

嵌入式系统

An Introduction to Embedded System

第九课、嵌入式系统的设备驱动

教师：蔡铭

cm@zju.edu.cn

浙江大学计算机学院人工智能研究所
航天科技—浙江大学基础软件研发中心

课程大纲

 外部设备及设备驱动概述

 VxWorks设备驱动概述

 VxWorks设备驱动代码分析

 Linux设备驱动概述

关于外部设备控制

- 嵌入式系统的几乎每种操作，最后都要映射到实际的物理设备上。
- 除处理器、内存和少数其他实体外，几乎所有设备的控制操作都由设备相关的设备驱动程序来实现。
- 嵌入式系统开发平台功能成熟度的重要指标：是否具有大量、丰富的设备驱动及**BSP**（板级支持包）支持。

设备驱动程序的重要性（1/3）

- 2007年8月，美国Los Angeles国际机场电脑当机超过8小时，17000个航班受影响，20000名乘客滞留机场。
 - 原因：网卡设备驱动程序存在bug，导致网络瘫痪。



设备驱动程序的重要性 (2/3)

设备驱动程序的重要性 (3/3)

□ 串口设备驱动程序的缓冲区溢出bug:

```
#define MaxFrameLen 1500
BYTE recBuf[MaxFrameLen];
WORD pos = 0;

VOID
RS232_12_Interrupt1_HandlerRoutine(
    RS232_12_HANDLE hRS232_12,
    RS232_12_Interrupt1_RESULT *intResult ) {
    recBuf[pos]=RS232_12_ReadByte(hRS232_12,(RS232_12_ADDR)0,0);
    if(++pos>=1000) {
        if (m_pEventToSignal) m_pEventToSignal->Set();
    }
}
```

读取串口数据

通知应用程序读取串口数据

- 为串口数据存储设置的缓冲区为1500字节，RS232的传输速率在9600bps-115200bps左右，如果应用程序在0.104秒-1.25秒之内不读取数据的话，1500字节的缓冲区就会发生溢出，从而导致操作系统内核数据区的数据损坏。

外部设备的分类（1/3）

□ 外部设备主要分为三类

- ✓ 字符设备
- ✓ 块设备
- ✓ 网络设备

□ 字符设备

- ✓ 能够像字节流一样被访问的设备，提供数据通道，不允许来回读写，在数据传输中以字符为单位进行传输。例如：串口、鼠标、终端、打印机

□ 字符设备特点：

- ✓ 在数据传输中，可以传输任意大小的字符数据；
- ✓ 可以通过文件方式（/tyCo/0）访问，但只能顺序访问数据通道；
- ✓ 字符设备中的缓存可有可无。

外部设备的分类（2/3）

□ 块设备

- ✓ 以数据“块”为单位，对数据进行来回读写/存取的设备。1个数据块可包括512B、1KB数据。块设备可以容纳文件系统。
- ✓ 块设备包括：FLASH、硬盘、光驱、软驱等。

□ 块设备特点：

- ✓ 在数据传输中，传输单位是固定大小的数据块。
- ✓ 块设备的存取是通过buffer、cache来进行。
- ✓ 可以随机访问块设备中存放的数据块。
- ✓ 块设备不直接与I/O系统连接，而是通过与文件系统关联，提供接口。

外部设备的分类（3/3）

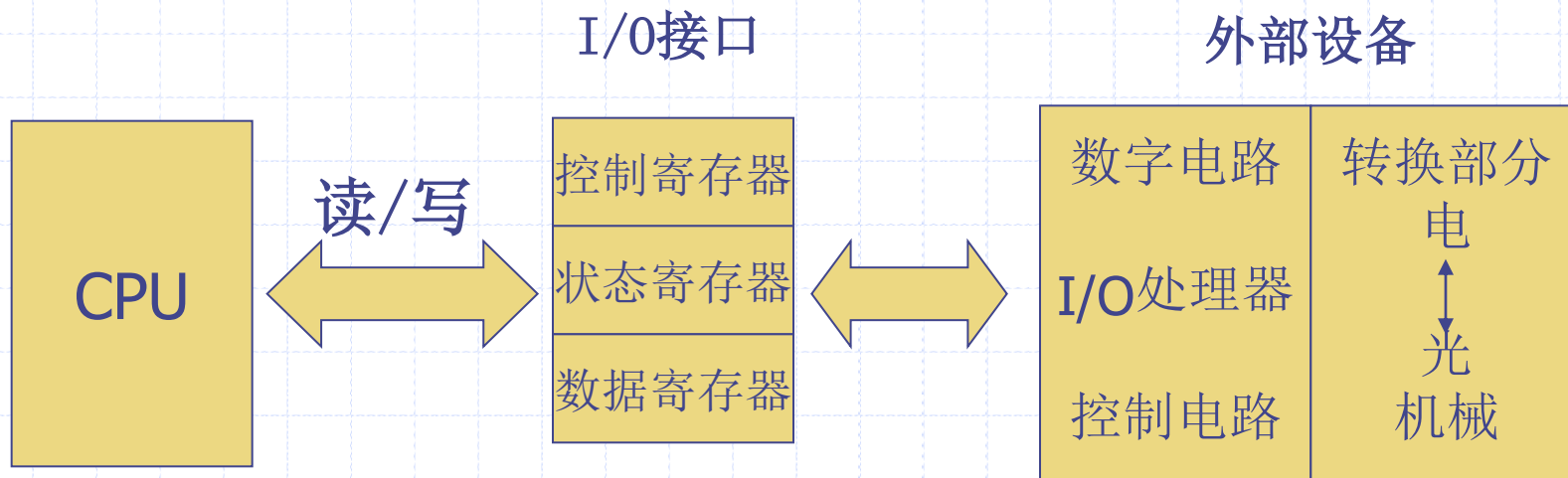
□ 网络设备

- ✓ 能够与其他主机进行网络通讯的设备。
- ✓ 与普通I/O设备不同，网络设备没有对应的设备文件，数据通信不是基于标准的I/O系统接口，而是基于BSD套接口访问，例如：`socket`、`bind`、`listen`、`accept`、`send`等系统调用。

外部设备与CPU的信息交换

□ 外部设备与CPU的通讯信息，主要包括：

- ✓ 控制信息：告诉要进行什么处理
- ✓ 状态信息：当前设备的状态
- ✓ 数据信息：不同的设备，传输数据类型及编码各异



对外部设备的输入、输出方式控制

□ 与外部设备的输入/输出的方式，主要包括：

- ✓ 直接控制I/O方式：通过指令直接对端口进行输入、输出控制。
如x86的in、out指令。
- ✓ 内存映射方式：可以访问较大的地址空间，实现快速数据交换，
如：ISA总线可以映射的空间为0xC8000~0xEFFF。
- ✓ 中断方式：采用中断实现数据的输入/输出。
- ✓ DMA方式：采用DMA控制器，实现数据传输。

□ 讨论：

- ✓ 数据量不大的情况下，如字符设备，常采用中断方式；
- ✓ 大量数据传输的I/O设备，如磁盘、网卡等设备，常采用DMA方式，以减少CPU资源占用。

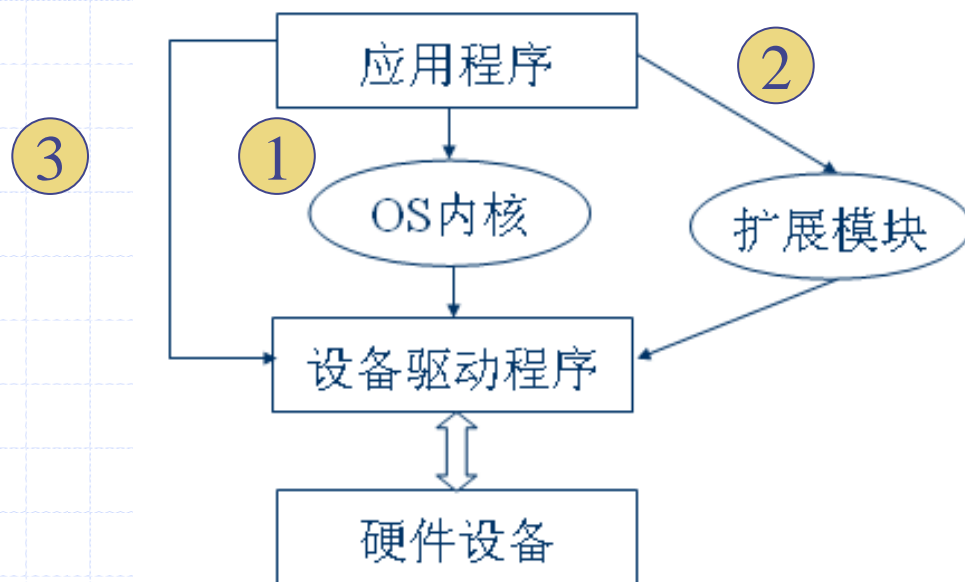
设备驱动程序概述

- ❑ 设备驱动程序，是直接控制设备操作的那部分程序，是设备上层的一个软件接口。
- ❑ 实际上从软件角度来说，设备驱动程序就是负责完成对I/O端口地址进行读、写操作。
- ❑ 设备驱动程序的功能是对I/O进行操作。
- ❑ 设备驱动程序不能自动执行，只能被操作系统，或应用程序调用。

嵌入式系统调用设备驱动的方式

□ 嵌入式系统中调用设备驱动程序的三种方式：

- ✓ 应用程序直接调用
- ✓ 应用程序通过操作系统内核调用
- ✓ 应用程序通过操作系统扩展模块进行调用



设备驱动程序的主要功能

□ 设备驱动程序的主要功能包括以下6项:

- ✓ 对设备进行初始化(create)
- ✓ 打开设备操作(open)
- ✓ 关闭设备操作(close)
- ✓ 从设备上接收数据，并提交给系统一读操作(read)
- ✓ 把数据从主机上发送给设备一写操作(write)
- ✓ 对设备进行控制操作(ioctl)

设备驱动程序的相关概念（1/3）

□ 设备驱动程序表：

- ✓ 应用程序对设备进行操作是通过操作系统的I/O管理子系统来进行的。
- ✓ I/O管理子系统对设备的管理，是通过“设备驱动程序表”进行的。
- ✓ “设备驱动程序表”描述了系统中所有设备驱动程序，以及设备驱动程序所支持操作（open/read/write/ioctl/close）的入口地址。

设备驱动程序的相关概念（2/3）

□ 设备的标识采用两种方式：

- 主、从设备号（系统标识法）
 - ✓ 主设备号：标识该设备的种类，也标识了该设备所使用的驱动程序
 - ✓ 从设备号：标识使用同一设备驱动程序的不同硬件设备
- 设备名称（外部标识法）
 - ✓ I/O管理子系统提供对设备进行命名的接口，以提高设备管理的直观性。

设备驱动程序的相关概念（3/3）

□ 设备驱动程序 vs BSP的区别与联系：

● BSP（Board Support Package）—板级支持包

- ✓ 针对具体目标机平台，所编写的与硬件结构相关的代码部分。
- ✓ 主要完成系统上加电后CPU初始化、各类控制芯片初始化等工作。
- ✓ BSP与bootloader、设备驱动程序关系密切。

● 设备驱动程序

- ✓ 与硬件结构相关的部分，包含在BSP中
- ✓ 通用部分，可以在其他部分中实现

课程大纲

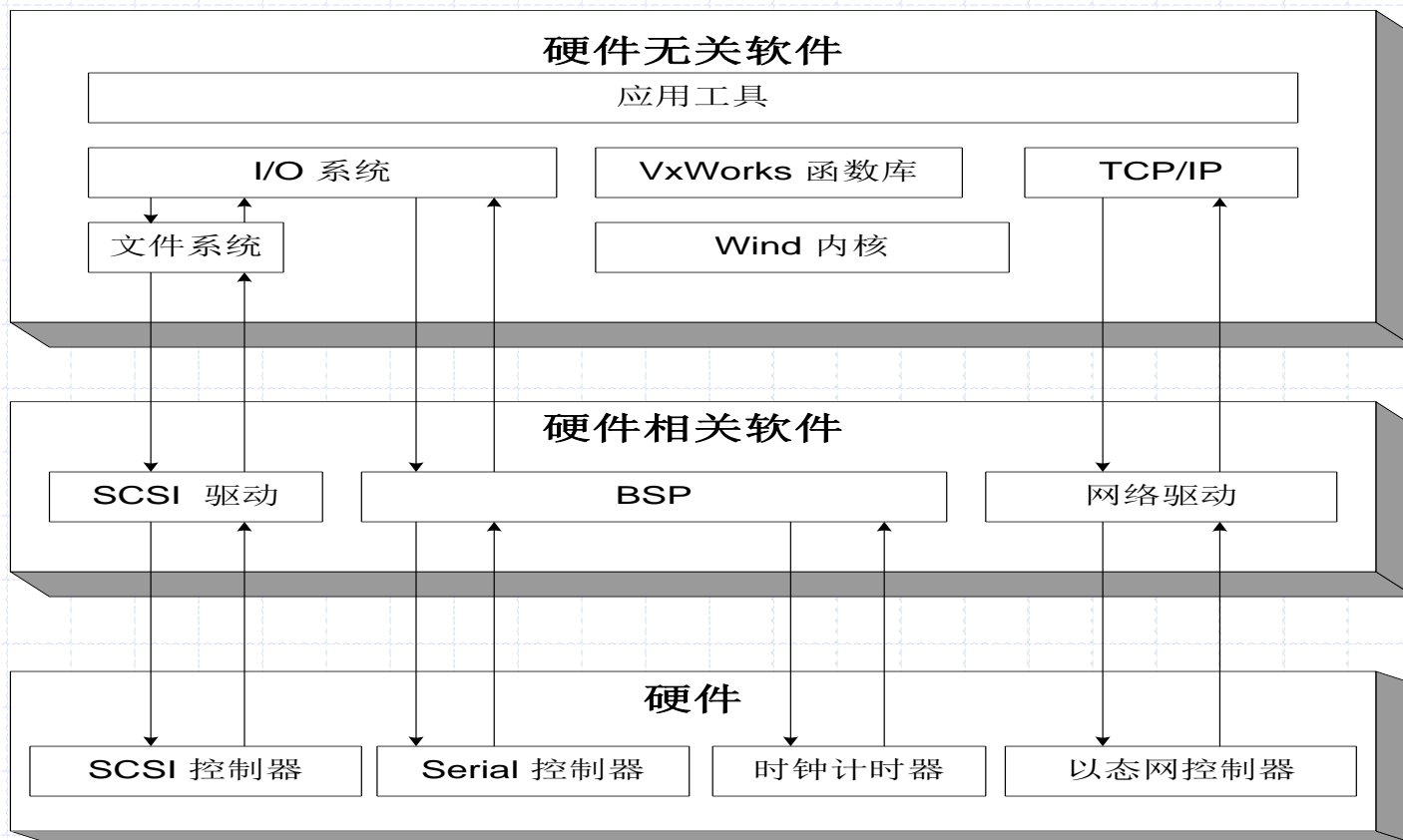
 外部设备及设备驱动概述

 VxWorks设备驱动概述

 VxWorks设备驱动代码分析

 Linux设备驱动概述

VxWorks操作系统组成

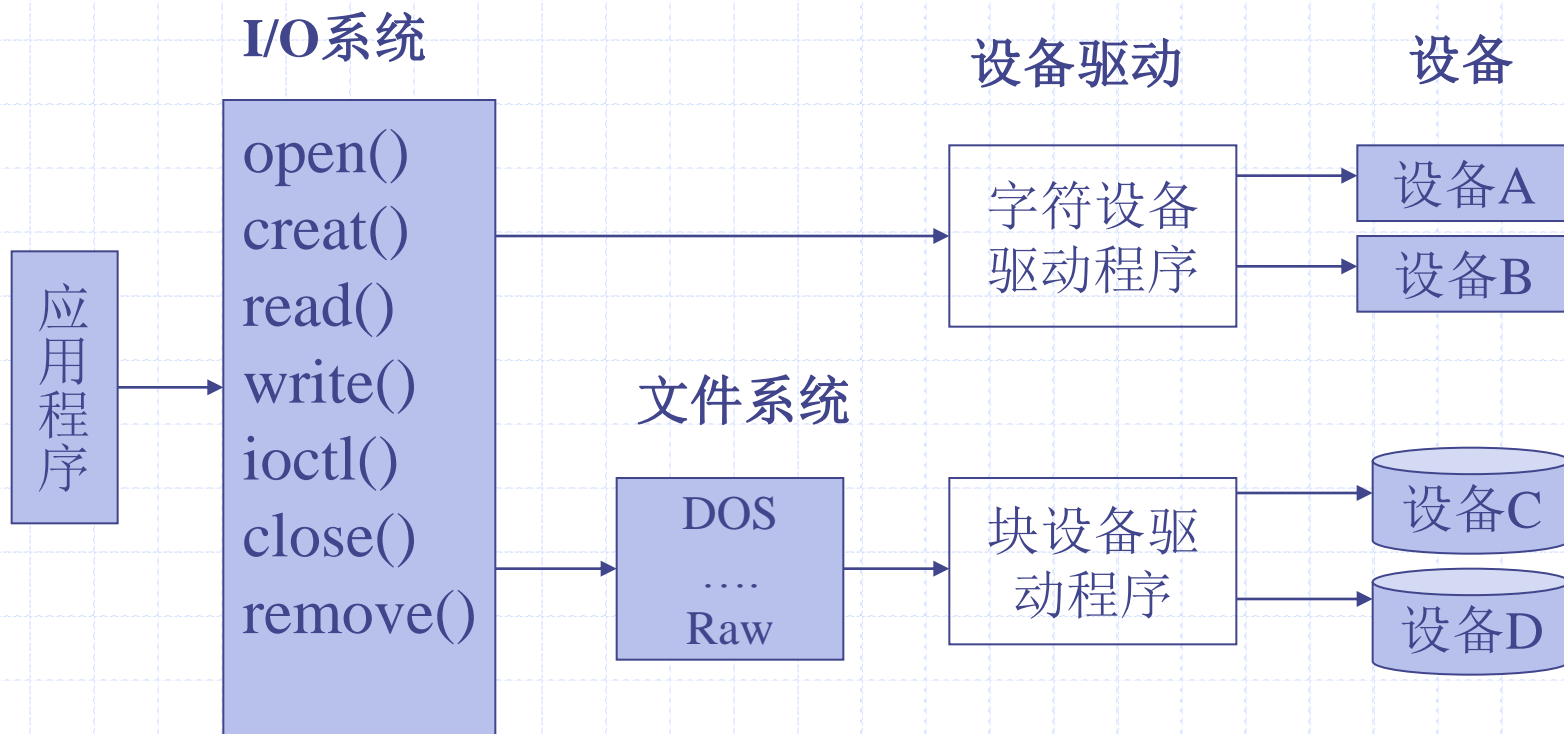


VxWorks体系结构

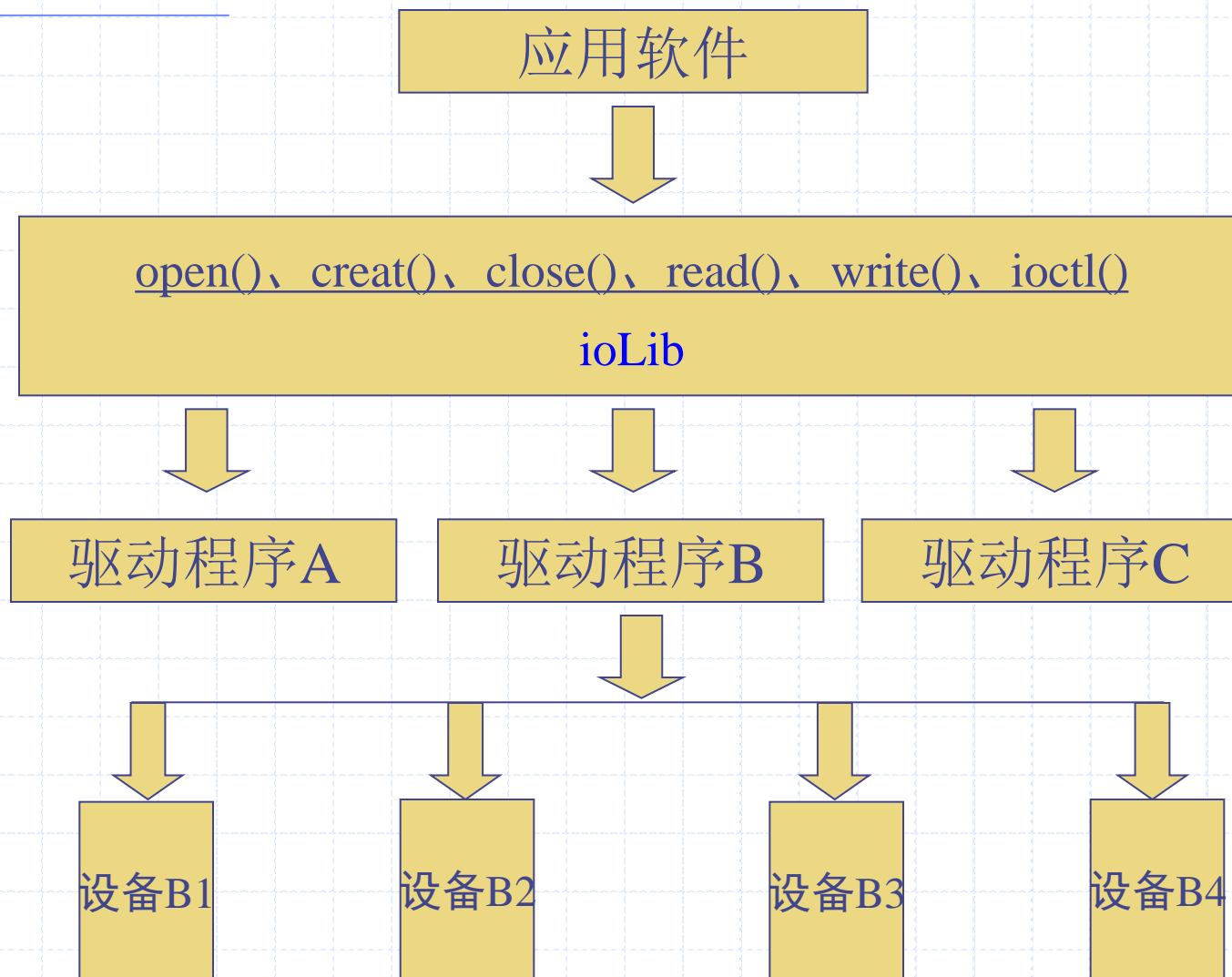
VxWorks的BSP及设备驱动支持

- VxWorks在板级软件上提供了BSP，解除了操作系统与硬件的耦合，系统移植时，只需要修改BSP。
- 在BSP提供上，WindRiver与Intel、Motorola等主流芯片提供商建立了紧密的合作关系，支持x86、i960、Sun Sparc、MIPS、PPC等，提供了丰富的BSP开发包及驱动程序源码。

VxWorks的I/O系统与设备驱动



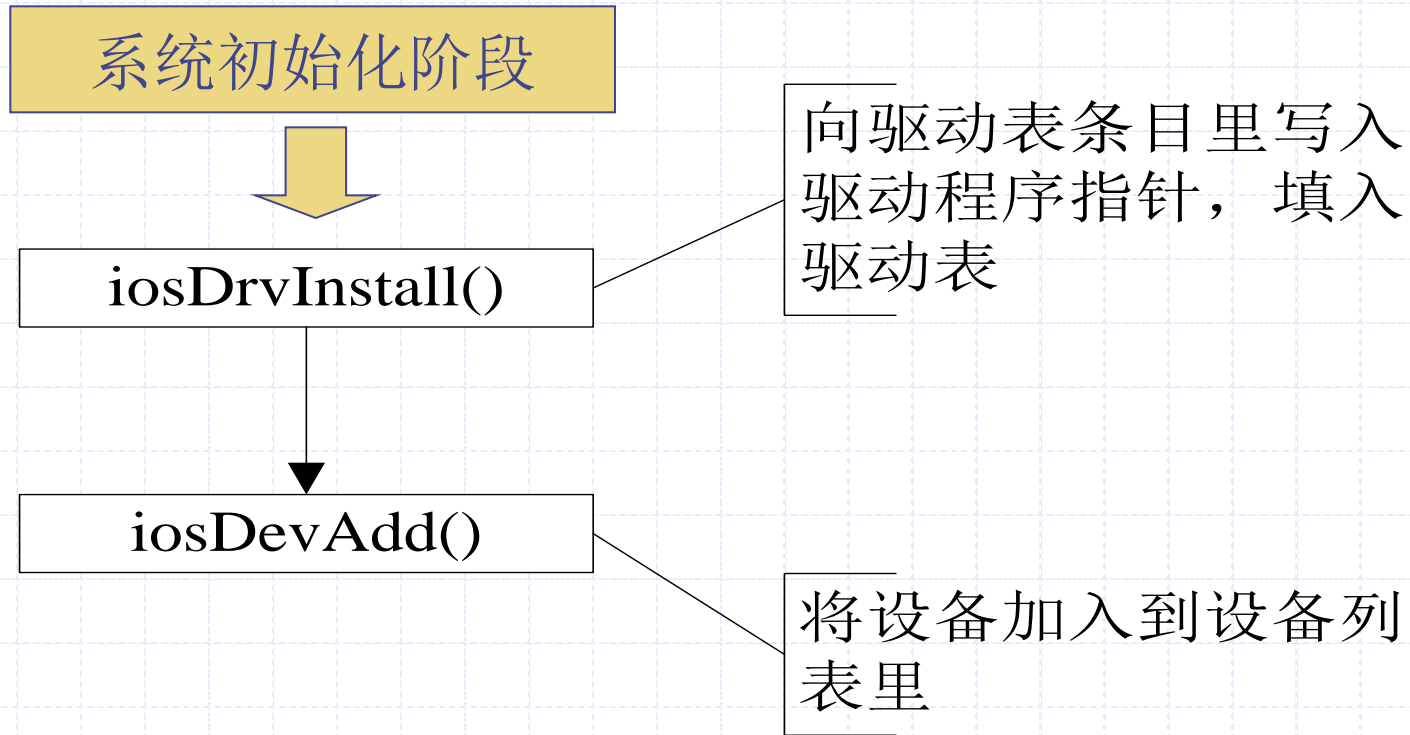
VxWorks的I/O系统结构



VxWorks的I/O系统对设备管理的支持

- 在VxWorks的I/O系统中包含三张表，对设备进行管理
 - ✓ 驱动程序描述表
 - ✓ 设备列表
 - ✓ 文件描述符表
- 通过VxWorks提供的函数，在设备列表和驱动程序描述表中，注册和卸载设备及设备驱动程序。

VxWorks中安装驱动程序及设备



设备驱动程序表

添加设备驱动，调用iosDrvInstall，例如

xxDrvNum = iosDrvInstall(xxOpen, NULL, xxOpen,
xxClose, xxRead, xxWrite, xxIoctl)

驱动程序
索引号

驱动程序 索引号	creat	remove	open	close	read	write	ioctl
0
1	xxOpen	NULL	xxOpen	xxClose	xxRead	xxWrite	xxIoctl
2

驱动程序表

设备驱动程序表显示

可以在系统运行中，使用iosDrvShow()函数来显示驱动程序描述表

--> isoDrvShow

Drv	create	remove	open	close	read	write	ioctl
1	421db4	0	421db4	421ddc	42b76c	42b69c	421e08
2	0	0	424fd4	0	425004	425044	425130
3	415f40	0	415f40	416000	42b76c	42b69c	416074

设备描述符

为每个新添加的设备提供一个设备描述符，其中，必须包括一个DEV_HDR的结构，例如：

```
typedef struct _MOXA_DEV  
{
```

```
    DEV_HDR DevHdr;
```

```
    BOOL    created;
```

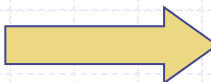
```
    int      portNum;
```

```
    RING_ID recvRngBuf;
```

```
    SEMAPHORE wmuteSem; /* write semaphore among writers */
```

```
    SEMAPHORE rmuteSem; /* read semaphore among readers*/
```

```
}MOXA_DEV;
```

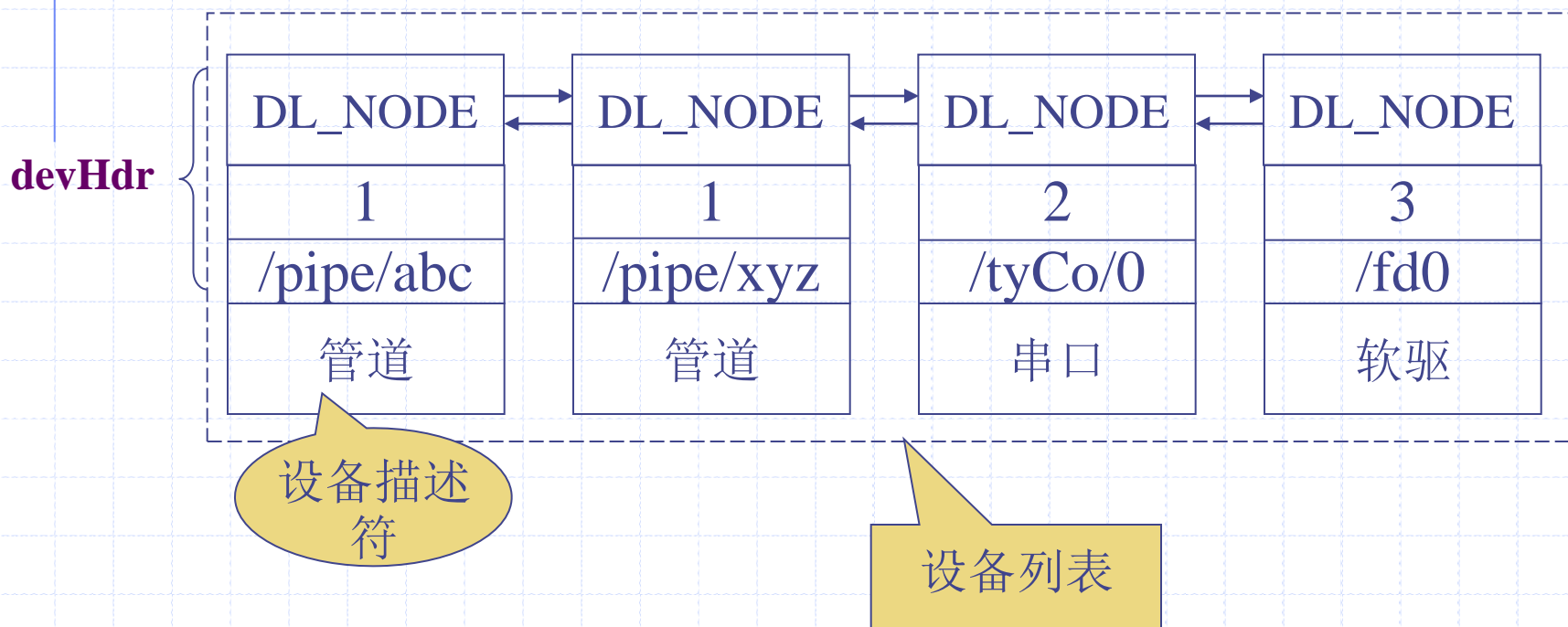


```
typedef struct  
{  
    DL_NODE  node;    /* list node      */  
    short    drvNum;   /* driver number */  
    char *   name;    /* device name  */  
} DEV_HDR;
```

设备列表

调用**iosDevAdd**，将新设备添加至设备列表中，形成一系列设备描述符的双向链表，例如

iosDevAdd(&pxxDev->devHdr, devName, xxDrvNum)



文件描述符表

文件描述符表 (调用open(ioLib.c)添加)

文件描述符	驱动程序索引号	驱动程序指定的设备号
3	2	0
4	5	0
5	5	1

文件描述符表是I/O系统将文件描述符与驱动程序、设备对应起来的手段

文件描述符、设备驱动、设备关联举例

- 使用下面的语句打开一个设备：

```
fd = open("/Dev/0", O-READ, 0);
```

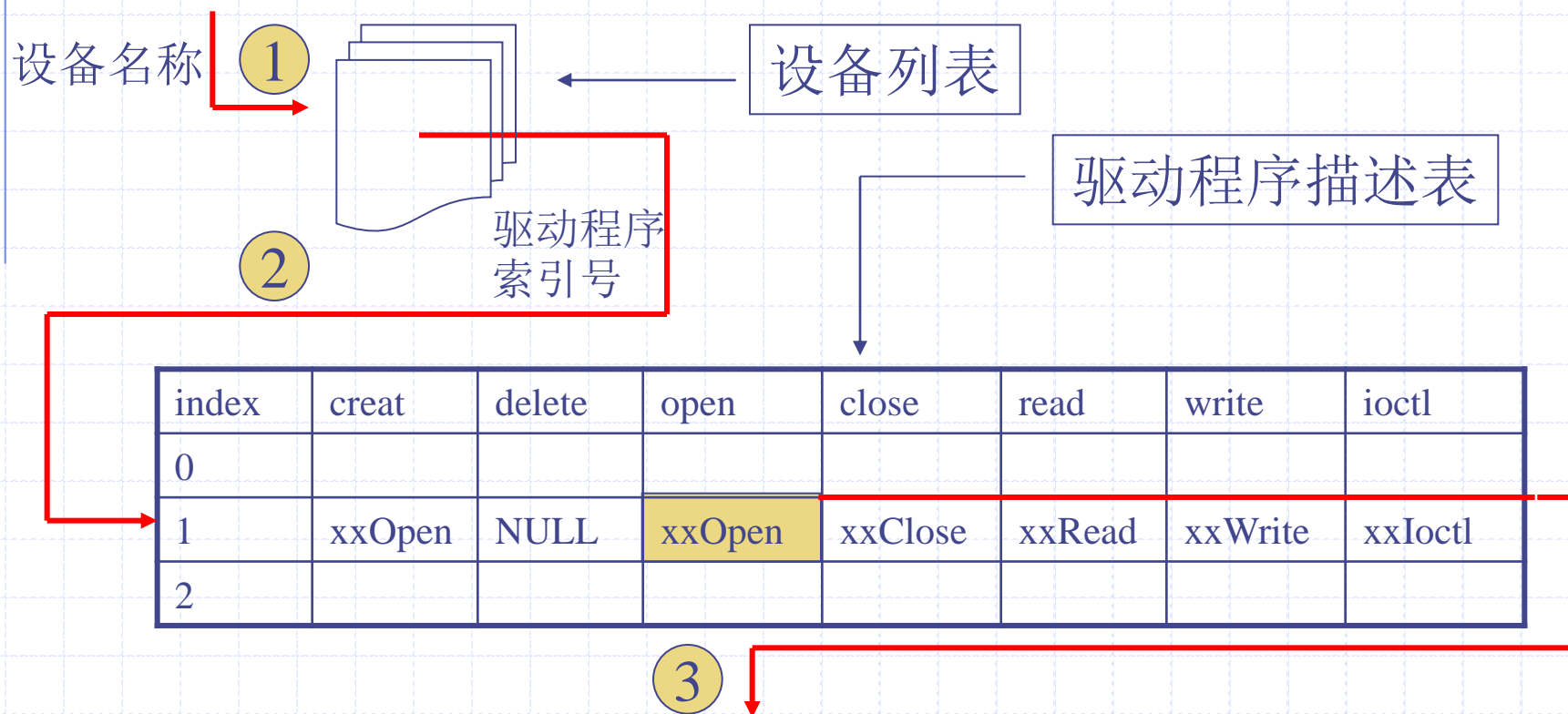
- fd是open()函数返回的文件描述符，/Dev/0是文件名，O-READ是文件打开方式，0是文件打开模式。

- VxWorks按如下方式进行操作：

- ✓ 在设备列表中查找最匹配文件名“/Dev/0”的设备；
- ✓ 提取设备描述符drvHdr中的驱动程序索引号；
- ✓ 在驱动程序描述表中，查找相应的驱动程序入口；
- ✓ 记录文件描述符、设备号，及驱动程序索引号关系。

VxWorks下打开设备的操作

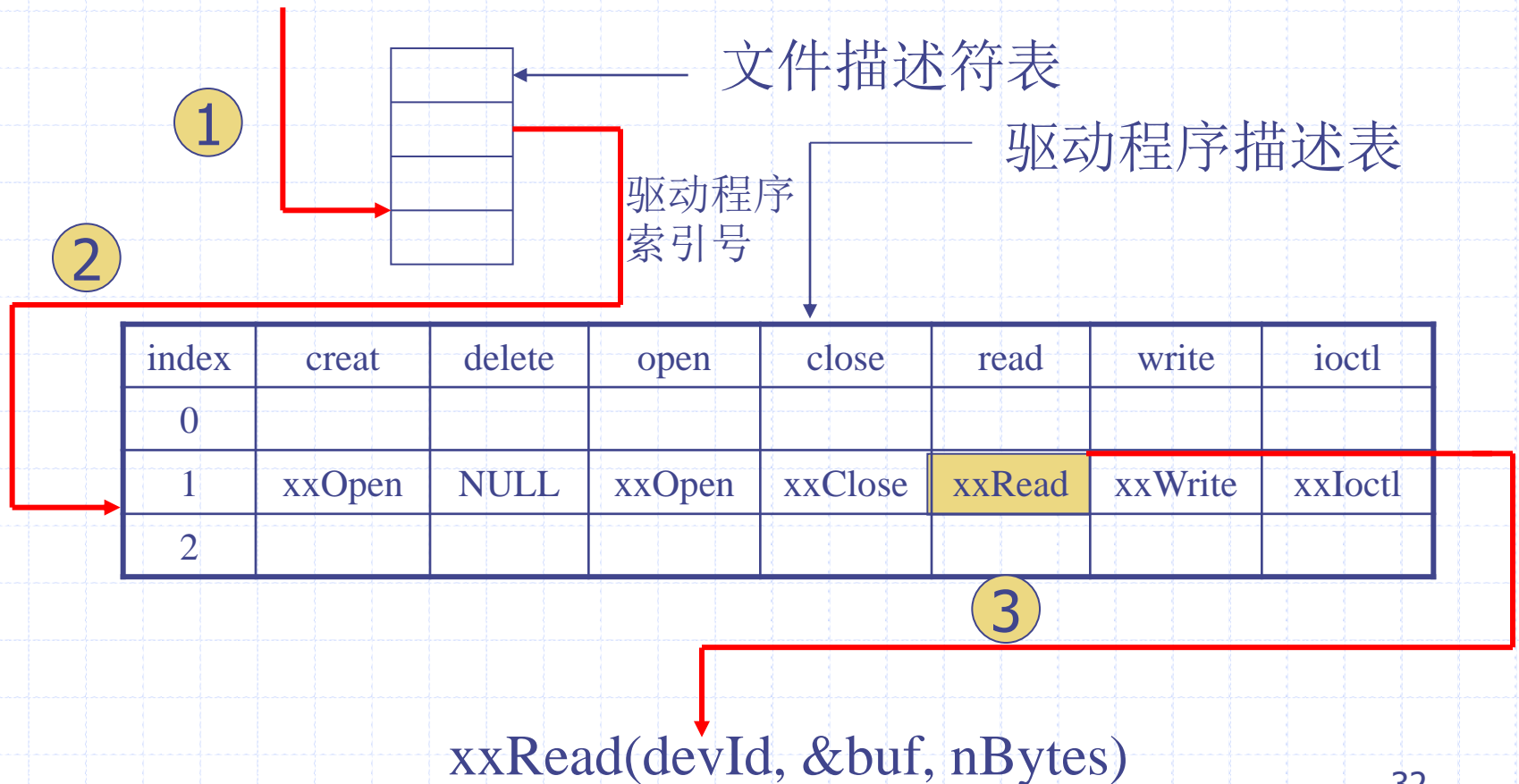
```
fd = open("/xxDevice", O-READ, 0)
```



```
xxOpen(pDevHdr, O-READ, 0)
```

VxWorks下读取设备的操作

`read(fd, &buf, nBytes)`



设备驱动程序的轮询和中断处理

- ❑ 设备驱动程序直接控制设备，需要具有测试设备状态的功能，通常采用两种方式获得这些状态：
 - ✓ 轮询方式
 - ✓ 中断处理方式
- ❑ 轮询(polling)方式：是指软件周期性查询硬件事件的发生。这种方式的实时性较差。
- ❑ 中断处理方式：当事件发生或状态改变时，硬件产生中断，然后执行相应的中断服务程序进行处理。这种方式的实时性较好，嵌入式系统一般采用。

VxWorks的中断服务程序安装

- 安装中断服务程序：用户可以用指定的程序作为中断服务程序。
- VxWorks提供了intConnect，以及pciIntConnect，实现普通的中断服务程序安装，及PCI总线的中断服务程序安装。
例如：

- ✓ 时钟中断服务程序安装：

intConnect(INUM_TO_IVEC(0x20),sysClkInt,0)



```
#define INUM_TO_IVEC(intNum) \
    ((VOIDFUNCPTR *) ((intNum) << 3))
```

中断服务程序开发中的限制（1/2）

□ 中断服务程序中的约束操作

引起这些限制是因为中断服务程序不在一个固定的任务上下文中执行，没有任务控制块，因而，一旦阻塞执行后系统就会出现错误或挂起。

因而，中断服务程序中禁止使用会引起阻塞的函数。

中断服务程序开发中的限制（2/2）

□ 中断服务程序的约束主要在下面几方面：

- ✓ 不可以调用I/O系统调用：例如printf()，因为I/O设备驱动程序会阻塞等待设备的调用者。
- ✓ 不可以进行阻塞操作：不可以获得信号量操作或调用任何可能需要获得信号量的函数，例如semTake、free、malloc。
- ✓ 不可以做请求任务上下文的操作：如不能够使用任务延迟taskDelay。





一个问题：

VxWorks、uc/os II中，均对ISR中调用任务延迟做了限制，但未对任务挂起操作做限制，为什么？

■ taskSuspend（VxWorks）

■ OSTaskSuspend（uc/os II）

课程大纲

-  外部设备及设备驱动概述
-  VxWorks设备驱动概述
-  VxWorks设备驱动代码分析
-  Linux设备驱动概述

设备驱动程序设计步骤

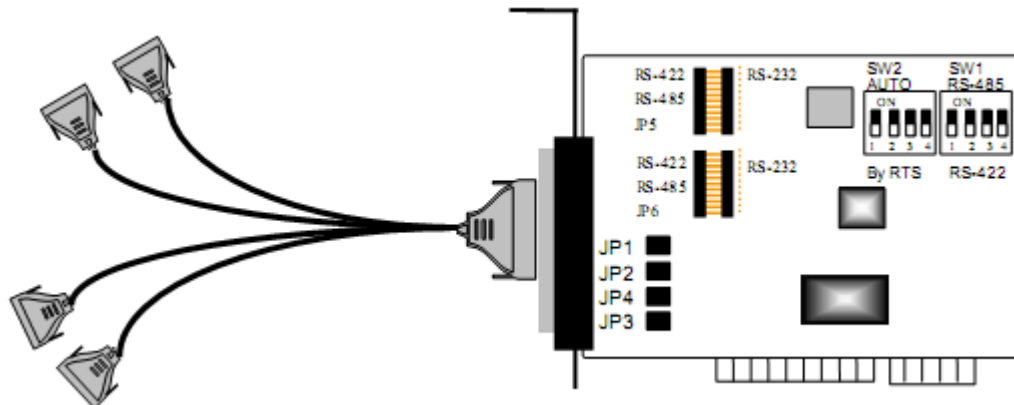
- 熟悉硬件设备、了解它的工作原理及功能；
- 设计设备驱动程序框架结构（参考驱动模板）；
- 设计具体功能函数；
- 进行功能、性能测试。

设备驱动实例—多功能串口接口卡(1/2)

□ MOXA公司的CP-114是一款多功能串口卡。

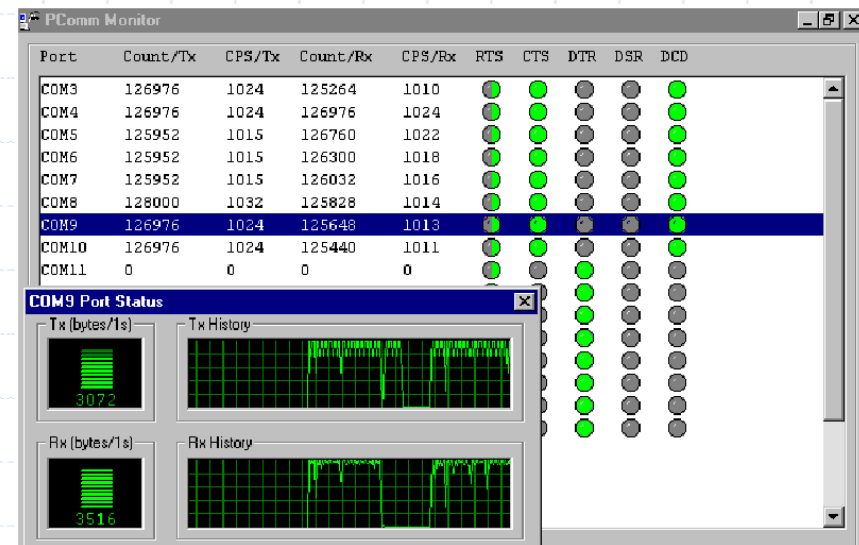
□ 特点：

- ✓ 具有4个通讯串口（RS232/RS422/RS485）
- ✓ Universal PCI插槽，兼容3.3/5V PCI和PCI-X
- ✓ 串口通信速率高达921.6Kbps
- ✓ 数据吞吐量高达800Kbps
- ✓ 具备128字节FIFO和板载硬件、软件流控



设备驱动实例—多功能串口接口卡(2/2)

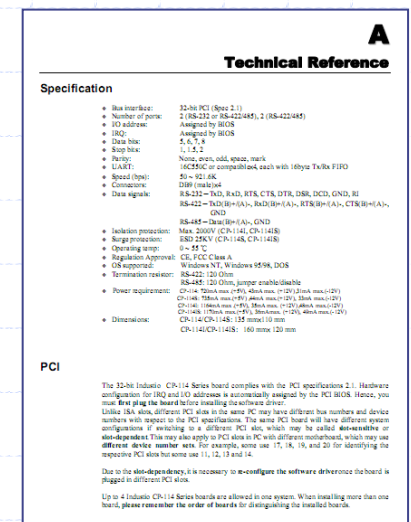
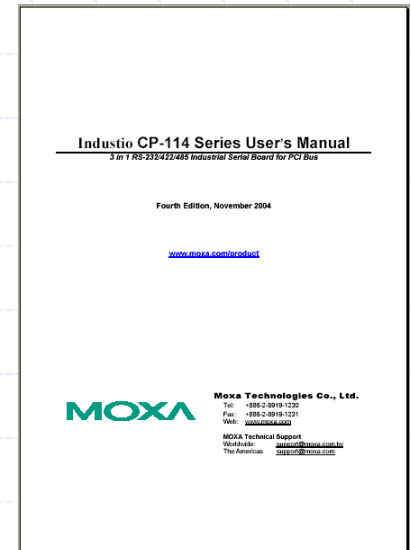
- ❑ MOXA公司提供了DOS、Windows9x/NT/2K/XP、Linux2.4、FreeBSD的设备驱动，缺乏VxWorks上的设备驱动支持。
- ❑ 希望在VxWorks中提供用户如下使用方式的标准接口：
 - ✓ `fd = open("/moxa/0", O_RDWR, 0)`
 - ✓ `read(fd, &buf, nBytes)`
 - ✓ `write(fd, &buf, nBytes)`



设备驱动实例—MOXA卡原理熟悉(1/2)

□ Technical Reference

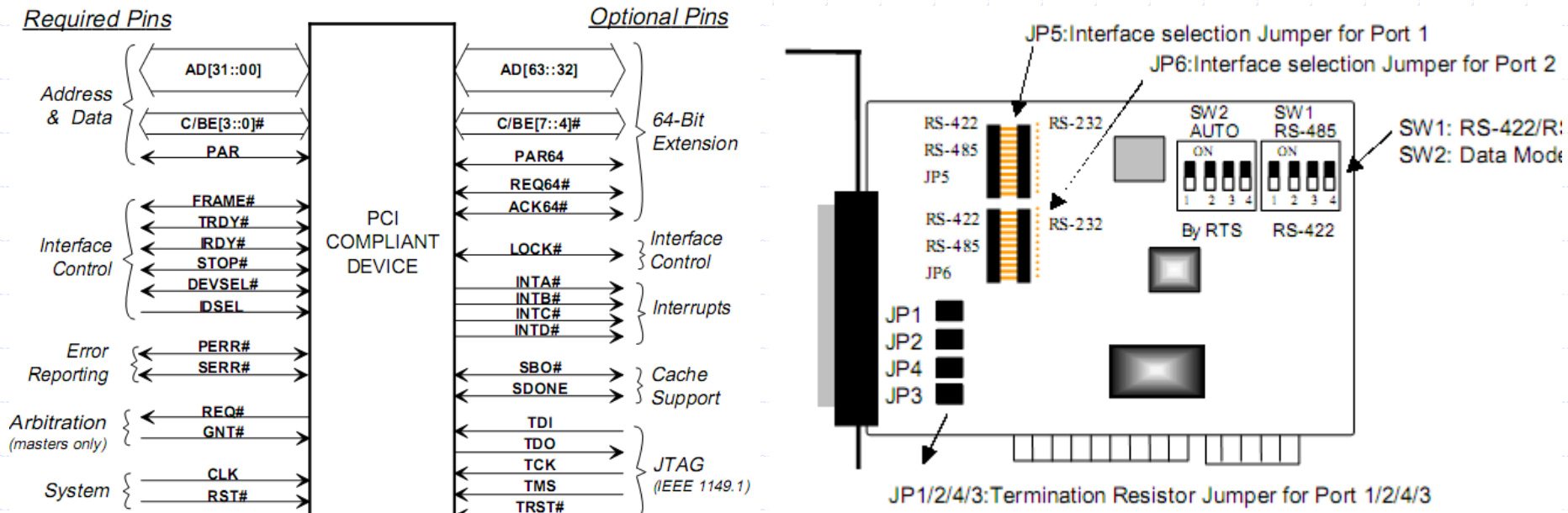
- ✓ **Bus interface: 32-bit PCI (Spec 2.1)**
- ✓ **Number of ports: 2 (RS-232/422/485), 2(RS-422/485)**
- ✓ **I/O address: Assigned by BIOS**
- ✓ **IRQ: Assigned by BIOS**
- ✓ **Data bits: 5, 6, 7, 8**
- ✓ **Stop bits: 1, 1.5, 2**
- ✓ **UART: 16C550C or compatible $\times 4$, each with 16byte Tx/Rx FIFO**
- ✓ **Speed (bps): 50 ~ 921.6K**



设备驱动实例—MOXA卡原理熟悉(2/2)

□ 需要熟悉:

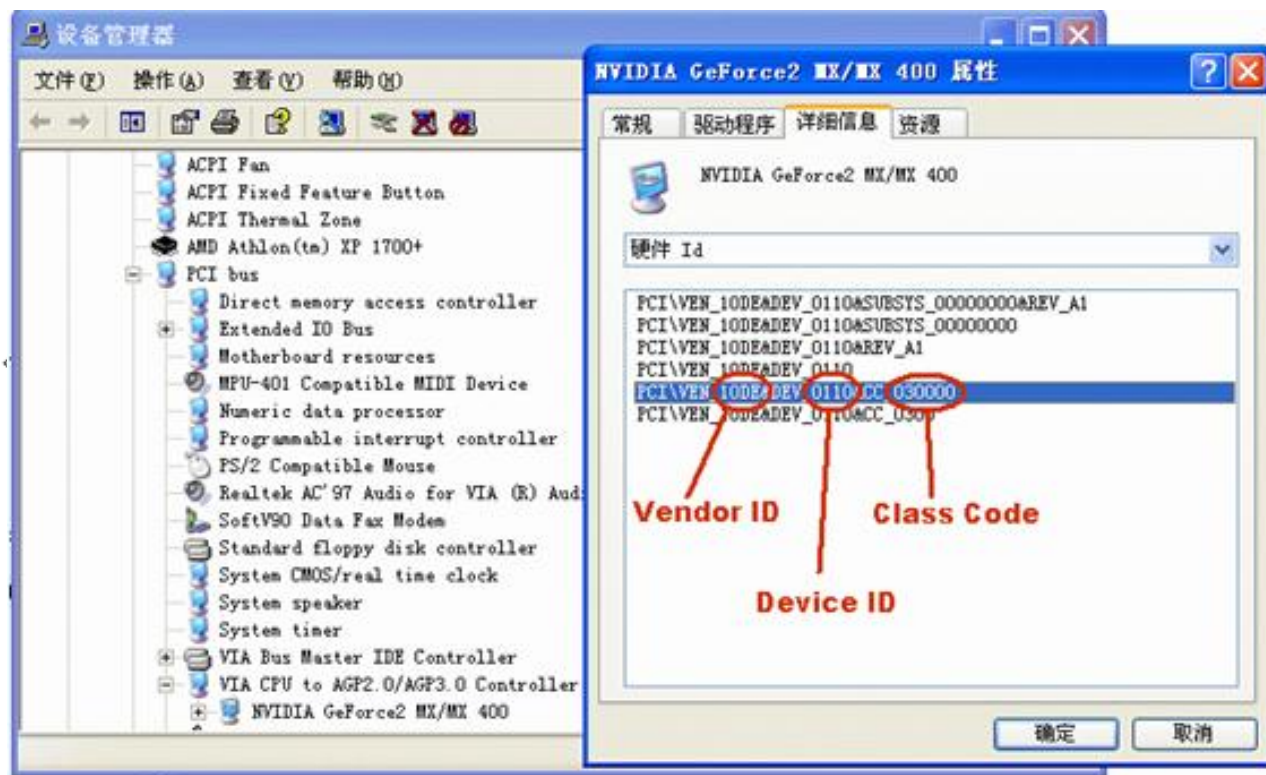
- ✓ PCI接口编程（驱动模板）
- ✓ 字符设备驱动编程（驱动模板）
- ✓ RS232/RS422/RS485基本原理



设备驱动框架

- ❑ 在VxWorks系统启动时，需完成设备驱动程序，及设备的安装工作，实现PCI总线驱动。
- ❑ 根据功能要求，至少实现Open、Read、Write接口。
- ❑ 为了达到对MOXA卡的读操作，需要提供一个中断处理函数，当MOXA卡有数据到来时，中断服务程序调用此函数把数据放入缓冲。
- ❑ 为了达到对MOXA卡的写操作，需要写一个发送数据的函数，把要发送的数据写到MOXA卡的寄存器中并发送出去。

设备驱动中的PCI驱动(1/2)



31		16		15		0			
Device ID				Vendor ID				00h	
Status				Command				04h	
Class Code						Revision ID		08h	
BIST		Header Type		Latency Timer		Cache Line Size		0Ch	
Base Address Registers									10h
									14h
									18h
									1Ch
									20h
									24h
Cardbus CIS Pointer									28h
Subsystem ID				Subsystem Vendor ID				2Ch	
Expansion ROM Base Address									30h
Reserved									34h
Reserved									38h
Max_Lat		Min_Gnt		Interrupt Pin		Interrupt Line		3Ch	

Configuration Space

- ❑ PCI支持即插即用，不占用固定内存地址或I/O地址。
- ❑ PCI三个地址空间：配置空间、存储器地址、I/O地址。
- ❑ 由操作系统决定PCI映射基址。

设备驱动中的PCI驱动(2/2)

/* find the device with device & vendor ID */

pciFindDevice(.....)

/* get memory base address and IO base address */

pciConfigInLong(.....);

pciConfigInByte(.....);

/* enable mapped memory and IO addresses */

pciConfigOutWord(.....);

/* connect the interrupt handler to PCI */

pciIntConnect(.....);

/* enable the PCI interrupt */

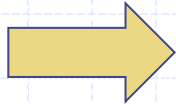
sysIntEnablePIC(.....);

序号	PCI功能接口	说明
1	pciConfigLibInit	pci库配置
2	pciFindDevice	发现设备
3	pciFindClass	发现类型
4	pciConfigInByte	读入配置
5	pciConfigInWord	读入配置
6	pciConfigInLong	读入配置
7	pciConfigOutByte	写入配置
8	pciConfigOutWord	写入配置
9	pciConfigOutLong	写入配置
10	pciIntConnect	连接中断

**VxWorks
PCI编程接口**

设备驱动中的字符设备驱动

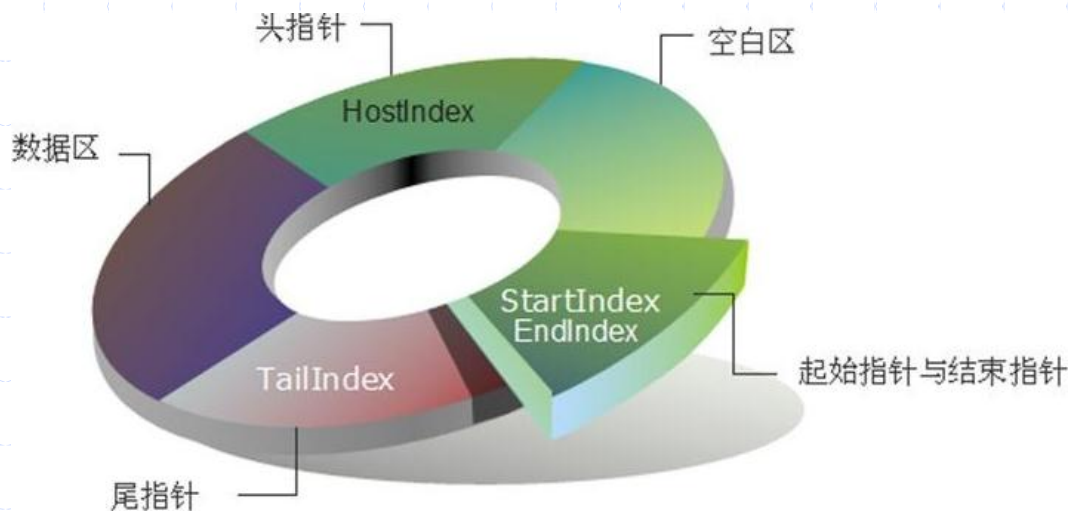
/* install the driver */
iosDrvInstall (.....);
/* create the device */
iosDevAdd(.....);



/* driver create function */
XXXCreate(.....);
/* driver open function */
XXXOpen(.....);
/* driver close function */
XXXClose(.....);
/* driver read function */
XXXRead(.....);
/* driver write function */
XXXWrite(.....);
/* driver ioctl function */
XXXIoctl(.....);

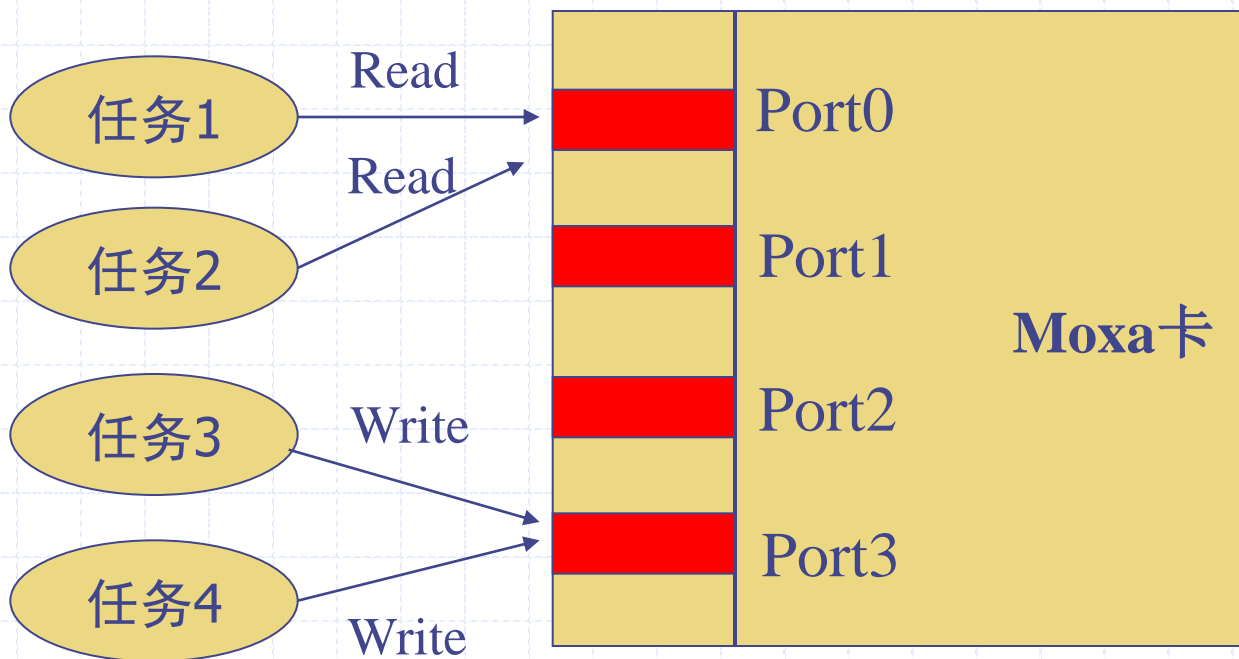
设备驱动中的缓冲管理

- 在设备驱动开发中，需解决低速外部设备和处理器速度之间的匹配。如果每次数据传送或接收时，系统任务都去操作，势必造成任务频繁切换，系统效率降低。
- 解决方法之一：环形数据缓冲
 - ✓ 当数据到达时，系统产生中断，中断服务程序将接收的数据不断写入环形缓冲区；
 - ✓ 应用程序调用read接口，从环形缓冲区读出数据。



设备驱动中的互斥设计

- 支持多任务并发对Moxa设备进行读、写操作。

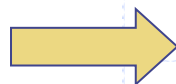


- 采用信号量等进行互斥操作。

Moxa设备驱动分析—数据结构

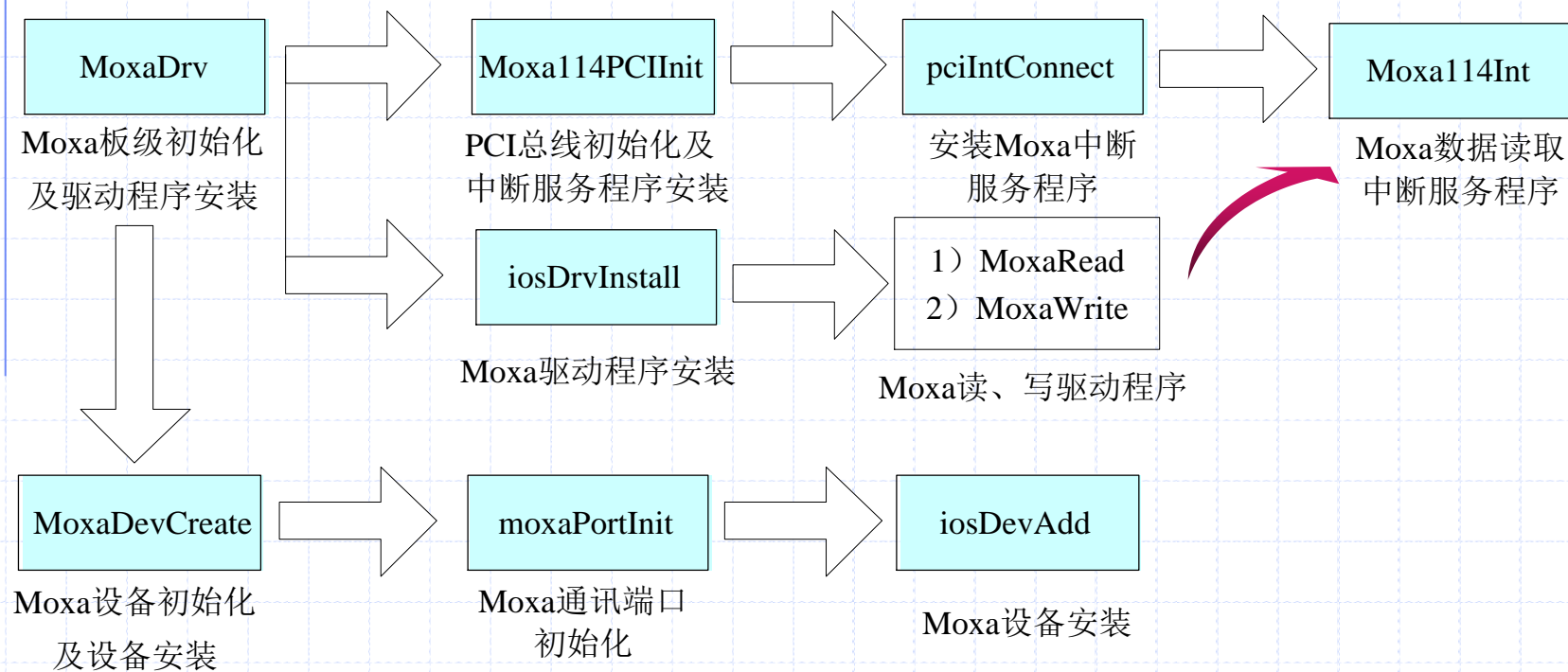
- Moxa有四个端口，每个端口有独立的读、写地址，因而需要作为四个设备看待，共享一个驱动程序。
- Moxa设备描述数据结构

```
typedef struct MOXADEV  
{  
    DEV_HDR DevHdr;  
    BOOL created;  
    int portNum;  
    RING_ID recvRngBuf;  
    SEMAPHORE wmuteSem;  
    SEMAPHORE rmuteSem;  
} MOXA_DEV;
```







- 设备节点
- 设备初始化标志
- 端口编号
- 环形缓冲
- 写缓冲区互斥量
- 读缓冲区互斥量

Moxa设备驱动分析—功能函数



课程大纲

-  外部设备及设备驱动概述
-  VxWorks设备驱动概述
-  VxWorks设备驱动代码分析
-  Linux设备驱动概述

Linux设备文件

- ❑ Linux抽象了对硬件的处理，所有的硬件设备都可以作为普通文件一样来看待。
- ❑ 可以使用和操作文件相同的、标准的系统调用接口来完成打开、关闭、读写和I/O控制操作
- ❑ 对用户来说，设备文件与普通文件并无区别。

Linux设备管理的三种模式（1/3）

□ 主、从设备管理模式：

- ✓ 主设备号，标识设备种类；从设备号，标识同一设备不同实例
- ✓ 采用**mknod**命令创建指定类型的设备文件，分配主、从设备号

```
[root@xsbase root]# mknod /dev/lp0 c 6 0
```

- ✓ 主设备号分配规则见：**Documentation/Devices.txt**
- ✓ 已安装设备驱动程序的主设备号，可从**/proc/devices**中获取
- ✓ 所有设备文件存储在**/dev**目录下

```
root@localhost:/etc/sysconfig
File Edit View Terminal Go Help
[root@localhost sysconfig]# cat /proc/devices
Character devices:
1 mem
2 pty
3 tty
4 ttyS
5 cua
6 lp
7 vcs
10 misc
13 input
29 fb
36 netlink
128 ptm
129 ptm
130 ptm
```

Block devices:

```
1 ramdisk
2 fd
8 sd
9 md
12 unnamed
14 unnamed
22 ide1
38 unnamed
39 unnamed
65 sd
66 sd
```

Linux设备管理的三种模式（2/3）

□ 主、从设备管理模式存在的问题

- ✓ 采用主、从设备号描述设备；
- ✓ 存在大量无用的设备文件，无法自动被创建。

□ 设备文件系统（**devfs**）管理模式：

- ✓ **devfs**从Linux内核的测试版本**2.3.46**开始引入
- ✓ **devfs**类似于**/proc**，是一种虚拟文件系统，内存空间占用小（**devfs: 72b / inode: 256b**）
- ✓ **devfs**是一种基于内核的动态设备文件系统，设备文件不需要手工采用**mknod**命令创建，在驱动程序安装时自动完成。

Linux设备管理的三种模式（3/3）

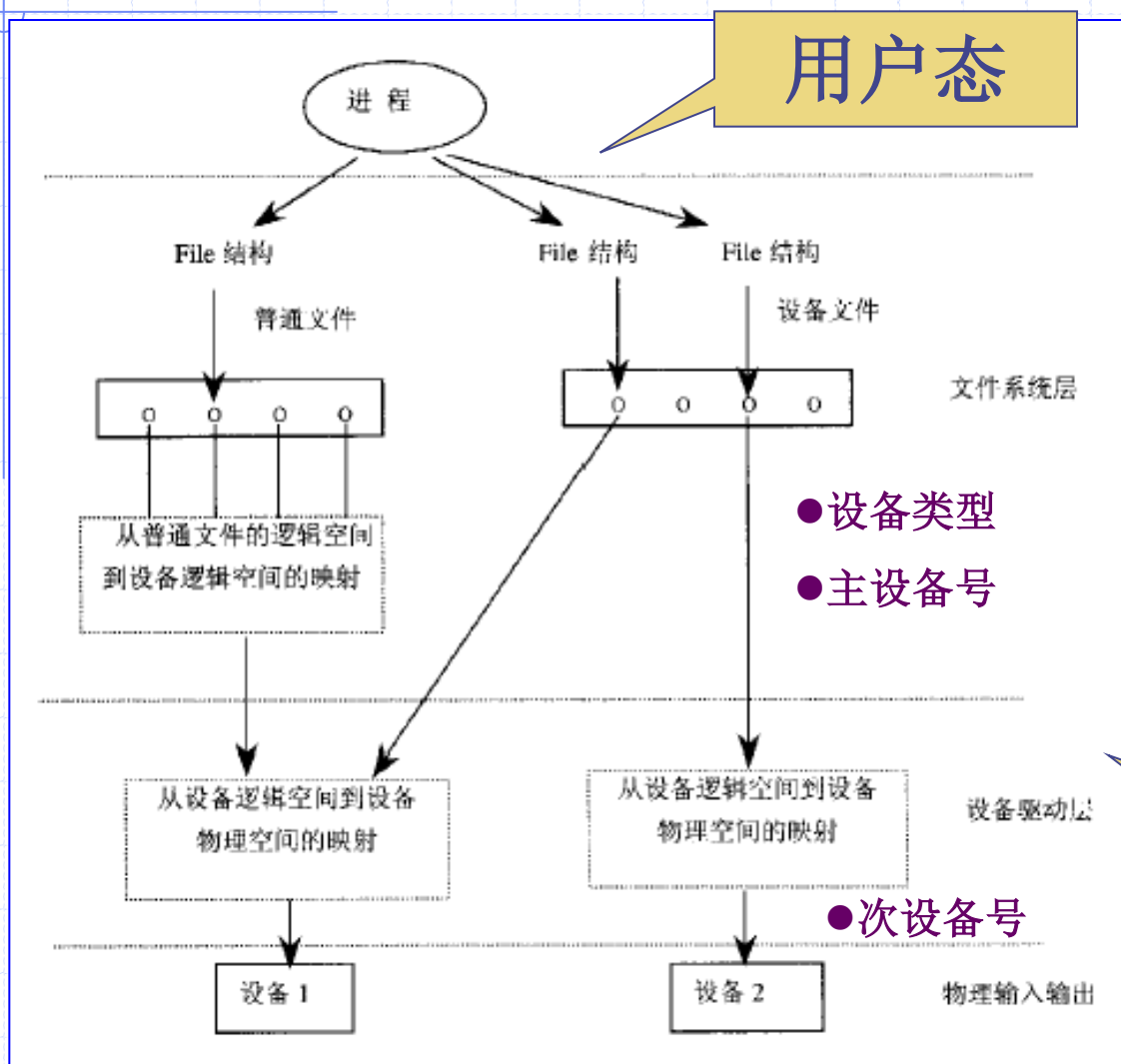
□ 设备文件系统（**devfs**）存在的问题

- ✓ 设备管理受主、从设备号（ **≤ 255** ）数量限制
- ✓ 不确定的设备映射问题
- ✓ 修改设备文件名字困难
- ✓ 内核内存消耗多

□ **sysfs**管理模式：

- ✓ **sysfs**是**linux2.6**内核引入的文件系统，是一种虚拟文件系统
- ✓ 对设备和总线进行管理
- ✓ 支持设备数量更多
- ✓ 支持设备热插拔
- ✓ 运行在用户空间

Linux设备操作执行过程



□ 设备操作功能调用

- ✓ 当应用程序对某个设备文件进行系统调用时，将从**用户态**进入到**内核态**；
- ✓ 内核根据该设备文件的**设备类型**和**主设备号**查询相应的设备驱动程序
- ✓ 由驱动程序判断该设备的**次设备号**，最终完成对相应设备的操作。

内核态

Linux设备驱动安装模式

- 将设备驱动以静态编译方式加入Linux内核。
- 将设备驱动模块，以动态加载方式加入内核。

Linux设备驱动特点

□ 内核编程模式

- ✓ 设备驱动是内核的一部分，必须使用标准的内核编程模式进行设计，如：内核内存分配、中断处理和等待队列。

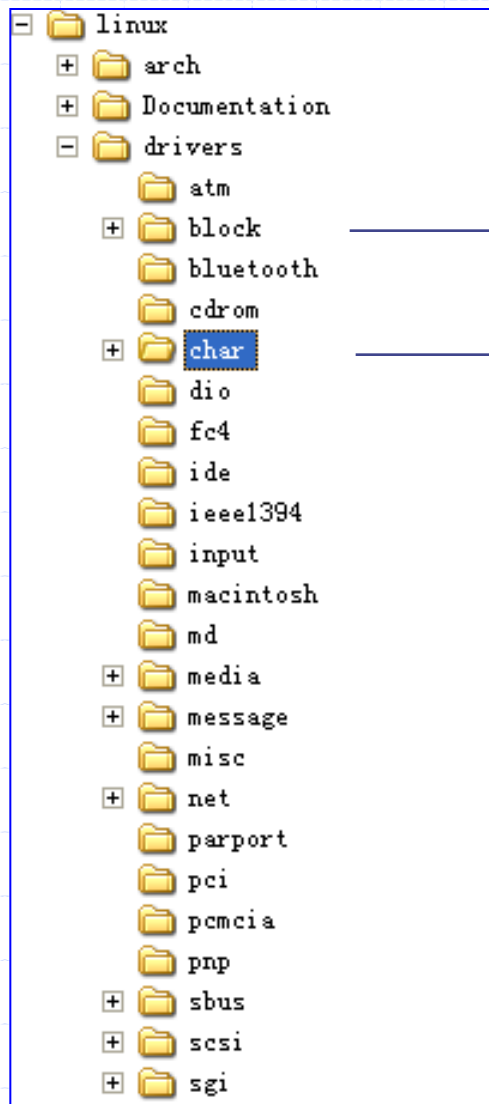
□ 内核、用户接口

- ✓ 设备驱动必须为内核或者其从属子系统提供标准接口；
- ✓ 设备驱动与应用程序交互接口，需完成用户空间/内核空间转换。

□ 动态可加载

- ✓ 设备驱动可以在需要的时候加载到内核，不再使用时被卸载，使内核能更有效地利用系统资源。

Linux设备驱动代码分布



块设备驱动公用部分

字符设备驱动公用部分

各种设备驱动源程序

Linux设备驱动与外界接口分类

- **Linux**的设备驱动程序与外界的接口，主要可以分成三个部分：
 - ✓ 与操作系统内核的接口：通过`include/linux/fs.h`中的`file_operations`数据结构完成。
 - ✓ 与系统引导的接口：完成对设备的初始化
 - 字符设备初始化在`drivers/char/mem.c`中的`chr_dev_init()`函数
 - 块设备初始化在`drivers/block/ll_rw_blk.c`中的`blk_dev_init()`函数
 - ✓ 与设备的接口：完成对设备的交互操作功能。

Linux设备驱动主要功能

□ Linux设备驱动程序代码结构大致可分为如下几个部分：

- ✓ 驱动程序的注册与注销
- ✓ 设备的打开与释放
- ✓ 设备的读写操作
- ✓ 设备的控制操作
- ✓ 设备的中断和轮询处理

字符设备驱动程序的注册、注销

□ 字符设备注册

✓ `int register_chrdev(unsigned int major, const char * name, struct file_operations *fops)`

- **major**: 主设备号, =0, 系统自动分配主设备号
- **name** : 设备名称
- **fops** : 设备操作接口

返回主设备号

```
/* Set up character device for user mode clients */  
i = register_chrdev(0, "pcmcia", &ds_fops);  
if (i == -EBUSY)  
    printk(KERN_NOTICE "unable to find a free device # for "  
            "Driver Services\n");  
else  
    major_dev = i;
```

```
static struct file_operations ds_fops = {  
    owner:      THIS_MODULE,  
    open:       ds_open,  
    release:    ds_release,  
    ioctl:      ds_ioctl,  
    read:       ds_read,  
    write:      ds_write,  
    poll:       ds_poll,  
};
```

pcmcia设备注册函数

□ 字符设备注销

✓ `unregister_chrdev()`

块设备驱动程序的注册、注销

□ 块设备注册

✓ `register_blkdev(unsigned int major, const char * name, struct block_device_operations *bdops)`

- `major`: 主设备号, =0, 系统自动分配主设备号
- `name`: 设备名称
- `bdops`: 设备操作接口

```
static struct block_device_operations floppy_fops = {  
    open:        floppy_open,  
    release:     floppy_release,  
    ioctl:       fd_ioctl,  
    check_media_change: check_floppy_change,  
    revalidate:   floppy_revalidate,  
};
```

```
if (register_blkdev(MAJOR_NR,"fd",&floppy_fops)) {  
    printk(KERN_ERR "Unable to get major %d for floppy\n",MAJOR_NR);  
    return -EBUSY;  
}
```

软驱设备注册函数

□ 块设备注销

✓ `unregister_blkdev()`

设备驱动程序的操作接口

□ 设备驱动程序的操作接口与文件系统的接口一致，在 **file** 和 **inode** 中均有 **file_operations** 数据结构。

✓ include/linux/fs.h

对打开文件的
处理函数

```
struct file {
    struct list_head f_list;
    struct dentry *f_dentry;
    struct vfsmount *f_vfsmnt;
    struct file_operations *f_op;
    atomic_t f_count;
    unsigned int f_flags;
    mode_t f_mode;
    loff_t f_pos;
    unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
    struct fown_struct f_owner;
    unsigned int f_uid, f_gid;
    int f_error;

    unsigned long f_version;

    /* needed for tty driver, and maybe others */
    void *private_data;

    /* preallocated helper kiobuf to speedup O_DIRECT */
    struct kiobuf *f_iobuf;
    long f_iobuf_lock;
} ? end file ? ;
```

```
struct inode {
    struct list_head i_hash;
    struct list_head i_list;
    struct list_head i_dentry;

    struct list_head i_dirty_buffers;
    struct list_head i_dirty_data_buffers;

    unsigned long i_ino;
    atomic_t i_count;
    kdev_t i_dev;
    umode_t i_mode;
    nlink_t i_nlink;
    uid_t i_uid;
    gid_t i_gid;
    kdev_t i_rdev;
    loff_t i_size;
    time_t i_atime;
    time_t i_mtime;
    time_t i_ctime;
    unsigned long i_blksize;
    unsigned long i_blocks;
    unsigned long i_version;
    struct semaphore i_sem;
    struct semaphore i_zombie;
    struct inode_operations *i_op;
    struct file_operations *i_fop;
    struct super_block *i_sb;
    wait_queue_head_t i_wait;
    struct file_lock *i_flock;
    struct address_space *i_mapping;
    struct address_space i_data;
    struct dquot *i_dquot[MAXQUOTAS];
    /* These three should probably be a union */
    struct list_head i_devices;
    struct pipe_inode_info *i_pipe;
    struct block_device *i_bdev;
    struct char_device *i_cdev;
```


字符设备驱动程序操作接口

□ 对于字符设备而言，**file_operations**是唯一的操作接口。

```
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 *, int datasync);
    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
} ? end file_operations ? ;
```

块设备驱动程序操作接口

- 对于块设备而言，`block_device_operations`是其主要操作接口。

设备注册函数



```
struct block_device_operations {  
    int (*open) (struct inode *, struct file *);  
    int (*release) (struct inode *, struct file *);  
    int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);  
    int (*check_media_change) (kdev_t);  
    int (*revalidate) (kdev_t);  
};
```

块设备
文件创建



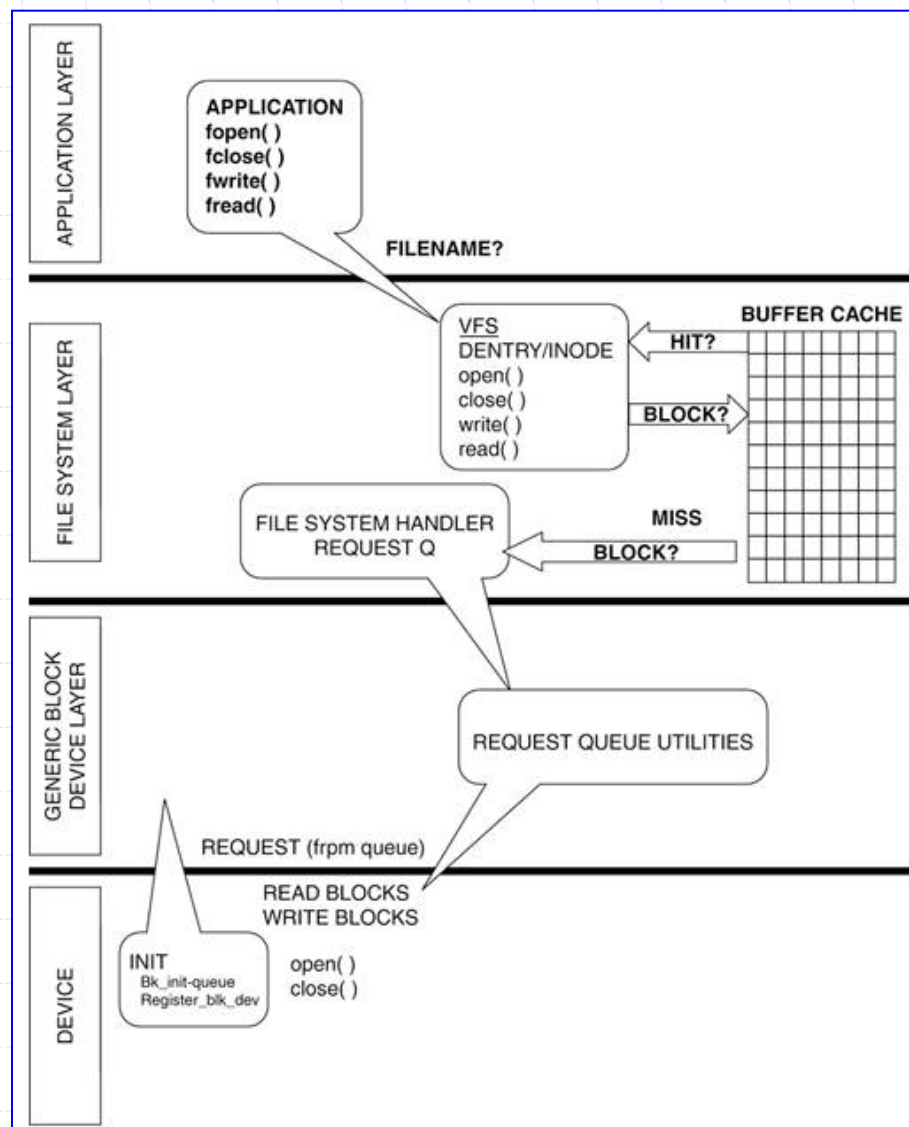
inode->i_fop



```
struct file_operations def_blk_fops = {  
    open:        blkdev_open,  
    release:     blkdev_close,  
    llseek:      block_llseek,  
    read:        generic_file_read,  
    write:       generic_file_write,  
    mmap:        generic_file_mmap,  
    fsync:       block_fsync,  
    ioctl:       blkdev_ioctl,  
};
```

块设备驱动程序的read/write操作原理

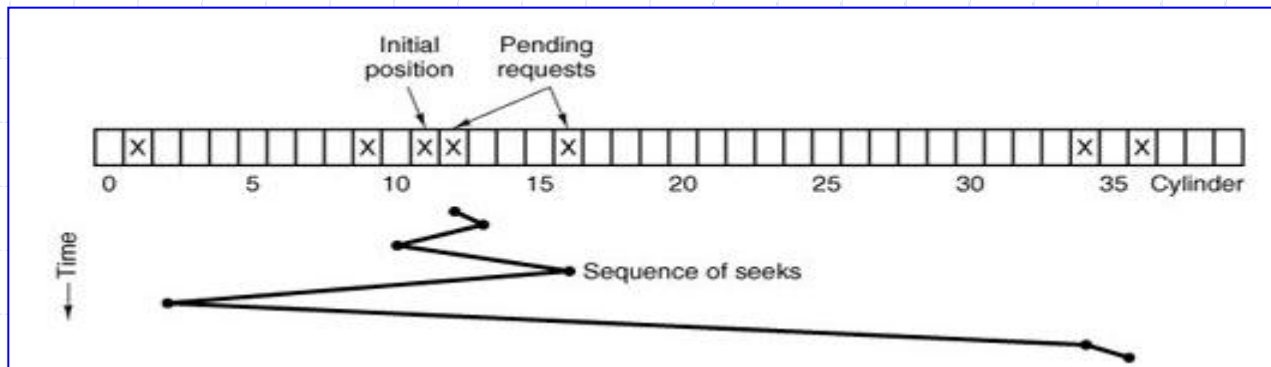
- 对于块设备而言，**read/write**操作不是从**block_device_operations**中获取。
- 在设备注册**register_blkdev**后，将进行请求队列初始化**blk_init_queue(request_queue_t *q, request_fn_proc *rfn)**
- 设备**read/write**请求，可采用最短查找优先算法（**SSF**）、或电梯算法（**Elevator Algorithm**）



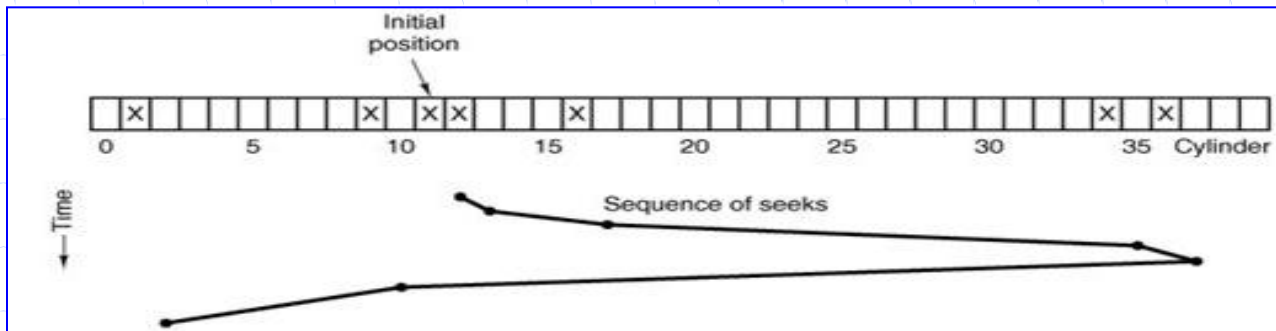
设备read/write调度优化算法

□ 设备read/write调度策略:

- ✓ First-Come, First-Served (FCFS)
- ✓ Shortest Seek First (SSF)



- ✓ elevator algorithm



设备驱动程序的轮询和中断处理

□ 设备驱动程序控制设备，通常采用两种方式：

- ✓ 轮询方式
- ✓ 中断处理方式

□ Linux系统中，在`/proc/interrupts`文件中可以看到设备驱动所对应的中断号及类型中断处理方式。

```
[root@redhat proc]# cat interrupts
          CPU0
 0:   70259356      XT-PIC  timer
 1:           4      XT-PIC  keyboard
 2:           0      XT-PIC  cascade
 5:   1487733      XT-PIC  usb-uhci, eth0
 8:           1      XT-PIC  rtc
 9:           0      XT-PIC  usb-uhci
10:           0      XT-PIC  ehci-hcd
11:           0      XT-PIC  usb-uhci
12:           6      XT-PIC  PS/2 Mouse
14:   214330      XT-PIC  ide0
NMI:           0
ERR:           0
```

设备驱动编程要点—内核编程模式（1/2）

- 可以使用系统调用，但不能使用libc库，如printf。

- 内存申请采用kmalloc

- ✓ 内核内存管理需求：对象分配/回收频繁；需要较小内存碎片

- ✓ slab算法思想：

- 预定义对象大小

- 按2的幂次方进行内存大小划分

- 对象缓存

- 对象复用

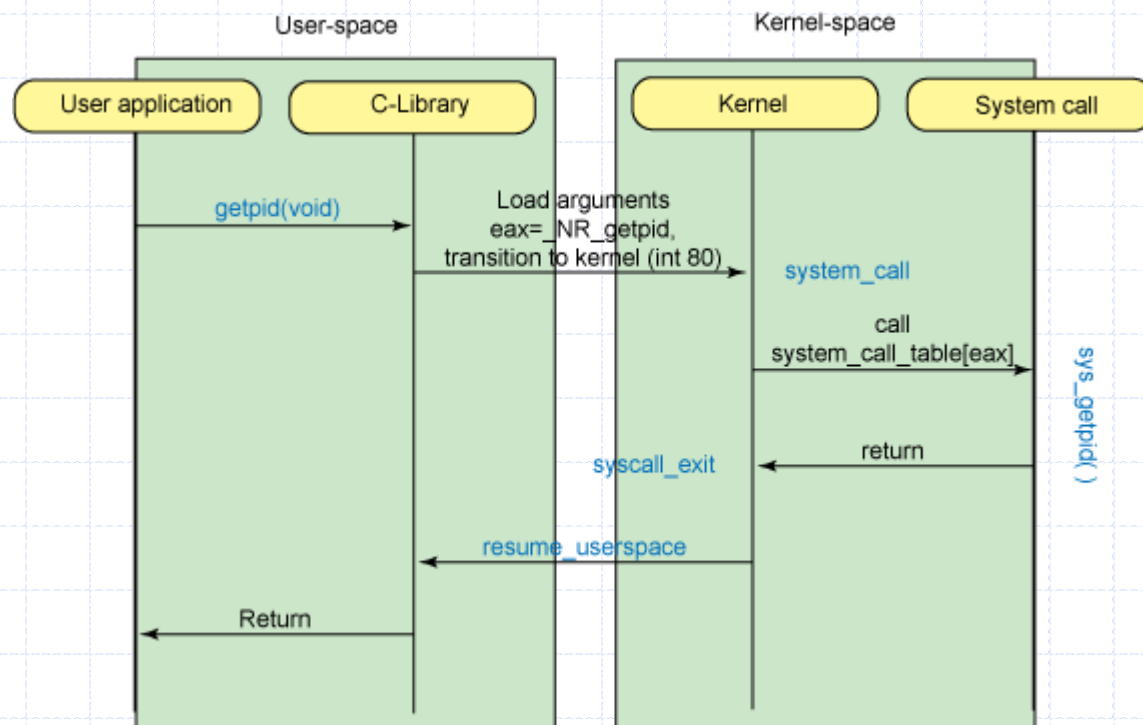
设备驱动编程要点—内核编程模式（2/2）

□ 内核编程中的并发处理：

- ✓ 多任务并发调用一个设备驱动程序
- ✓ 一个功能部分可能运行于多个上下文中（可重入）
- ✓ 一个设备驱动程序运行于多处理器上
- ✓ 设备驱动执行可被其他**ISR**中断

设备驱动编程要点—内核空间 vs 用户空间（1/3）

- ❑ 驱动程序运行于内核态，用户程序运行于用户态，如何将用户空间的数据传递入内核空间？



设备驱动编程要点一内核空间 vs 用户空间 (2/3)

□ 内核空间、用户空间数据传递方式

- ✓ 寄存器方式传递方式
- ✓ 寄存器传参数地址，调用内容放内存参数表中，可用于传递大量数据结构
- ✓ 堆栈传递方式

设备驱动编程要点一-内核空间 vs 用户空间 (3/3)

□ Linux提供的内核空间、用户空间数据传递函数 (include/asm/uaccess.h)

- ✓ `access_ok(type, address, size)`: 用户空间指针有效性判断
- ✓ `get_user(var, ptr)`: 移动int/long数据至内核空间
- ✓ `put_user(var, ptr)`: 移动int/long数据至用户空间
- ✓ `strncpy_from_user(char *dst, const char __user *src, long count)`: 将字符串从用户空间移动到内核空间中
- ✓ `copy_from_user(void *to, const void __user *from, unsigned long n)`: 由用户空间拷贝一块数据
- ✓ `copy_to_user(void *to, const void __user *from, unsigned long n)`: 向用户空间复制一块数据

设备驱动模块动态加载机制（1/4）

- Linux中的可加载模块(Module)是对内核功能的扩展

 - ✓ insmod

 - ✓ rmmod

- Linux模块载入内核后，它就成为内核代码的一部分

- 若某个模块空闲，用户便可将它卸载出内核

设备驱动模块动态加载机制（2/4）

□ 与模块相关的命令有：

- ✓ **lsmod**: 列出内核中已经安装的模块
- ✓ **insmod**: 安装模块到内核中
- ✓ **rmmod**: 将模块从内核中卸载
- ✓ **depmod**: 分析可载入模块的依赖关系

设备驱动模块动态加载机制（3/4）

□ 与模块相关的系统调用

系统调用	功能说明
<code>Sys_create_module</code>	为模块分配空间，将模块链入系统的模块链中
<code>Sys_init_module</code>	初始化模块，修正指针使模块正常工作
<code>Sys_delete_module</code>	从系统模块链中删除模块，释放内存空间
<code>Sys_get_kernel_syms</code>	将系统的所有符号表全部取出到用户空间

设备驱动模块动态加载机制（4/4）

□ 模块装入的两种方法

- ✓ 通过`insmod`命令，手动将模块载入内核
- ✓ 根据需要载入模块(**demand loaded module**): 当内核发现需要某个模块时，内核请求守护进程(**kerneld**)载入该模块

□ 卸载模块的两种方法

- ✓ 使用`rmmod`命令，手动卸载模块
- ✓ 自动卸载：每隔一定时间，**kerneld**调用 **sys_delete_module**，将它装入的且不再被使用的模块从系统中卸载。定时器的值在**kerneld**启动时设置

模块编程规范

□ Linux模块的编程规范（2.4系列内核）

```
/* 包含的头文件*/  
#include <linux/kernel.h>    /* 包含内核头文件 */  
#include <linux/module.h>    /* 包含模块头文件 */  
  
int init_module()             /* 模块初始化      */  
{  
    ...  
}  
void cleanup_module()         /* 模块卸载      */  
{  
    ...  
}
```

□ 模块编译的makefile文件中，必须添加编译参数

-c -D__KERNEL__ -DMODULE



谢谢!

