

I 内容

- n 设备管理概述

- n 缓冲技术

- n 设备分配

- n I/O设备控制

- n 设备驱动程序

I 重点

- n 理解缓冲的作用

- n 理解SPOOLING技术

- n 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)

设备类型和特征

I 1. 按交互对象分类

- n 人机交互设备：视频显示设备、键盘、鼠标、打印机
- n 与CPU等交互的设备：磁盘、磁带、传感器、控制器
- n 计算机间的通信设备：网卡、调制解调器

I 2. 按交互方向分类

- n 输入设备：键盘、扫描仪
- n 输出设备：显示设备、打印机
- n 双向设备：输入/输出：磁盘、网卡

I 3. 按外设特性分类

- n 使用特征：存储、输入/输出、终端
- n 数据传输率：低速(如键盘)、中速(如打印机)、高速(如网卡、磁盘)
- n 信息组织特征：字符设备(如打印机), 块设备(如磁盘), 网络设备

设备管理的主要功能

- 丨 设备分配
- 丨 设备映射
- 丨 设备驱动
- 丨 I/O缓冲区的管理

设备分配

- 丨 多道程序系统中的设备不允许用户直接使用，而是由操作系统统一调度和控制。
- 丨 设备分配功能是设备管理的基本任务。
- 丨 设备分配程序按照一定的策略，为申请设备的用户进程分配设备，记录设备的使用情况。

设备映射

I 设备无关性

- n 为了提高应用软件对运行平台的适应能力，方便实现应用软件I/O重定向，大多数现代操作系统均支持应用软件对设备的无关性，即通常所说的设备无关性，或者设备独立性。
- n **设备无关性**：应用软件所引用的、用于实现I/O操作的设备与物理I/O系统中实际安装的设备没有固定的联系。

设备映射

I 逻辑设备和物理设备

- n 逻辑设备：应用软件所引用的用于实现I/O操作的设备
- n 物理设备：物理I/O系统中实际安装的设备。
- n 从应用软件的角度看，逻辑设备是一类物理设备的抽象。从操作系统设备管理程序的角度看，物理设备则是某种逻辑设备的实例。

设备映射

I 设备映射

- n 如果某系统支持设备无关性，那么该系统中应用软件所引用的逻辑设备与实际安装的物理设备没有固定的联系。
- n 事实上，在应用软件运行期间，操作系统的设备管理程序必须将该应用软件对逻辑设备的引用转换成对相关物理设备的引用。
- n 设备管理的这种功能称为逻辑设备到物理设备的映射功能，简称设备映射功能。

设备驱动

I 设备驱动

n指对物理设备进行控制，以实现真正的I/O操作。

n设备驱动的主要任务是：接收上层软件发来的抽象服务请求，例如读/写命令，再把它转换为具体要求，通过一系列的I/O指令，控制设备完成请求的操作；同时，设备驱动程序还将设备发来的有关信号传送给上层软件，例如设备是否已损坏等。

设备驱动

I 设备驱动程序

- n 设备驱动程序与硬件密切相关，应为每一类设备配置一种驱动程序。
- n 设备驱动程序一般由设备开发厂商根据操作系统的要求组织编写，操作系统仅对与设备驱动的接口提出要求，一般不负责具体设备驱动程序的编写。
- n 有时候，某些硬件无法在某种操作系统中使用，原因很可能就是没有专门的或通用的设备驱动程序，或者设备驱动程序设计有问题，使得设备无法正常工作。

I/O缓冲区管理

- I 为了缓和处理机与外部设备间速度不匹配的矛盾，提高处理机和外部设备间的并行性，现代操作系统大都在设备管理部分引入了缓冲技术。
- I 通常，缓冲区是指内存中的若干区域，用于缓存进程与外部设备间的数据传输。又称为I/O缓冲区。
- I I/O缓冲区管理的任务是：组织I/O缓冲区，并为使用者提供获得和释放I/O缓冲区的手段。

设备管理系统的4个设计目标

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

I 1. 提高设备利用率

- n 合理分配和使用外部设备

- n 提高设备同CPU的并行程度

 - u 中断/通道/DMA等技术和缓冲技术

I 2. 设备的统一管理

- n 隐蔽设备的差别，向用户提供统一的设备使用接口

 - u read/write

 - u UNIX把外设作为特别文件处理，用文件的方法操作设备

I 3. 设备独立性

- n 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。

 - u 物理名：ID或字符串

- n 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。

 - u 逻辑名是友好名 (Friendly Name)

Winodws的例子

I 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

I 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);  
}
```

I 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

I 缓冲作用

n 连接两种不同数据传输速度的设备的媒介。

u 平滑传输过程

u 典型场景：CPU和I/O设备速度不匹配

n 提升数据的存取效率

u CPU 的数据成批（而不是逐字节）传送给I/O设备

u I/O设备的数据成批（而不是逐字节）传送给CPU

u 解决逻辑记录和物理记录大小不匹配的问题

I 缓冲的组成

n 硬件缓冲器

- u 容量较小，一般放在接口中，不需要处理机的直接管理，不会影响系统性能。

n 软件缓冲区

- u 容量较大，一般放在内存中。

提前读

- 丨 **提前读**指用户进程从 **I/O缓冲区**中取走前一个数据以后，立即发出对下一个数据的输入请求。
- 丨 操作系统将在适当的时候响应该请求以便把用户进程需要的下一个数据从用户进程指定的输入设备读入到**I/O 缓冲区**中。
- 丨 显然，用户进程加工前一个数据的工作与操作系统输入下一个数据的工作可以同时进行。

从I/O buffer取数据A	处理数据A	处理数据B
	从外设取数据B到I/O buffer	

延后写

- 丨 延后写是指，当用户进程请求输出数据时，操作系统将很快把用户进程请求输出的数据从用户进程的工作区中取走并将其暂时存放在I/O缓冲区中。
- 丨 直到用户进程指定的输出设备空闲时，操作系统才把暂时存放在I/O缓冲区中的用户进程的输出数据写入用户进程指定的输出设备上。
- 丨 显然，用户进程生成下一个输出数据的工作与操作系统真正输出前一个输出数据的工作可以同时进行。

把数据A写到I/O buffer	处理数据B	把数据B写到I/O buffer
	把数据A从I/O buffer写到外设	

I 常用的缓冲技术

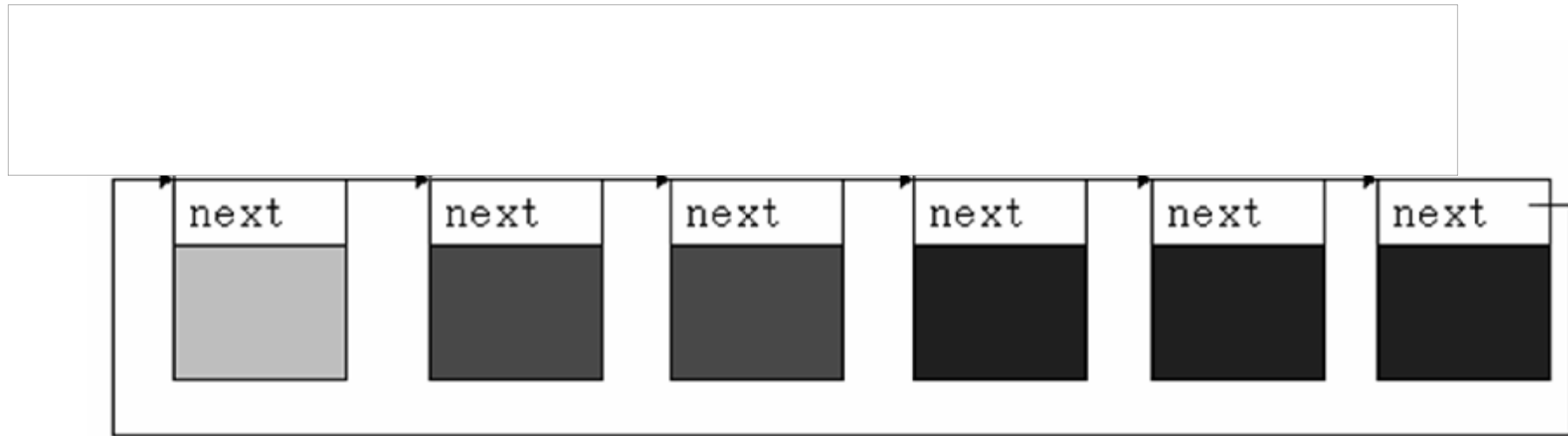
n 双缓冲

n 环形缓冲

n 缓冲池

环形缓冲

Ⅰ 若干缓冲单元首尾链接形成一个环：环形缓冲区



n两个线程：输出线程（读），输入线程（写）

n三个指针：输入指针in，输出指针out，开始指针：start

n系统初始：start = in = out

I 输入时，要判断(**in == out**)相等

n 若相等，则要等待（意味系统没有空缓冲区了）。

n 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）。

I 输出，要判断(**out == in**)相等

n 若相等，则要等待（意味系统中没有数据可取）。

n 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SP00L

3 设备分配

I 设备分配方法

n 独占设备的分配

n 共享设备的分配

n 虚拟分配

独占型设备 VS 共享型设备

I 独占型设备包括所有的字符型设备。

- n 任意时间段内最多只能被一个进程占用。

- n 使用时，进程首先向系统申请，可能进入阻塞状态。当设备可用时，唤醒一个等待进程。使用完毕以后，进程必须释放设备。

I 共享型设备包括所有块型设备。

- n I/O传输单位：块

- n 宏观上，一个共享型设备可以被多个进程同时占用；

- n 微观上，多个进程交替使用同一设备。

- n 进程使用这类设备时，无须申请或释放设备，也不存在某个进程占用设备的问题。

独占型设备的分配

- I 进程使用独占设备的过程：申请、使用、释放。
 - n1.进程申请设备
 - n2.系统分配设备
 - n3.进程发送使用命令
 - n4.系统将转到设备驱动模块完成一次I/O传输
 - n5.进程释放设备
 - n6.系统回收设备
- I 当进程的一次设备申请成功，直到发出释放命令期间，一直独占设备。

共享型设备的分配

- 丨 进程使用共享型设备不需要申请，当然也就不存在占用或释放设备的问题。
- 丨 但是，I/O期间只能有一个进程使用设备。因此，进程使用共享型设备之前，有一个隐含的申请命令；使用完毕以后，有一个隐含的释放命令，以实现一次**基本I/O传输**期间的排它性，保证操作的正常进行。
- 丨 使用共享型设备的过程中，进程完全有可能进入阻塞等待状态。

虚拟分配

I 虚拟技术

- n 在一类物理设备上模拟另一类物理设备的技术

- u 通常借助辅存将独占设备转化为共享设备。

I 虚拟设备

- n 用来代替独占设备部分辅存(包括控制接口)称为虚拟设备。

I 虚拟分配

- n 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。

- n SPOOLing系统是虚拟技术和虚拟分配的实现

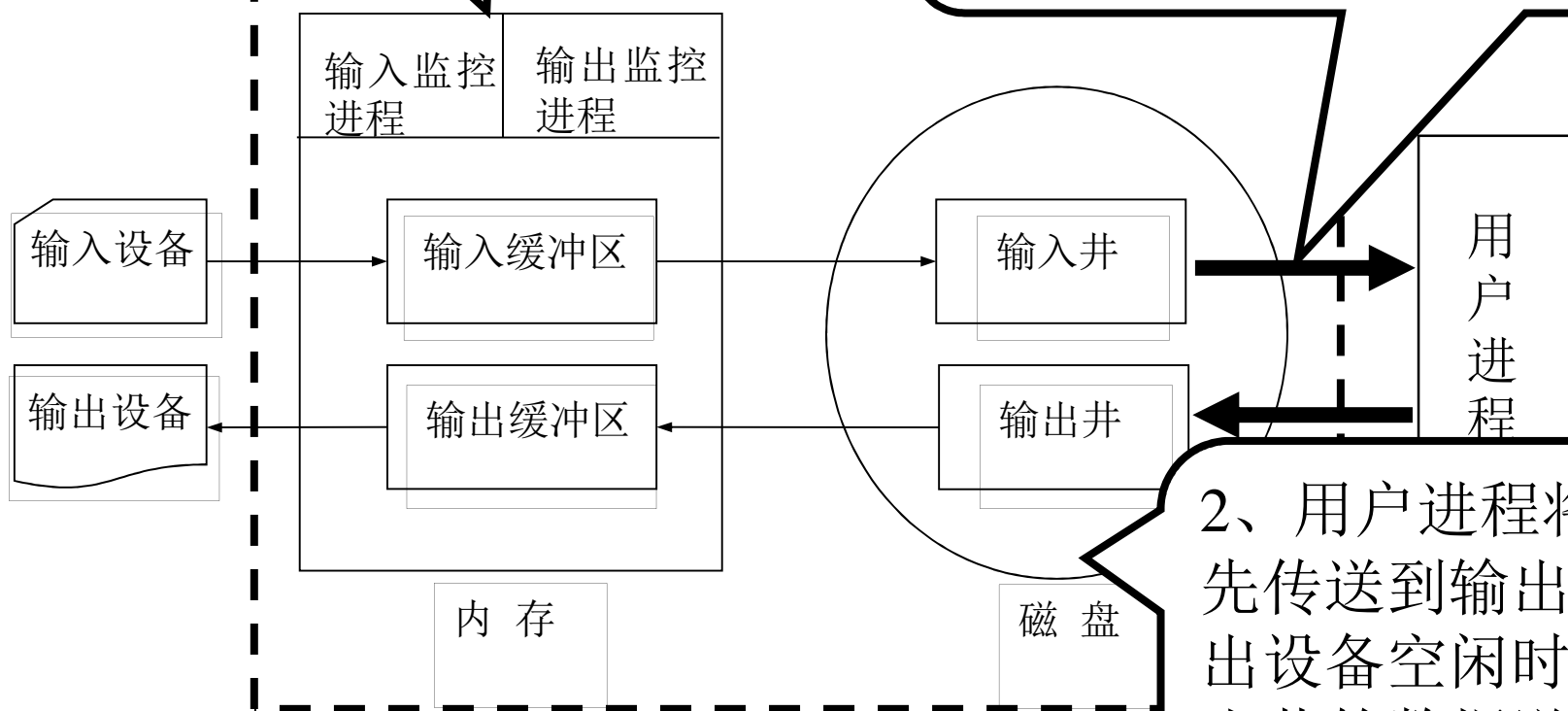
- u Simultaneous Peripheral Operations OnLine

- u 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。（提前读）



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。（延后写）

SPOOLing的结构

I 输入井和输出井

- n 磁盘上开辟的两个存储区域
 - u 输入井模拟脱机输入时的磁盘
 - u 输出井模拟脱机输出时的磁盘

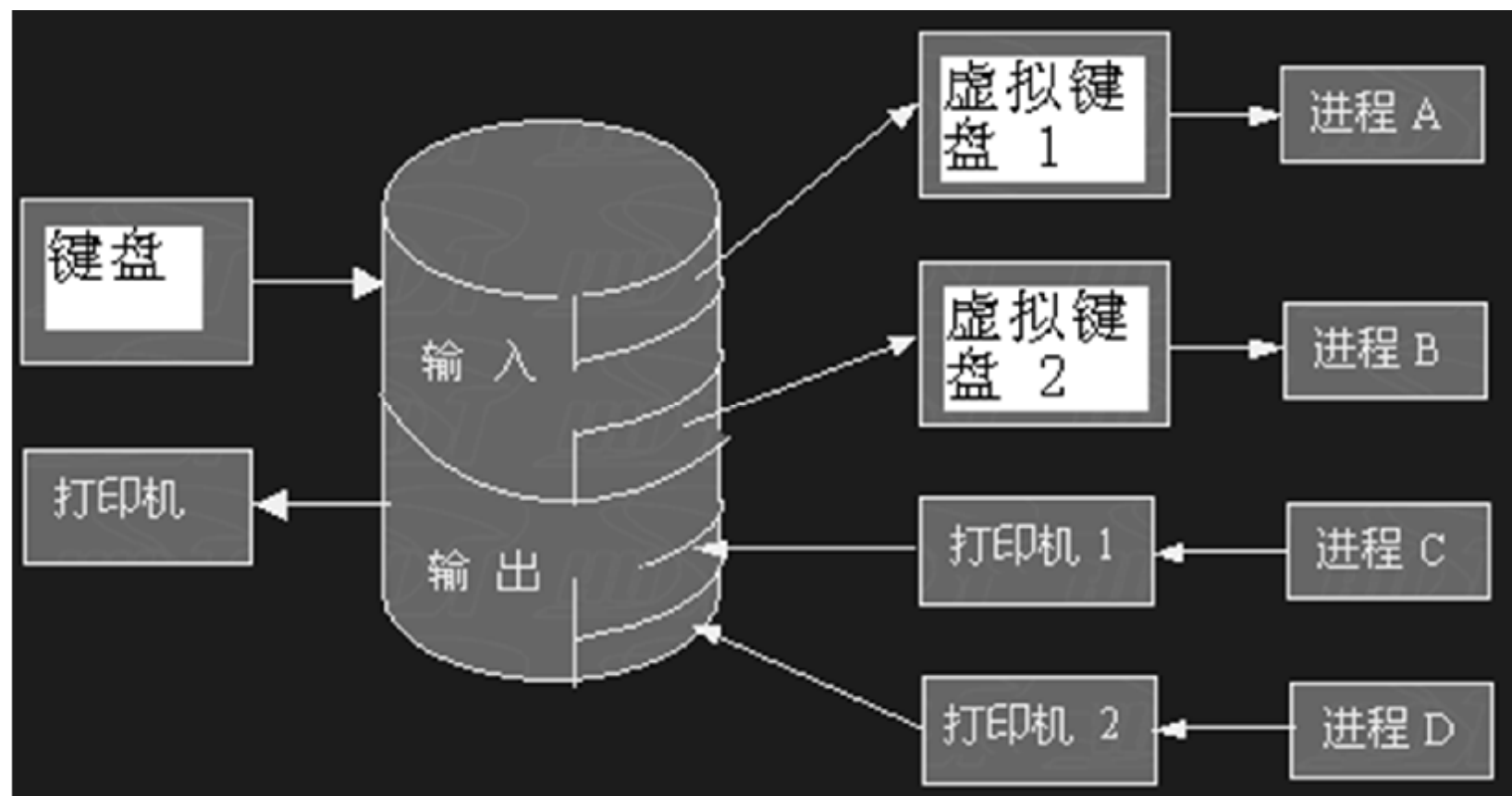
I 输入缓冲区和输出缓冲区

- n 内存中开辟的存储区域
 - u 输入缓冲区：暂存输入数据，以后再传送到输入井。
 - u 输出缓冲区：暂存输出数据，以后再传送到输出设备。

I 输入监控进程和输出监控进程

- n 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；
- n 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

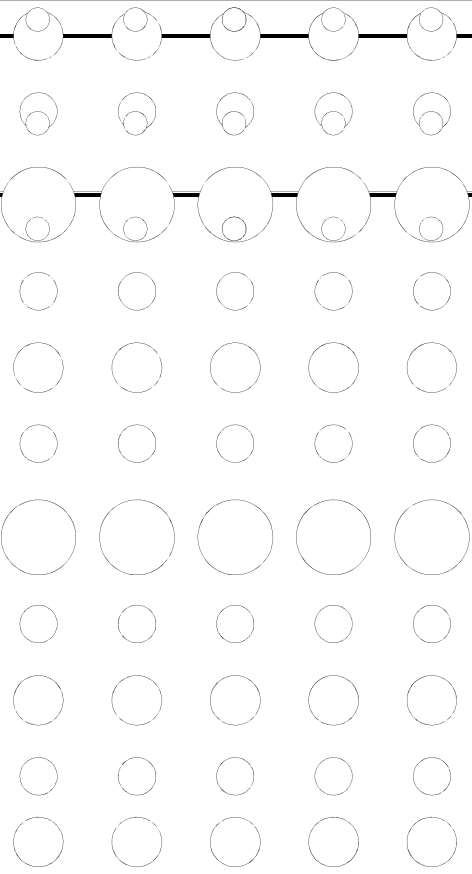
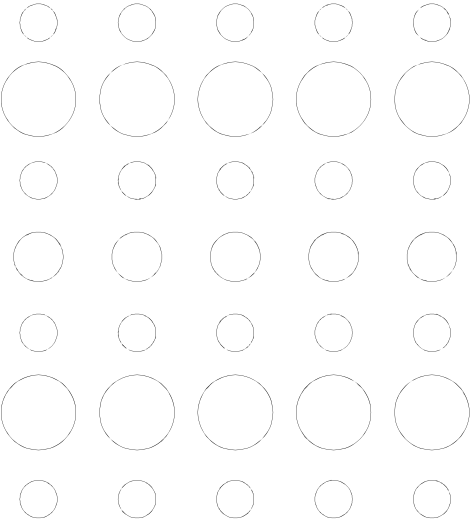


I **SPOOLing**系统原理小结

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

I **SPOOLing**优点

- n “提高”了I/O速度
- n 将独占设备改造为“共享”设备
 - u 实现了虚拟设备功能



4. I/O控制

I I/O数据传输机制

- n 查询方式（异步传送）

- n 无条件传送方式（同步传送）

- n 中断方式

- n 通道方式

- n DMA方式

查询方式（异步传送）

I 基本原理

n 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。

n 输入时：外设数据“准备好”；

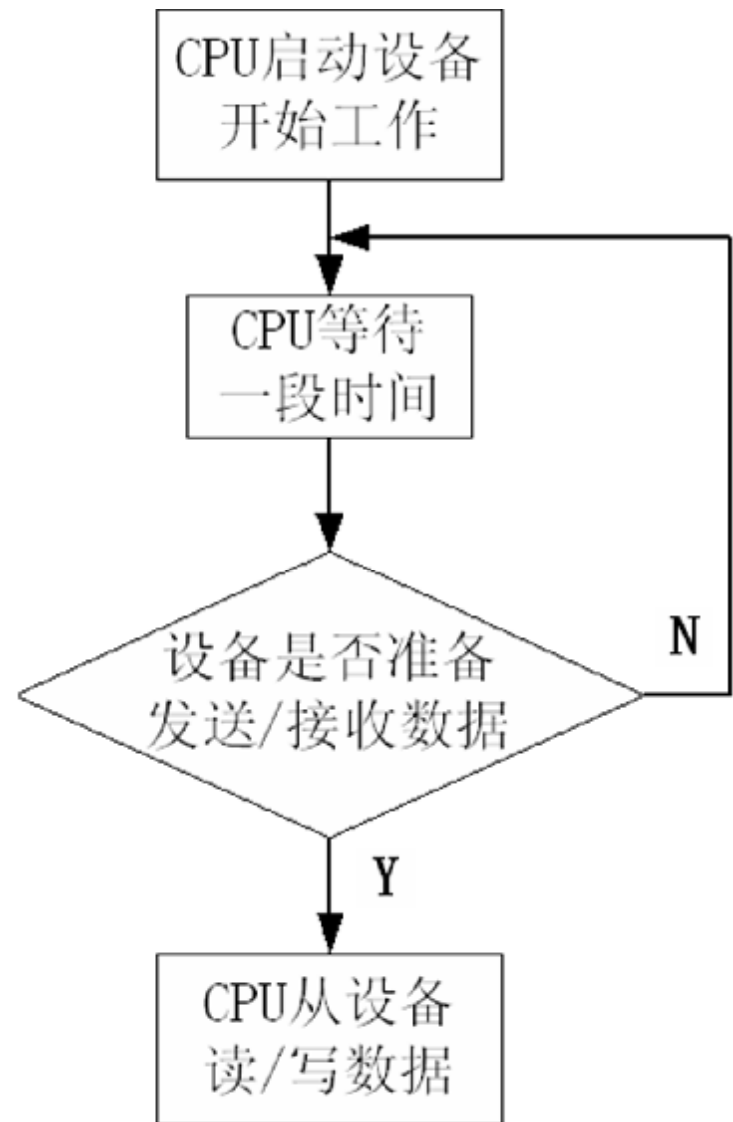
n 输出时：外设“准备好”接收。

I 特点

n I/O操作由程序发起并等待完成

u 指令：IN / OUT

n 每次读写必须通过CPU。



无条件传送（同步传送）

I 工作过程

n进行I/O时无需查询外设状态，直接进行。

n主要用于外设时钟固定而且已知的场合。

n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

I 工作原理

n 外设数据准备好或准备好接收时，产生中断信号

n 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。

n CPU处理完毕后继续原来工作。

I 特点和缺点

n CPU和外设并行工作

n CPU效率提高

n 设备较多时中断频繁，影响CPU的有效计算能力。

n CPU数据吞吐小（几个字节），适于低速设备。

通道方式

I 概念

- n 通道是用来控制外设与内存数据传输的专门部件。

- n 通道有独立的指令系统，既能受控于CPU又能独立于CPU。

- n I/O处理机

I 特点

- n 有很强I/O能力，提高CPU与外设的并行程度

- n 以内存为中心，实现内存与外设直接数据交互。

- n 传输过程基本无需CPU参与。

DMA(直接内存访问)方式

I 概念

- n 外设和内存之间直接进行数据交换，不需CPU干预
- n 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。
- n DMA控制器：DMA Controller（DMAC）
- n DMA的局限
 - u 不能完全脱离CPU
 - p 传送方向，内存始址，数据长度由CPU控制
 - u 每台设备需要一个DMAC
 - p 设备较多时不经济
- n 微机广泛采用



5. 设备驱动

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

I 驱动程序概念

- n 应用程序通过驱动程序来使用硬件设备或底层软件资源。
- n 驱动程序工作在核心层（和OS一个层次）
- n 不同的操作系统提供不同机制来实现驱动程序

I 基本硬件基础

- n 接口/端口
- n 中断机制
- n DMA机制
- n 虚拟驱动

I 驱动开发人员必须具备的基本素质

- n 熟练运用C语言

- n 熟悉多任务、线程、内存管理核心等基本概念

- n 基本的硬件知识

 - u 总线、PCI、USB、寄存器、IO地址、中断、DMA、即插即用...

- n 基本的VC和MFC应用程序开发和调试经验

- n 强烈的求知欲和顽强的毅力

编驱动程序用什么编程语言？

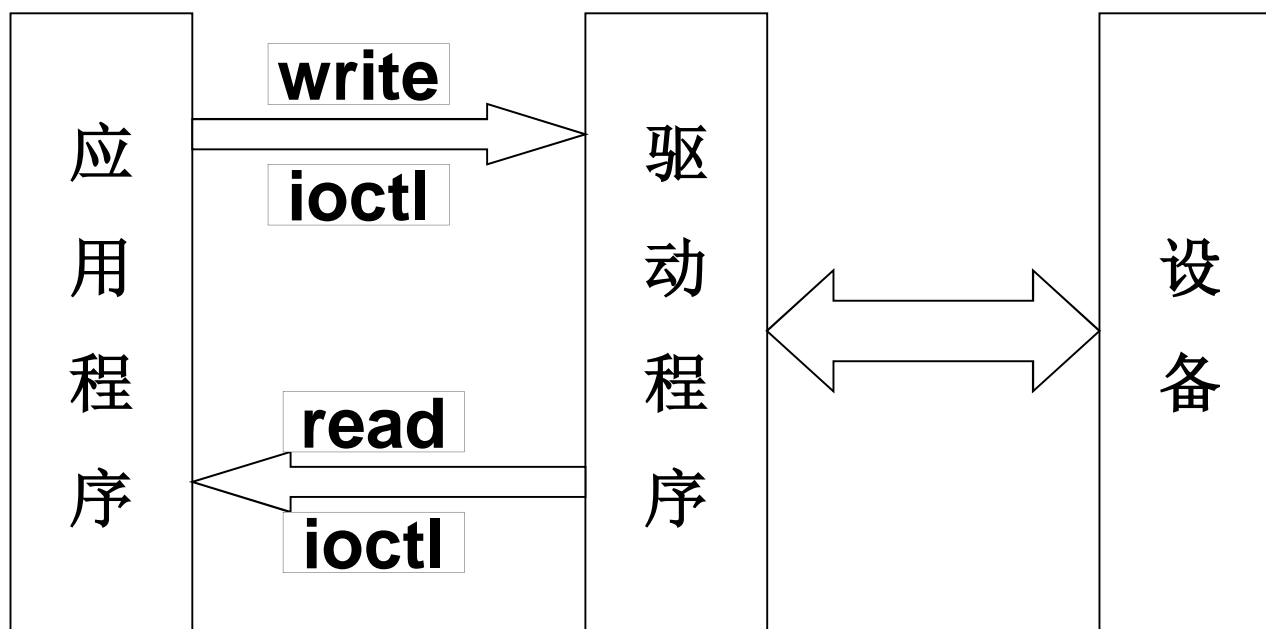
- | **C语言**
- | **C++语言**
- | **%1的情况下会用到汇编语言**
- | **目前还不能用其它高级语言编写驱动程序。**



1 Linux设备驱动(LDD)

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

I LDD程序概念



I 用户态与内核态

nLinux的两种运转模式。

u内核态：

u用户态：

n驱动程序工作在内核态。

n应用程序和驱动程序之间传送数据

uget_user

uput_user

ucopy_from_user

ucopy_to_user

I Linux设备的分类

n 字符设备

- u 以字节为单位逐个进行I/O操作
- u 字符设备中的缓存是可有可无
- u 不支持随机访问
- u 如串口设备

n 块设备

- u 块设备的存取是通过buffer、cache来进行
- u 可以进行随机访问
- u 例如IDE硬盘设备
- u 支持可安装文件系统

n 网络设备

- u 通过BSD套接口访问（SOCKET）

I 设备文件

- n 硬件设备作为文件看待

- n 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

- n 字符设备和块设备通过设备文件访问。

- u 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。

I 命令 ls -l /dev 列出设备文件

```
root@localhost proc1# ls -l /dev | more
total 228
crw----- 1 root    root      10,  10 Jan 30  2003 adbmous
crw-r--r-- 1 root    root     10, 175 Jan 30  2003 agpgart
crw----- 1 root    root      10,   4 Jan 30  2003 amigamou
crw----- 1 root    root      10,   7 Jan 30  2003 amigamou1
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 atarimou
crw----- 1 root    root      10,   3 Jan 30  2003 atibm
crw----- 1 root    root      10,   3 Jan 30  2003 atimou
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
```


I 主设备号和次设备号

n 主设备号

- u 标识该设备种类，标识驱动程序

- u 主设备号的范围：1-255

- u Linux内核支持动态分配主设备号

n 次设备号

- u 标识同一设备驱动程序的不同硬件设备

- u 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

I 功能完整的LDD结构

n 设备的打开

n 设备的释放

n 设备的读操作

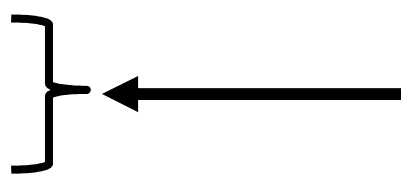
n 设备的写操作

n 设备的控制操作

n 设备的中断和轮询处理

n 驱动程序的注册

n 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

I **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;  
}
```

nMOD_INC_USE_COUNT

nMOD_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

```
1 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);           //LED 开  
close(fd);                       //关
```

设备注册（初始化）

```
I 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, "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

I 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)
```

I 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

I 动态加载

n通过insmod等命令

n调试过程

模块动态加载

I 驱动程序模块插入内核

```
#insmod led.o
```

I 查看是否载入？载入成功会显示设备名my_led

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

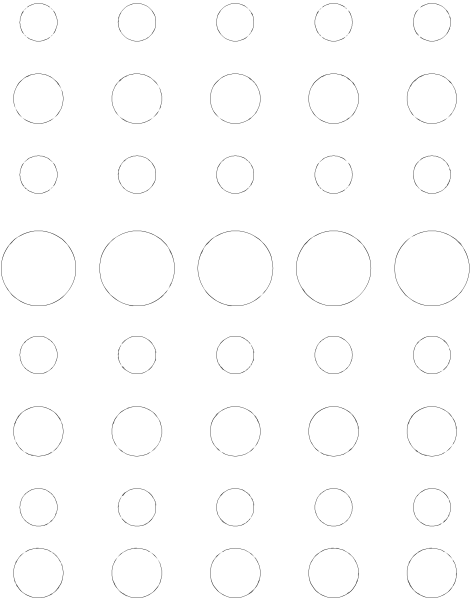
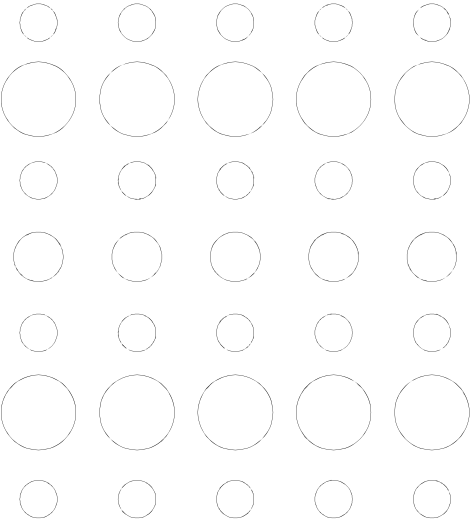
I 从内核移除设备

```
#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;
}
```



2 Windows设备驱动(不作要求)