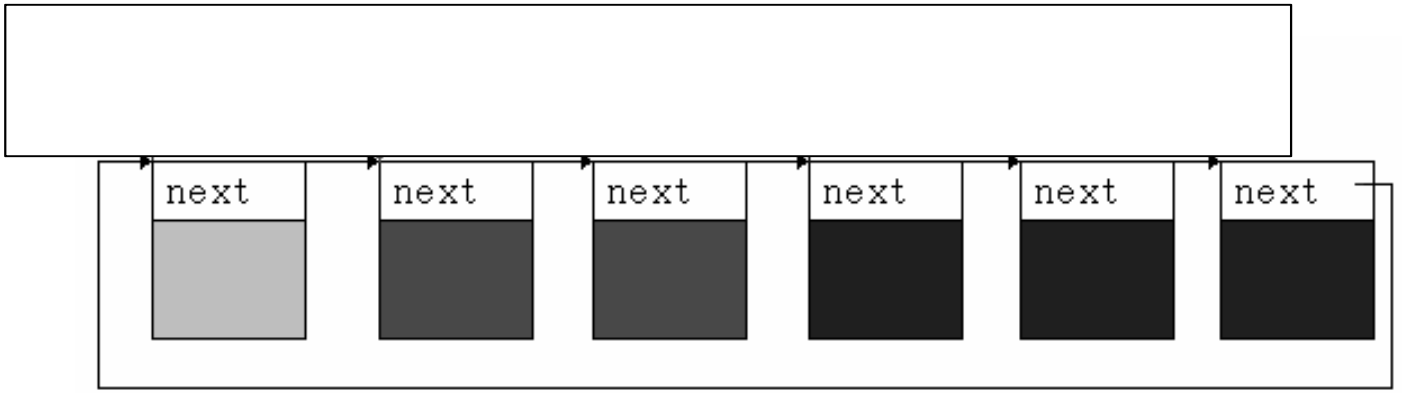
# 缓冲技术

- 缓冲作用
  - 连接两种不同数据传输速度的设备的媒介。
    - 平滑传输过程
    - 典型场景：CPU和I/O设备速度不匹配
  - 提升数据的存取效率
    - CPU 的数据成批（而不是逐字节）传送给I/O设备
    - I/O设备的数据成批（而不是逐字节）传送给CPU
    - 解决逻辑记录和物理记录大小不匹配的问题
- 缓冲的组成
  - 硬件缓冲器
    - 容量较小，一般放在接口中
  - 软件缓冲区
    - 容量较大，一般放在内存中。

- 常用的缓冲技术
  - 双缓冲
  - 环形缓冲
  - 缓冲池

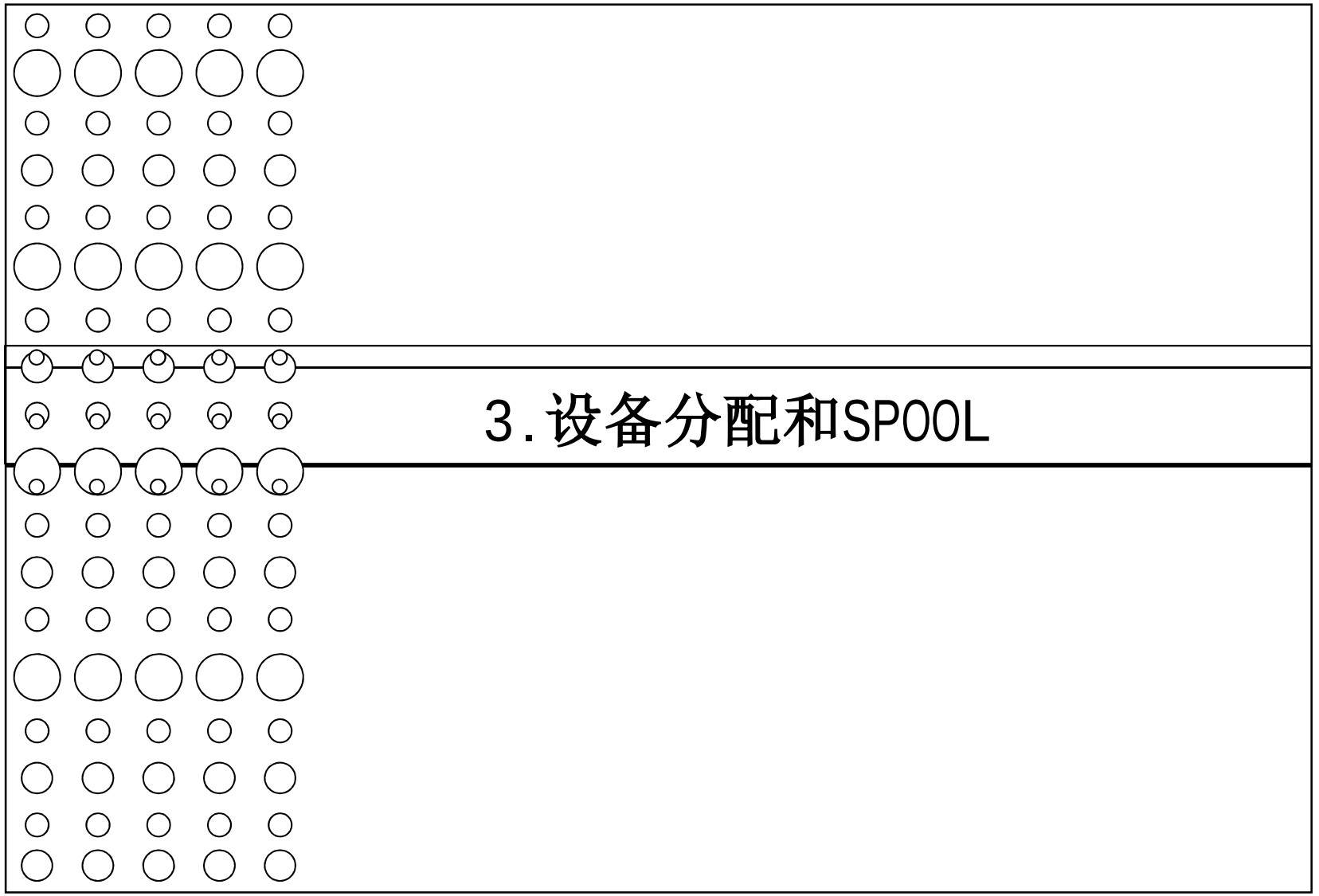
# 环形缓冲

- 若干缓冲单元首尾链接形成一个环：环形缓冲区




- 两个线程：输出线程（读），输入线程（写）
- 三个指针：输入指针in，输出指针out，开始指针：start
- 系统初始：start = in = out

- **输入时，要判断(in = = out)相等**
  - 若相等，则要等待（意味系统没有空缓冲区了）。
  - 否则，将信息送入in指向的缓冲区，然后用缓冲区的next 来更新in（即将in移到下一个缓冲区上）。
- **输出，要判断(out = = in)相等**
  - 若相等，则要等待（意味系统中没有数据可取）。
  - 否则，取出缓冲区中的信息，然后用缓冲区的next来更新out（即将out移到下一个缓冲区上）。



# 3.设备分配和SPOOL

## 3 设备分配

- 设备分配方法
  - 独享分配
  - 共享分配
  - 虚拟分配

- **独享分配**
  - 在进程执行前将其所要使用的设备分配给它；当其结束时才把设备释放收回系统。又称静态分配。
  - 独占设备往往采用独享分配。
  - 避免发生死锁

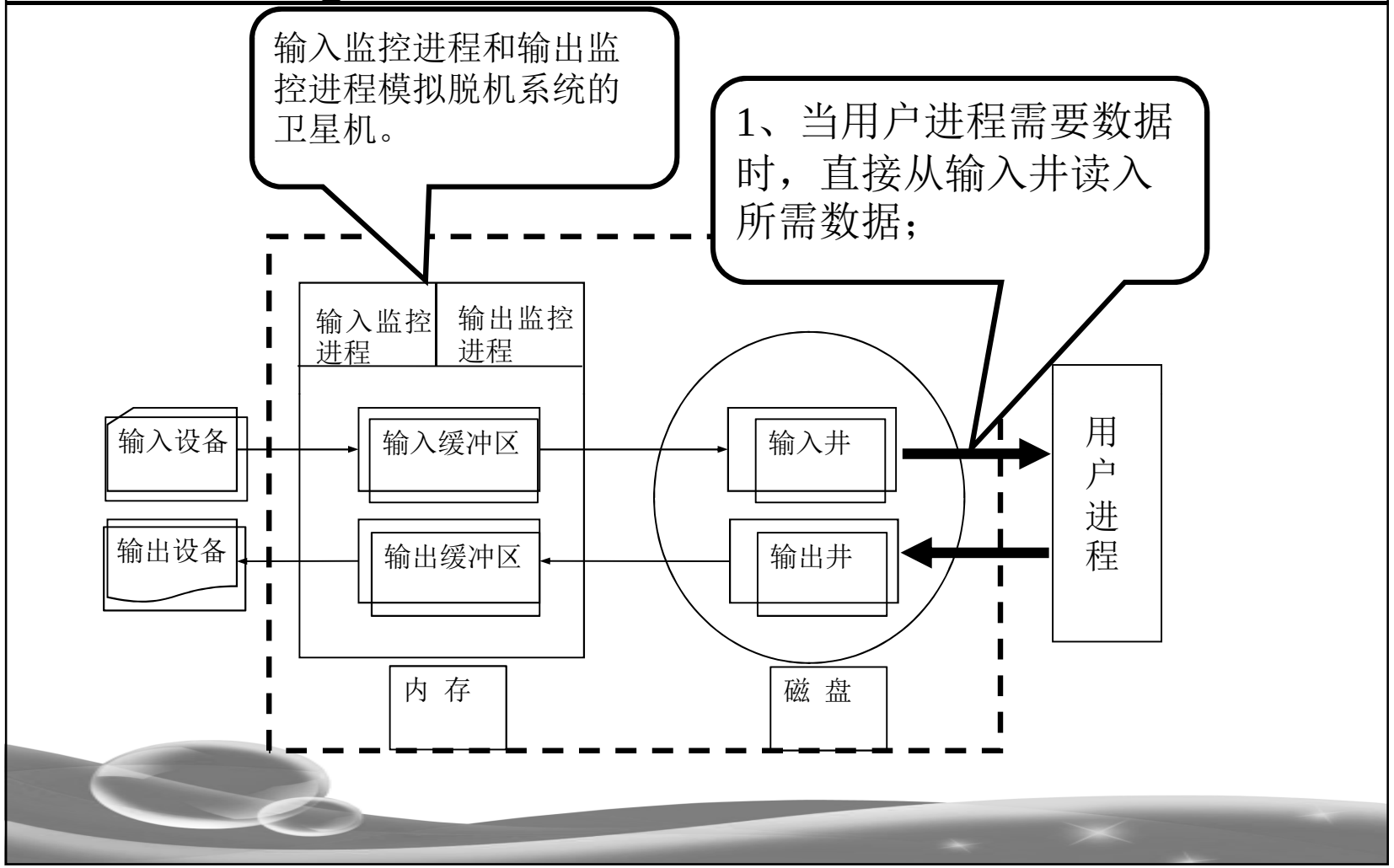
## l 共享分配

- n 当进程提出资源申请时，由设备管理模块进行分配，进程使用完毕后，立即归还。
- n 共享分配就是动态分配。
- n 共享设备往往采用共享分配。
- n 可能发生死锁

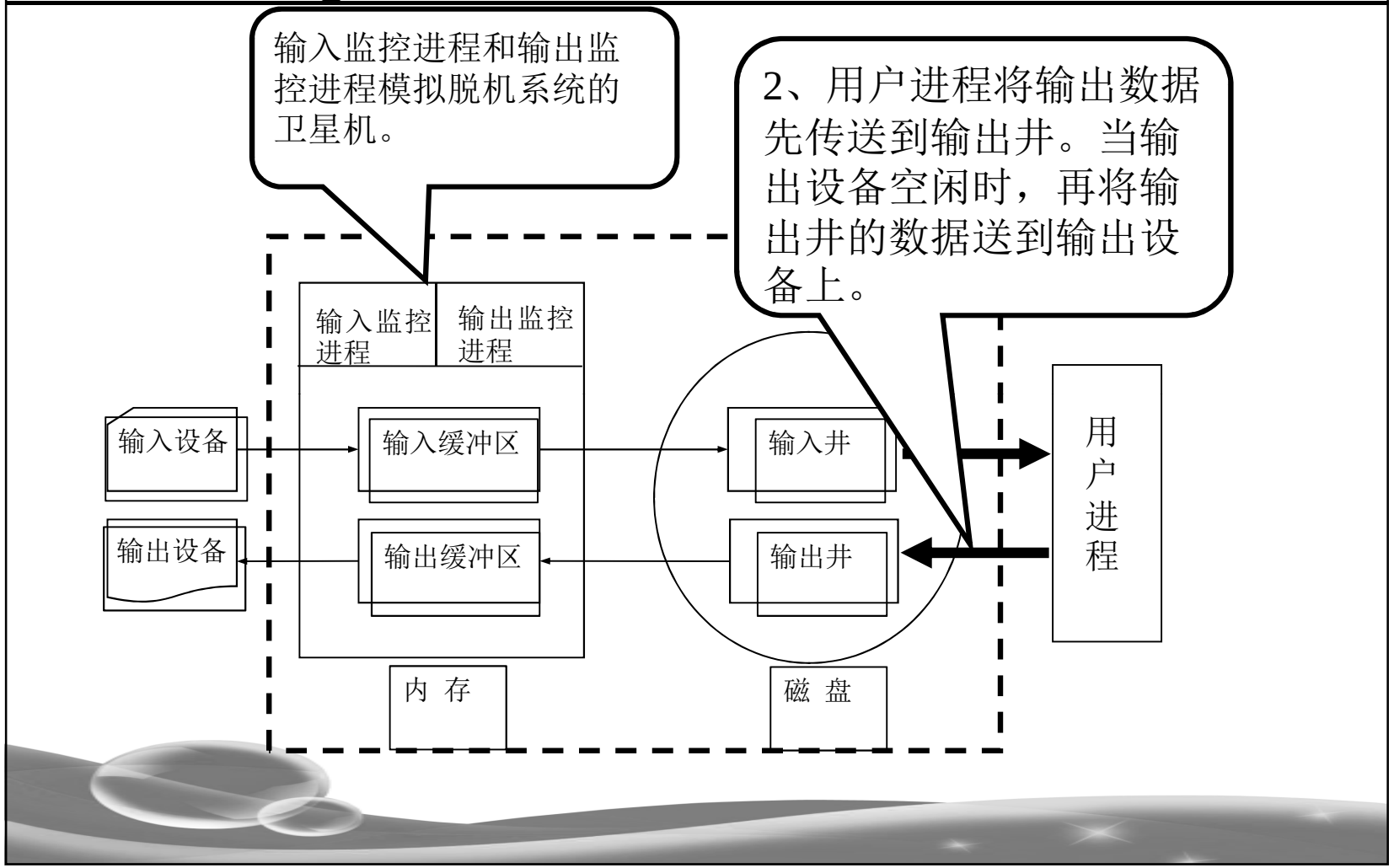
# 虚拟分配

- **虚拟技术**
  - 在一类物理设备上模拟另一类物理设备的技术
    - 通常借助辅存将独占设备转化为共享设备。
- **虚拟设备**
  - 用来代替独占设备部分辅存(包括控制接口)称为虚拟设备。
- **虚拟分配**
  - 当进程需要与独占设备交换信息时，就采用虚拟技术将与该独占设备所对应的虚拟设备（部分辅存）分配给它。
  - SPOOLing系统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/输出操作】

# SPOOLing


输入监控进程和输出监控进程模拟脱机系统的卫星机。
1、当用户进程需要数据时，直接从输入井读入所需数据；
输入监控进程
输出监控进程
输入设备
输出设备
输入缓冲区
输出缓冲区
输入井
输出井
用户进程
内 存
磁 盘

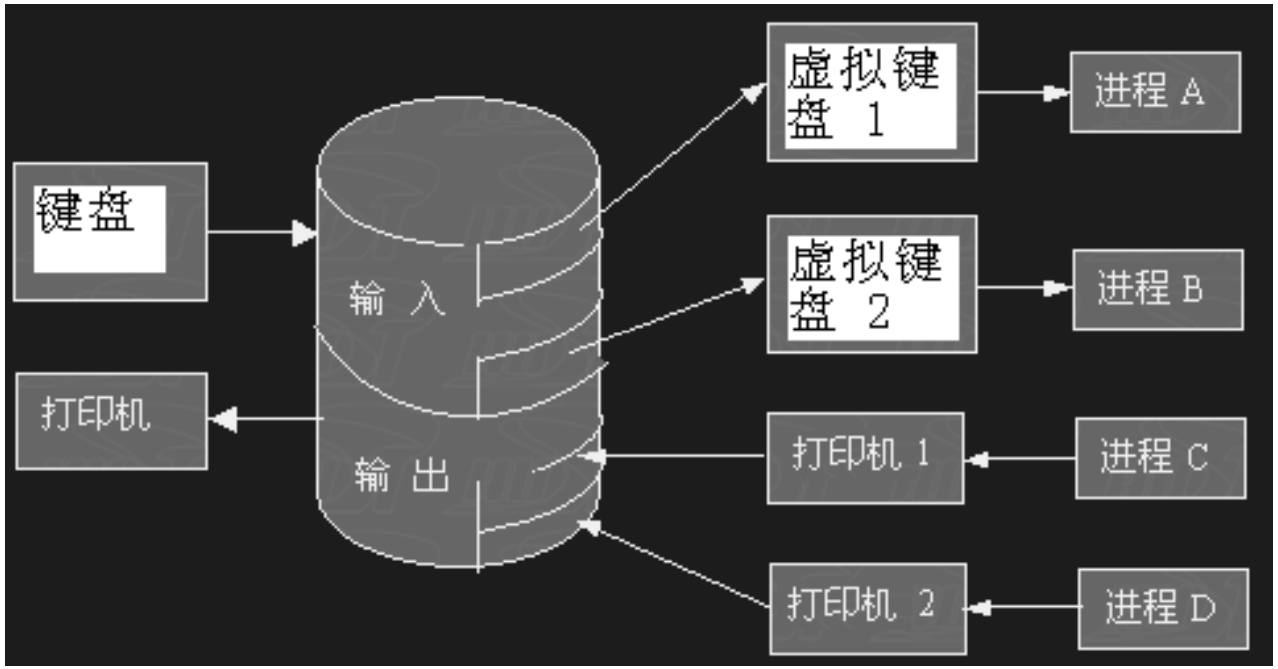
# SPOOLing


输入监控进程和输出监控进程模拟脱机系统的卫星机。
2、用户进程将输出数据先传送到输出井。当输出设备空闲时，再将输出井的数据送到输出设备上。
输入监控进程
输出监控进程
输入设备
输出设备
输入缓冲区
输出缓冲区
输入井
输出井
用户进程
内 存
磁 盘

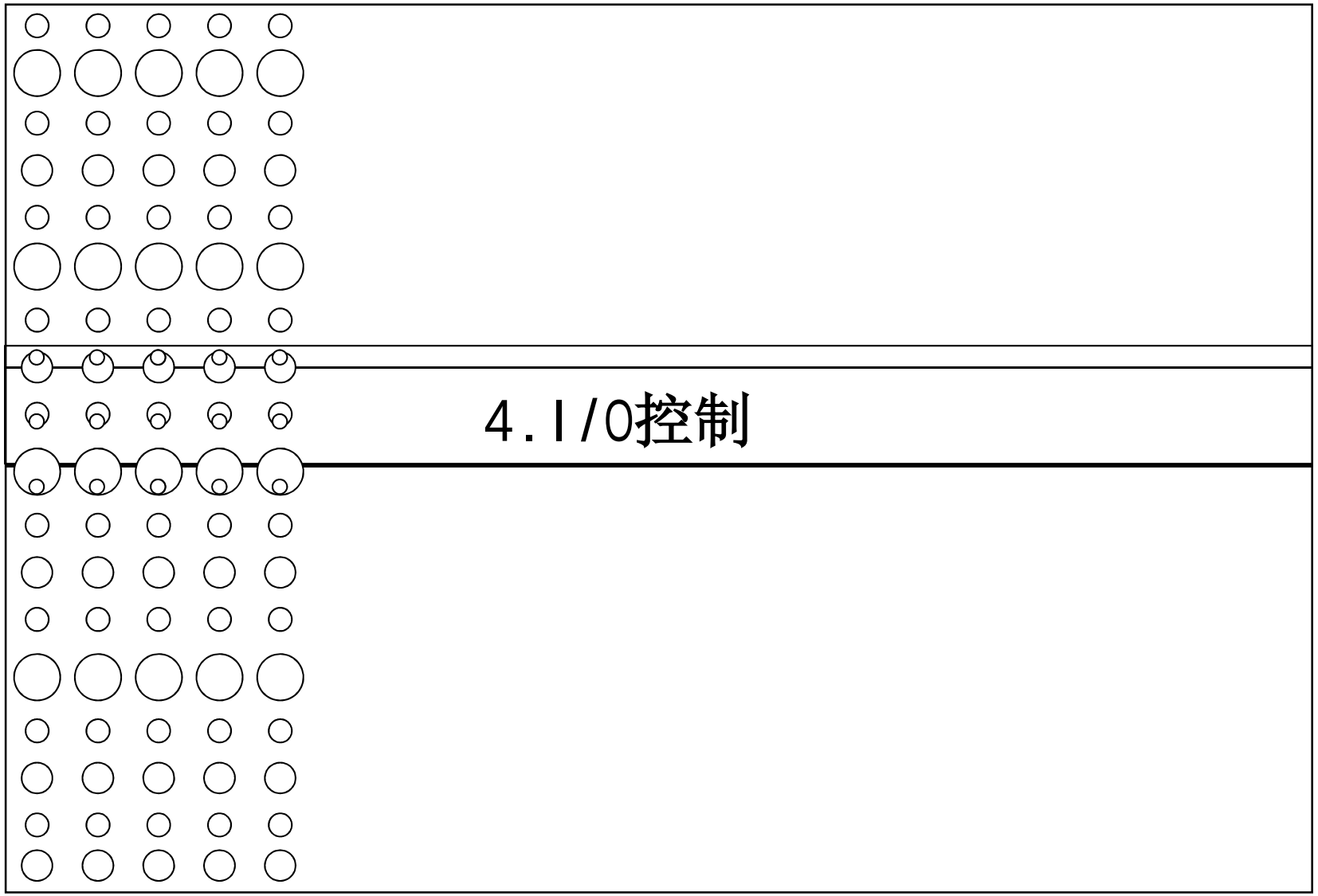
## SPOOLing的结构

- **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。
- **输入监控进程和输出监控进程**
  - 输入监控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；
  - 输出监控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



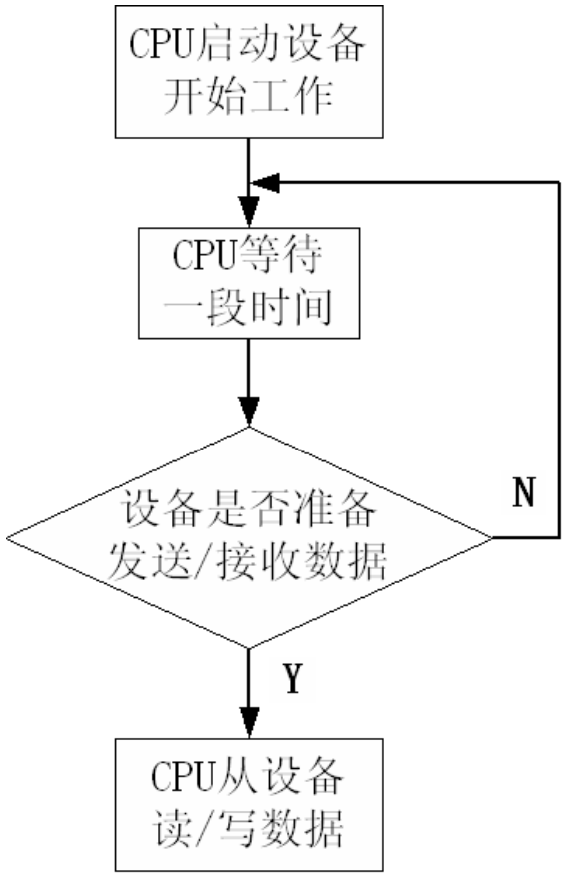
- **SPOOLing系统原理小结**
  - 任务执行前：预先将程序和数据输入到输入井中
  - 任务运行时：使用数据时，从输入井中取出
  - 任务运行时：输出数据时，把数据写入输出井
  - 任务运行完：外设空闲时输出全部数据和信息
- **SPOOLing优点**
  - “提高”了I/O速度
  - 将独占设备改造为“共享”设备
    - 实现了虚拟设备功能


4.I/O控制

- **I/O数据传输机制**
  - 查询方式（异步传送）
  - 无条件传送方式（同步传送）
  - 中断方式
  - 通道方式
  - DMA方式

# 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。



# 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。

## 中断方式

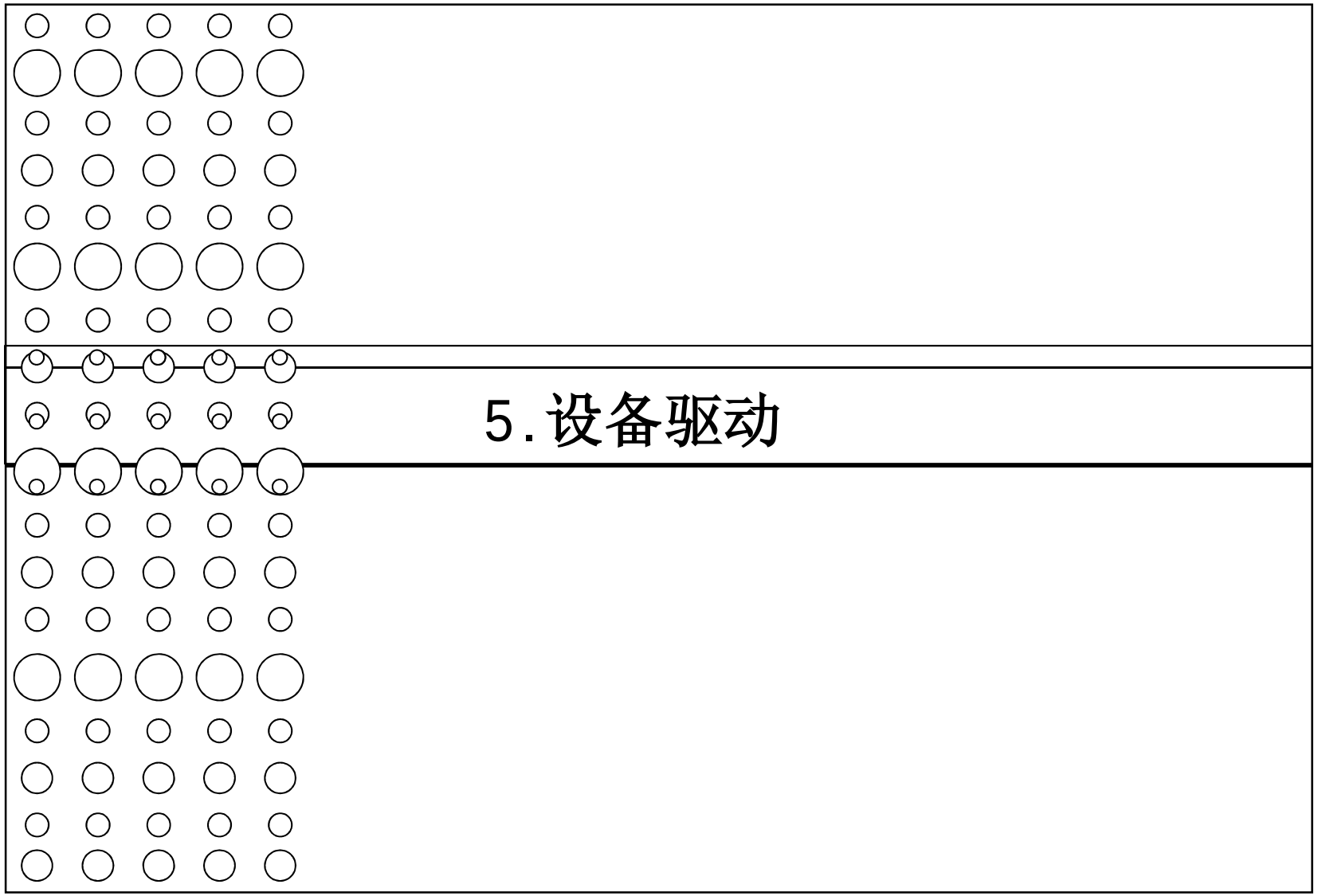
- 工作原理
  - 外设数据准备好或准备好接收时，产生中断信号
  - CPU收到中断信号后，停止当前工作，处理该中断事情：完成数据传输。
  - CPU处理完毕后继续原来工作。
- 特点和缺点
  - CPU和外设并行工作
  - CPU效率提高
  - 设备较多时中断频繁，影响CPU的有效计算能力。
  - CPU数据吞吐小（几个字节），适于低速设备。

# 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。

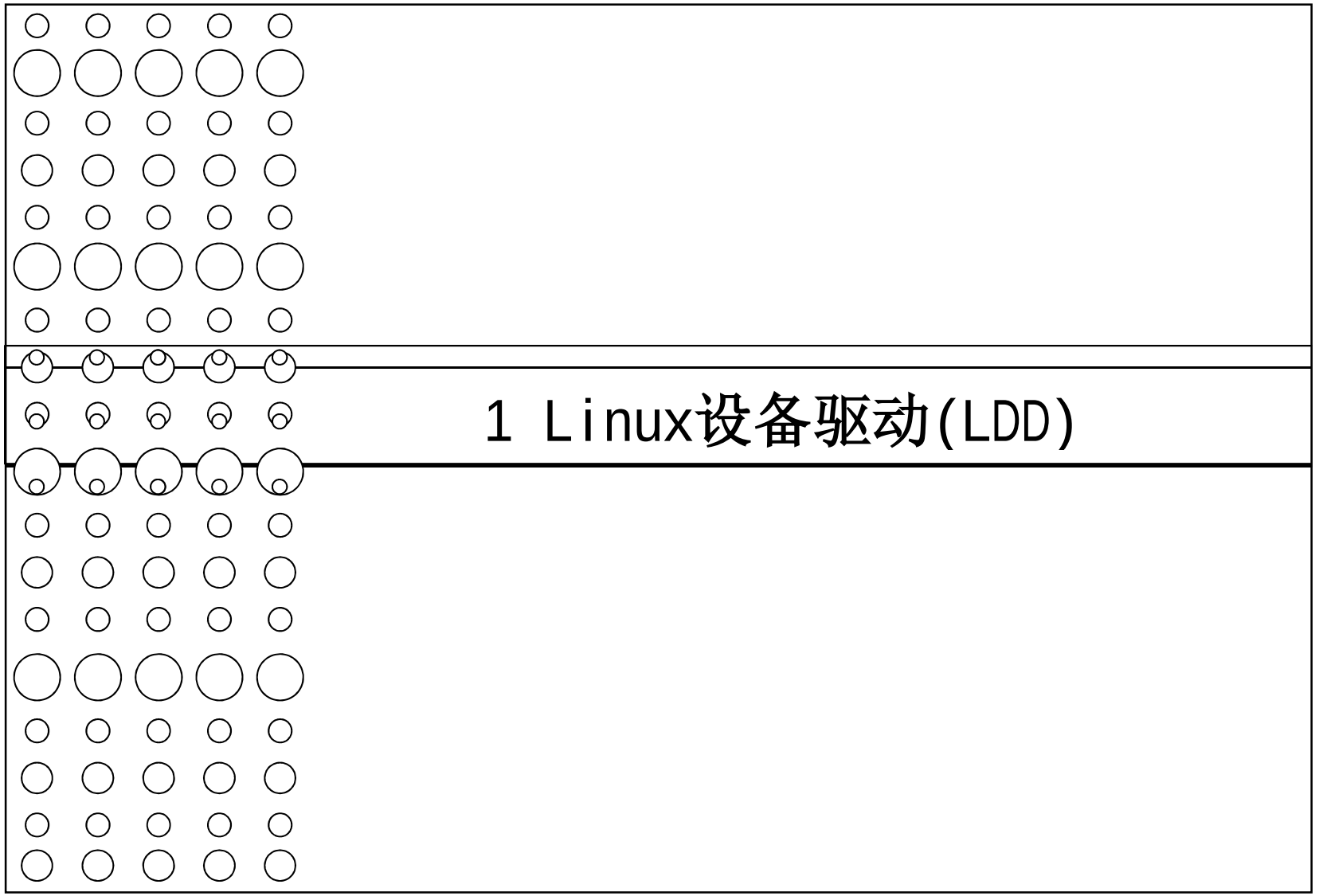
# DMA(直接内存访问)方式

- 概念
  - 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


5.设备驱动

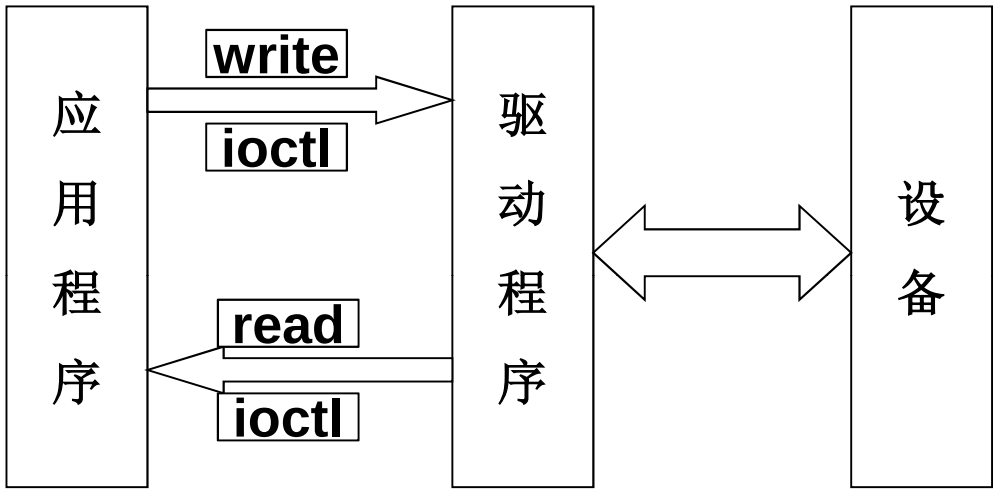
- 驱动程序概念
- **Linux驱动(LDD: Linux Device Driver)**
- **Windows驱动（WDM）**

- **驱动程序概念**
  - 应用程序通过驱动程序来使用硬件设备或底层软件资源。
  - 驱动程序工作在核心层（和OS一个层次）
  - 不同的操作系统提供不同机制来实现驱动程序
- **基本硬件基础**
  - 接口/端口
  - 中断机制
  - DMA机制
  - 虚拟驱动


1 Linux设备驱动(LDD)

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

## 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

- **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）

- **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。

## l 命令 ls –l /dev 列出设备文件

```
[root@localhost proc]# ls -l /dev|more
total 228
crw-------    1 root     root      10,  10 Jan 30  2003 adbmouse
crw-r--r--    1 root     root      10, 175 Jan 30  2003 agpgart
crw-------    1 root     root      10,   4 Jan 30  2003 amigamouse
crw-------    1 root     root      10,   7 Jan 30  2003 amigamouse1
crw-------    1 root     root      10, 134 Jan 30  2003 apm_bios
drwxr-xr-x    2 root     root         4096 Jun 29  2006 ataraid
crw-------    1 root     root      10,   5 Jan 30  2003 atarimouse
crw-------    1 root     root      10,   3 Jan 30  2003 atibm
crw-------    1 root     root      10,   3 Jan 30  2003 atimouse
crw-------    1 root     root      14,   4 Jan 30  2003 audio
crw-------    1 root     root      14,  20 Jan 30  2003 audio1
crw-------    1 root     root      14,   7 Jan 30  2003 audioctl
brw-rw----    1 root     disk      29,   0 Jan 30  2003 aztcd
crw-------    1 root     root      10, 128 Jan 30  2003 beep
brw-rw----    1 root     disk      41,   0 Jan 30  2003 bpcd
crw-------    1 root     root      68,   0 Jan 30  2003 capi20
crw-------    1 root     root      68,   1 Jan 30  2003 capi20.00
crw-------    1 root     root      68,   2 Jan 30  2003 capi20.01
crw-------    1 root     root      68,   3 Jan 30  2003 capi20.02
crw-------    1 root     root      68,   4 Jan 30  2003 capi20.03
crw-------    1 root     root      68,   5 Jan 30  2003 capi20.04
crw-------    1 root     root      68,   6 Jan 30  2003 capi20.05
```

- **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
    - 次设备号只在驱动程序内部使用，系统内核直接把次设备号传递给驱动程序，由驱动程序去管理。

# 功能完整的Linux设备驱动程序结构

- 功能完整的LDD结构
  - 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
  - 驱动程序的注册
  - 驱动程序的注销

# 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… }
```

# 打开和关闭操作

- 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

# 写入操作

```
static int my_write(struct file *file, const char * buffer, size_t count,
loff_t * ppos){
    char led_status = 0;
    copy_from_user(&led_status, buffer, sizeof(led_status));
    if(led_status == 0x01){            //如果应用程序传来的数据是0x01
        AT91F_PIOB_SetOutput(LED);     //打开LED
   }else{
        AT91F_PIOB_ClearOutput(LED);   //关闭LED
   }
   return 0;
}
```

# 文件操作接口：文件操作结构体

```
l   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

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

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

write(fd, &led_on, 1);                    //L
close(fd);                                //关
```

# 设备注册（初始化）

```
l static int __init my_init(void){
    //硬件初始化
    AT91F_PIOB_Enable(LED);
    AT91F_PIOB_OutputEnable(LED);
    //字符设备注册
    Led_Major = register_chrdev(0, DEVICE_NAME, &my_fops);
    }
    // 创建设备文件
    #ifdef CONFIG_DEVFS_FS
        Devfs_Led_Dir = devfs_mk_dir(NULL, "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=led.o
SOURCE=io_led.c
CC=arm-linux-gcc
COMP=-Wall -O2 -DMODULE -D_KERNEL_ -I /home/armlinux/linux-2.4.19-rmk7/include –c
$(OBJ):$(SOURCE)
        $(CC) $(COMP) $(SOURCE)
clean:
        rm $(OBJ)
```

- 运行make 命令,编译后就生成名为led.o的驱动程序

# 驱动程序加载

- 动态加载
  - n 通过insmod等命令
  - n 调试过程

# 模块动态加载

- 驱动程序模块插入内核

```
#insmod 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 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;
}
```

# 驱动程序加载

- 2种加载方法
  - 动态加载
    - 通过insmod等命令
    - 调试过程
  - 静态编译入内核
    - 发行过程